Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет Комп 'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «Аналіз швидкодійності платформ ASP NET та ASP NET Core»
за освітньою програмою **12 Інженерія програмного забезпечення**
зі спеціальності: **121 Інженерія програмного забезпечення**

Виконав: студент групи ПЗ2222: / Олександр ГЕТМАНЕНКО /

Керівник: / Олександр ІВАНОВ /

Нормоконтролер: / Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає
запозичень з праць інших авторів
без відповідних посилань.

Студент

Дніпро – 2024 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis

on the topic: «Performance analysis of ASP.NET and ASP.NET Core platforms»

according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2222: / Oleksandr HETMANENKO /

Scientific Supervisor: / Oleksandr IVANOV/

Normative controller: / Svitlana VOLKOVA /

Dnipro – 2024

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем
Кафедра: Комп’ютерні інформаційні технології
Рівень вищої освіти: магістр
Освітня програма: Інженерія програмного забезпечення
Спеціальність: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ
_________ Вадим ГОРЯЧКІН
_________ січня 2024 р.

ЗАВДАННЯ

На кваліфікаційну роботу ____Магістр______

студенту ГЕТМАНЕНКУ ОЛЕКСАНДРУ РУСЛАНОВИЧУ

1. Тема дипломної роботи: Аналіз швидкодійності платформ ASP NET та ASP NET Core.

   Керівник роботи: ОЛЕКСАНДР ПЕТРОВИЧ ІВАНОВ

   затверджені наказом 1196 ст від 05.12.2022 року

2. Строк подання студентом роботи 24.01.2024 року

3. Вихідні дані до дипломної роботи:

   пояснювальна записка, створений додаток для виконання досліджень.

4. Зміст пояснювальної записки (перелік питань до розробки):

   4.1. збір та аналіз вимог;

   4.2. методи дослідження та їх обґрунтування;

   4.3. проектування й розробка;

4.4. дослідження продуктивності платформ;

4.5. висновки.

5. Перелік демонстраційного матеріалу:

5.1. доповідь;

5.2. презентація;

5.3. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1	Вступ	03.03.23 – 14.04.23	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	15.04.23 – 22.08.23	10%
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	23.08.23 – 18.10.23	
4	Постановка задачі, технічне завдання	19.10.23 – 20.10.23	30%
5	Розробка інструментальних засобів дослідження	21.10.23 – 13.11.23	
6	Виконання досліджень	14.11.23 – 28.11.23	60%
7	Оформлення пояснювальної записки	29.11.23 – 15.01.24	
8	Розробка демонстраційних матеріалів	16.01.24 – 18.01.24	100%
9	Подання кваліфікаційної роботи до кафедри	19.01.24	
10	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	24.01.24	

Студент _____ / Олександр ГЕТМАНЕНКО /

Керівник роботи ______/ Олександр ІВАНОВ /

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра.

63с., 55 рис., 15 табл., 3 додатки, 7 джерел.

Мета дослідження: метою даного дослідження є аналіз та порівняння продуктивності й ефективності двох популярних платформ розробки веб-додатків – ASP.NET та ASP.NET Core. Основний акцент робиться на оптимізації обробки даних та використанні різних структур даних для забезпечення швидкодії та ефективності роботи веб-додатків.

Об'єкт дослідження: об'єктом дослідження є колекції даних (словник, список, черга) та їх робота на платформах ASP.NET та ASP.NET Core.

Предмет дослідження: предметом дослідження є реалізація колекцій даних на основі структур даних веб-додатків на платформах ASP.NET та ASP.NET Core та швидкість обробки асинхронних й синхронних запитів у вказаних платформах. Досліджується взаємодія з цими колекціями в різних умовах виконання.

Пояснювальна записка складається зі вступу, 4 розділів, висновків, бібліографічного списку та 1 додатку:

– вступ – опис загальної суті проблеми дослідження, 1 сторінка;
– збір та аналіз вимог – опис поточного стану проблеми відносно інших досліджень, постановка задачі до дослідження, 4 сторінки;
– методи дослідження та їх обґрунтування – опис використаних методів дослідження, 3 сторінки;
– проектування й розробка – описує важливі кроки розробки та планування проекту, 18 сторінок;
– дослідження продуктивності платформ – опис ходу досліджень та їх результатів, 31 сторінки;
– додатки містять технічне завдання.

Ключові слова: ASP.NET Core, ASP.NET, Efficiency, Data Collections, Synchronous, Asynchronous, Design Patterns, Data Volume, Analysis, C#.

ЗМІСТ

ВСТУП 3
1 ЗБІР ТА АНАЛІЗ ВИМОГ 4
1.1 Аналіз сучасного стану дослідження 4
1.2 Аналіз стану програмно-апаратного забезпечення, яке потребує вдосконалення під час вирішення проблем дослідження 5
1.3 Постановка задачі 5
2 МЕТОДИ ДОСЛІДЖЕННЯ ТА ЇХ ОБҐРУНТУВАННЯ 8
2.1 Вибір методики дослідження 8
2.2 Методи дослідження 8
2.2.1 Вимірювання часу виконання операцій CRUD. 8
2.2.2 Оцінка обсягу використаної пам'яті процесу для операції.............. 8
2.2.3 Оцінка обсягу використаної пам'яті збірника сміття при виконанні операції.. 8
2.2.4 Тести часу виконання 8
2.2.5 Оцінка з використанням графіків лінії тренду. 8
2.3 Порівняльні оцінки та висновки 9
3 ПРОЕКТУВАННЯ Й РОЗРОБКА 11
3.1 Формалізація задачі 11
3.2 Базова архітектура системи 14
3.3 Внутрішнє проектування 15
3.3.1 Вибір мови програмування 15
3.3.2 Технологічна платформа 15
3.3.3 Ієрархія та взаємодія класів системи 16
3.3.4 Використані принципи проектування 22
3.3.5 Використані шаблони проектування 23
3.4 Тестування та налагодження програми 25
3.4.1 Метод тестування граничними умовами 25
3.4.2 Налагодження програми 26
4 ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ПЛАТФОРМ................................... 29

4.1 Підготовка до експерименту 29
4.2 Проведення експерименту з дослідження колекції «Словник» 30
4.3 Проведення експерименту з дослідження колекції «Список» 39
4.4 Проведення експерименту з дослідження колекції «Черга» 45
4.5 Проведення експерименту з дослідження синхронного режиму відправки запиту 53
4.6 Проведення експерименту з дослідження асинхронного режиму відправки запиту 53
4.7 Результати експериментів 53
ВИСНОВКИ 60
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 62
ДОДАТКИ 63

ВСТУП

Сфера веб-розробки продовжує постійно розвиватись, з'являється все більше технологій веб-розробки, однією з таких є .NET, зокрема ASP.NET та ASP.NET CORE, враховуючі, що ці дві технології є новими, особливо ASP.NET CORE, порівняння їх швидкодійності є надзвичайно важливим для розробників та кінцевих користувачів.

Швидкість роботи будь-якого додатку є важливою характеристикою програмного забезпечення. Якщо веб-додаток працює повільно, це зменшує ефективність роботи з ним й збільшує ймовірність втрати користувачів. Теж саме стосується й сфери розробки, в залежності від обраної технології розробник може витрати багато часу для створення свого додатку, зручність написання коду, швидкість роботи обраної платформи та її можливості є одними з ключових факторів розробки.

Порівняння швидкодійності платформ ASP.NET та ASP.NET CORE може допомогти розробникам обрати, яка з цих платформ є більш оптимальною для виконання поставленого завдання.

Окрім того, зростаюча кількість веб-додатків, що працюють в режимі реального часу, підвищує значимість швидкодійності. У таких веб-додатках необхідно забезпечити максимальну швидкість відгуку системи на запити користувачів.

1 ЗБІР ТА АНАЛІЗ ВИМОГ

1.1 Аналіз сучасного стану дослідження

Останні дослідження в галузі швидкодії платформ ASP.NET та ASP.NET CORE виконувалися з метою зрозуміти, наскільки нові версії кращі за продуктивністю від старих.

Компанія Microsoft, що стоїть за розробкою ASP.NET CORE, провела дослідження, спрямоване на порівняння швидкодії нової версії ASP.NET CORE з попередніми ітераціями платформи[1]. Також компанія вивчала продуктивність більш ранніх версій платформи ASP.NET[2].

Результати дослідження платформи ASP.NET CORE показали помітне покращення швидкодії порівняно з її попередніми ітераціями. Щодо дослідження платформи ASP.NET, було обрано версію фреймворку 4.8 та для порівняння обрано платформу ASP.NET CORE з версією 3.1. В результаті здебільшого платформи мали майже однакову продуктивність по часу виконання та за обсягом використаної пам’яті, проте в деяких випадках платформа ASP.NET CORE показувала набагато більшу оптимізацію використання пам’яті та менші затрати часу.

При розробці проектів враховуючи мету й завдання роботи проекту обирають оптимальну платформу для його реалізації. Нове в цьому випадку не завжди краще, оскільки проекти зазвичай не пишуться з самого початку, а надходять на оновлення та розширення зі старими платформами, зазвичай це ASP.NET Framework 4.7.2 або 4.8. Враховуючи частоту виходу версії ASP.NET CORE та швидкий темп розвитку платформи виникає, необхідність постійного аналізу нових версій порівняно з попередніми ітераціями та порівнянні її з версією платформи ASP.NET.

Незважаючи на наявність багатьох матеріалів, що описують швидкодію окремих версій платформ, за межами основних джерел, присвячених даній темі, відсутнє достатнє порівняння продуктивності між платформою ASP.NET та новими версіями платформи ASP.NET CORE.

1.2 Аналіз стану програмно-апаратного забезпечення, яке потребує вдосконалення під час вирішення проблем дослідження

Для більш глибокого розуміння продуктивності платформи ASP.NET та її нових версій, додатковий аналіз програмно-апаратного забезпечення є важливою складовою. Основним об'єктом дослідження стануть операції CRUD (створення, читання, оновлення, видалення) з колекціями даних, такими як списки, черги та словники.

Переважно, у фокусі дослідження буде час виконання операцій CRUD для кожного типу колекції. Також розглядається обсяг зайнятої пам'яті процесом перед тестуванням та після його завершення. Окрема увага приділяється аналізу обсягу пам'яті, який використовує збірник сміття.

Додатково, проводяться тести часу виконання в для простого запиту на отримання елементів порожнього списку. Тестування включає в себе вимірювання часу обробки для послідовно відправлених запитів та запитів які відправляються серверу одночасно.

Цей аналіз дозволить отримати глибоке розуміння продуктивності платформи в різних сценаріях використання та допоможе виявити можливості для вдосконалення як програмного, так і апаратного забезпечення. Також враховуючи сучасну тенденцію розміщення серверів у хмарних сховищах дослідження дозволить встановити найважливіші аспекти сховища й правильно розподілити ресурси, що відповідно може підвищити продуктивність серверу й зменшити його вартість.

У контексті вищезгаданого дослідження це стане додатковим кроком у напрямку комплексного оцінювання ефективності та оптимізації платформ ASP.NET та ASP.NET CORE.

1.3 Постановка задачі

Мета роботи: дослідити порівняльну продуктивність платформ ASP.NET та ASP.NET CORE в контексті операцій CRUD над різними типами колекцій (список, черга, словник) з урахуванням впливу обсягу даних на їх продуктивність.

Завдання:

1) Розробка тестових контролерів: розробити тестові контролери для кожного типу колекції на платформах ASP.NET та ASP.NET CORE. Ці контролери повинні включати методи для виконання операцій додавання, оновлення, видалення та отримання всіх елементів з колекцій.
2) Використання інтерфейсів та репозиторіїв: використовувати інтерфейси та репозиторії (IListRepository, IQueueRepository, IDictionaryRepository) для забезпечення гнучкості й розширюваності системи та можливості легкої зміни представлених реалізацій колекцій.
3) Вимірювання часу виконання та обсягу використаної пам'яті: реалізувати методи контролерів для вимірювання часу виконання операцій та обсягу використаної пам'яті для операцій додавання, оновлення, видалення в різних сценаріях роботи з колекціями. Методи повинні мати однакові алгоритми на обох платформах.
4) Тести часу виконання для порожньої колекції: провести тест часу виконання для простих запитів на отримання елементів порожніх колекцій. Вимірювати час обробки для послідовно відправлених запитів та запитів, які відправляються серверу одночасно.
5) Збір та структурування даних: збирати дані про вимірювання продуктивності у вигляді структурованого об'єкта (JSON формат) для полегшення подальшого аналізу та порівняння результатів.
6) Дослідження на різних обсягах даних: провести дослідження на різних обсягах даних для визначення впливу обсягу даних на продуктивність системи.

Висновки до розділу 1

У результаті аналізу сучасного стану досліджень у галузі продуктивності платформ ASP.NET та ASP.NET CORE, виявлено, що нові версії ASP.NET CORE демонструють помітне покращення швидкодії порівняно з попередніми ітераціями. Однак, порівнюючи їх з версією ASP.NET, зауважено, що більшість платформ мають майже однакову продуктивність за часом виконання та обсягом використаної пам'яті. Тим не менш, ASP.NET CORE виявляє більшу оптимізацію використання пам'яті та менші затрати часу у деяких сценаріях.

Очевидно, що при виборі оптимальної платформи для проектів важливо враховувати їхню сумісність з існуючими системами та потребами проекту. Оновлення до нових версій не завжди гарантує автоматичне поліпшення продуктивності, оскільки проекти часто вже розроблені на старих версіях та потребують адаптації до нових платформ.

Аналіз програмно-апаратного забезпечення показав, що для глибокого розуміння продуктивності платформи ASP.NET та її нових версій, важливо досліджувати операції CRUD з колекціями даних. Фокус дослідження на часі виконання операцій CRUD для кожного типу колекції, а також на обсягу використаної пам'яті та аналізі роботи збірника сміття, є ключовим для повного розуміння їхньої продуктивності.

Метою подальших досліджень є порівняння продуктивності платформ ASP.NET та ASP.NET CORE у контексті операцій CRUD над різними типами колекцій з урахуванням впливу обсягу даних на їхню продуктивність. З цією метою визначено ряд завдань, включаючи розробку тестових контролерів для кожного типу колекції, використання інтерфейсів та репозиторіїв для забезпечення гнучкості системи, вимірювання часу виконання та обсягу використаної пам'яті, а також дослідження впливу обсягу даних на продуктивність системи. Важливо збирати та структурувати дані про продуктивність для подальшого аналізу та порівняння результатів дослідження.

2 МЕТОДИ ДОСЛІДЖЕННЯ ТА ЇХ ОБҐРУНТУВАННЯ

2.1 Вибір методики дослідження

Враховуючи важливість порівняльного аналізу продуктивності платформ, дослідження буде здійснюватися експериментальним методом. Основною метою є визначення продуктивності кожної платформи для операцій CRUD над різними типами колекцій.

2.2 Методи дослідження

2.2.1 Вимірювання часу виконання операцій CRUD.

Використовуючи інструменти надані мовою C# зафіксувати час затрачений на виконання операції.

2.2.2 Оцінка обсягу використаної пам'яті процесу для операції.

Перед тестом вимірюємо обсяг пам'яті процесу, в якому відбуватиметься операція, та після неї. Це допомагає встановити, скільки пам'яті використовується під час виконання операції.

2.2.3 Оцінка обсягу використаної пам'яті збірника сміття при виконанні операції.

Вимірюємо обсяг пам'яті збірника сміття до та після виконання операції для розуміння, скільки пам'яті використовується при обробці сміття.

2.2.4 Тести часу виконання

Вимірювання часу обробки запитів до порожньої колекції для послідовно відправлених запитів та одночасно відправлених запитів.

2.2.5 Оцінка з використанням графіків лінії тренду.

Для кожного виду операцій CRUD (створення, читання, оновлення, видалення) у кожного типу колекцій будується графік залежності часу виконання та використаної пам'яті процесу й збірника сміття відносно запиту який виконується.

На графіках буде відображено лінію тренду, яка демонструватиме загальну тенденцію ефективності платформ для різних операцій та типів даних при навантаженні системи.

Графіки допоможуть візуалізувати залежність між завантаженістю системи, яка відображатиметься у вигляді номеру запиту та ефективністю виконання операцій відносно використання пам’яті процесу й збірника сміття та часу виконання, що дозволить визначити загальні тенденції експерименту відносно кожного з показників, що відповідно дозволить порівняти дві платформи за цими показниками.

2.3 Порівняльні оцінки та висновки

Результати експерименту дозволять провести порівняльні оцінки продуктивності кожної платформи. На основі цих результатів буде зроблено висновок щодо ефективності платформ для операцій CRUD над колекціями даних різного обсягу.

Висновки до розділу 2

Дослідження продуктивності платформ буде спрямоване на об’єктивну оцінку ефективності ASP.NET та ASP.NET CORE, враховуючи їхню працездатність у реальних умовах та потенційний вплив обсягу даних та кількості запитів на продуктивність системи.

3 ПРОЕКТУВАННЯ Й РОЗРОБКА

3.1 Формалізація задачі

На етапі зовнішнього проектування формалізуємо задачу використовуючи діаграму прецедентів. Ця діаграма моделює взаємодію між користувачами та системою, показуючи різні сценарії взаємодії. Прецеденти – це окремі функціональні можливості системи або послуги, які виконуються користувачами або іншими системами. Кожен прецедент відображається у вигляді овалу на діаграмі, а взаємодія між користувачем та системою показується стрілками або лініями. Це дозволяє краще зрозуміти, як користувачі будуть спілкуватися з системою та які функції вони зможуть використовувати.

Для подальшого опису системи варто прояснити деякі аспекти її роботи, а саме інструмент веб-розробки Swagger та принцип його використання.

Swagger – інструмент, що використовують програмісти у веб-розробці. Цей інструмент призначений для документування створеного API веб-сервісу та надає зручний і простий інтерфейс для взаємодії користувача з кінцевими точками API. Також цей інструмент автоматично аналізує коментарі, які програмісти записують у своєму коді, що дозволяє більш детально пояснити процес роботи програмного коду, або прояснити іншим розробникам складні етапи реалізації програмного забезпечення. Також, аналізуючи код, цей інструмент створює пояснення щодо вхідних та вихідних даних кінцевої точки й надає моделі даних з якими вона працює, що дозволяє іншим програмістам швидко підлаштовувати сервіси, що працюють з цим API до приймання чи відправки даних вказаного формату.

Все це дозволяє пришвидшити розробку програмного забезпечення та швидко інтегрувати сервіс у роботу інших проектів.

На діаграмах рис. 3.1 – 3.3 представлені діаграма прецедентів, які описує взаємодію користувача з інтерфейсом swagger для платформ ASP.NET та ASP.NET Core при роботі з колекціями.

На діаграмі рис. 3.4 представлена взаємодія з проектом для виконання тестів.

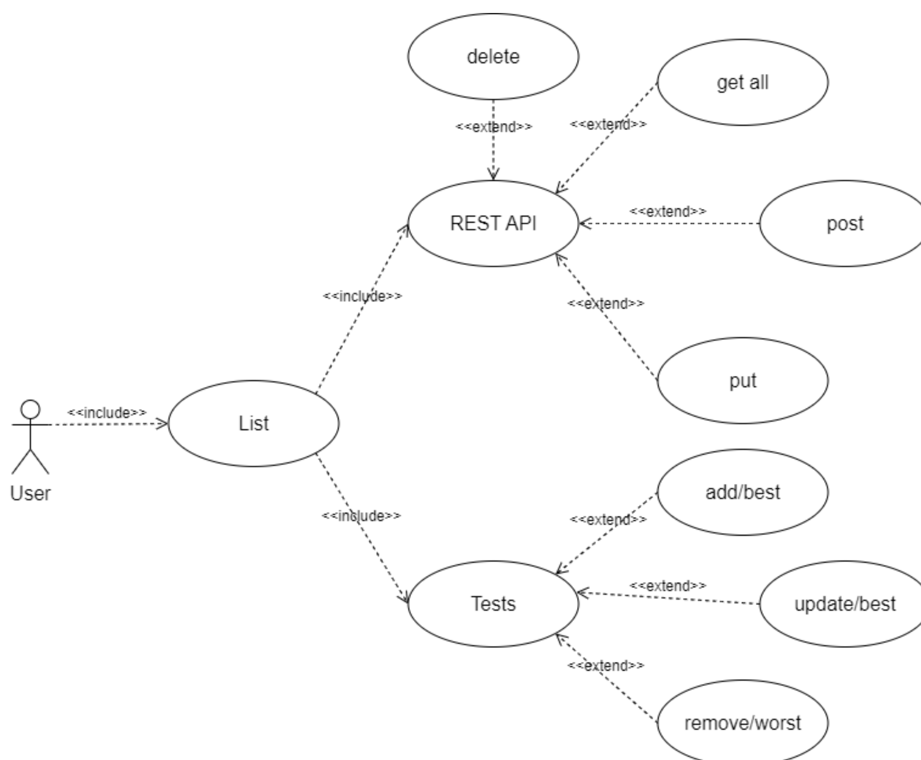

Рисунок 3.1 – Діаграма прецедентів, робота користувача з сервісами списку

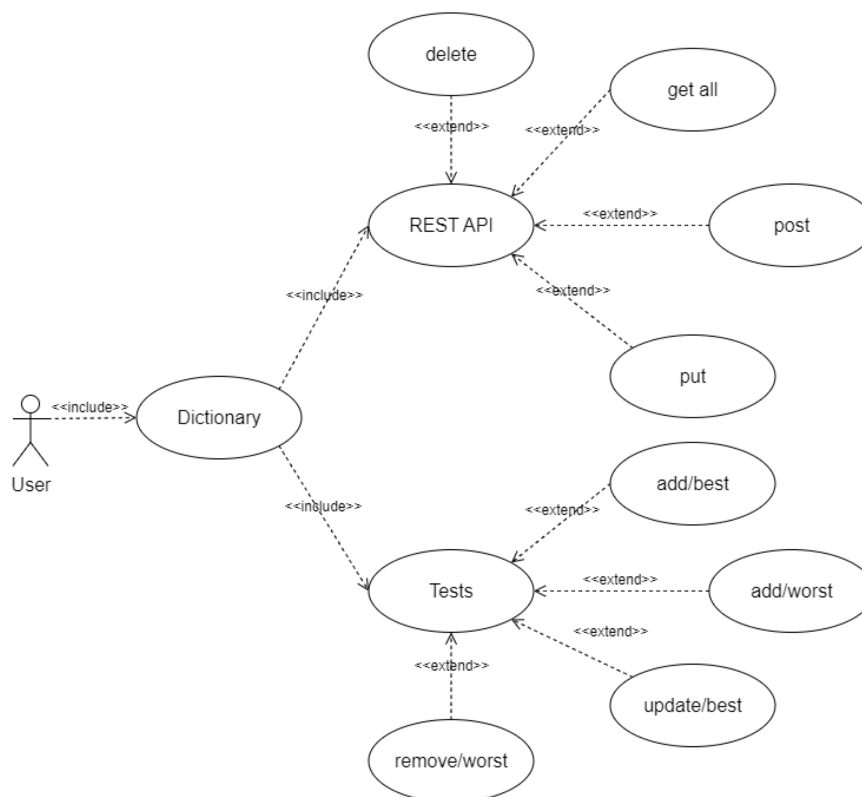

Рисунок 3.2 – Діаграма прецедентів, робота користувача з сервісами словника

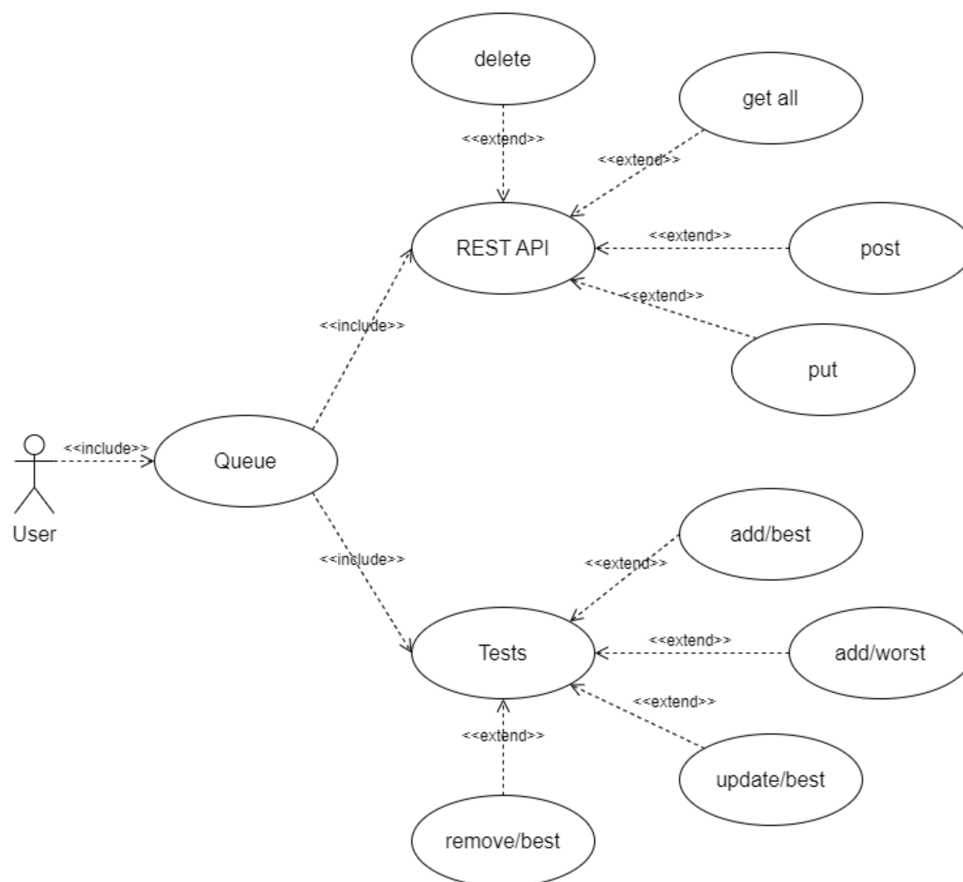

Рисунок 3.3 – Діаграма прецедентів, робота користувача з сервісами черги

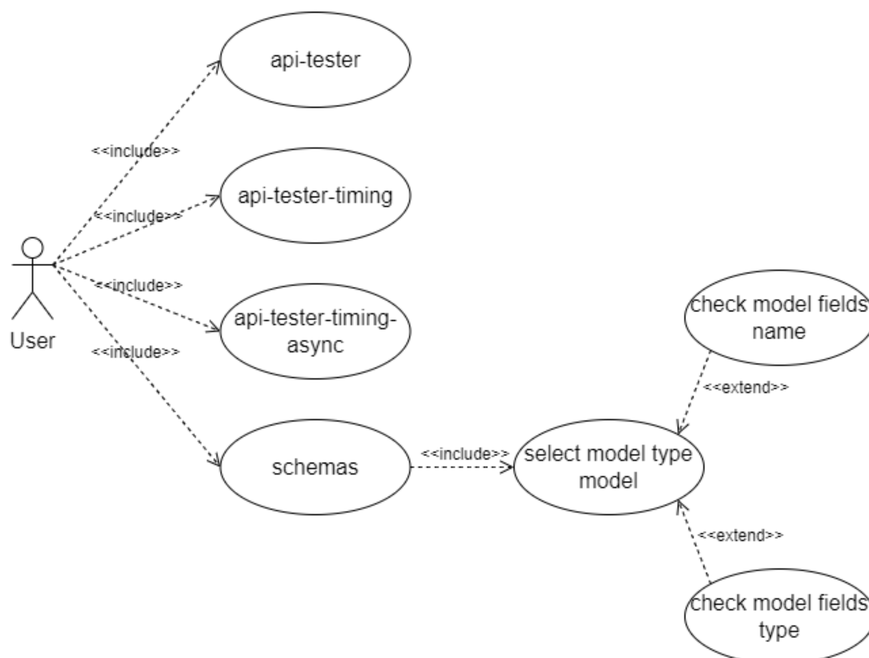

Рисунок 3.4 – Діаграма прецедентів, робота користувача з сервісами проекту тестування

3.2 Базова архітектура системи

Архітектурний патерн Model-View-Controller (MVC) є одним із найпоширеніших у розробці програмного забезпечення. У ньому програма розбивається на три основні частини, що дозволяє відокремлювати логіку застосунку, його представлення та управління даними.

Основні компоненти патерну MVC:

1. Модель (Model): це компонент, що відповідає за управління даними, бізнес-логікою та даними застосунку. У проекті цю роль відіграватимуть інтерфейси та класи, які є контейнерами даних кожної з колекцій та надають доступ до даних всередині нього.
2. Представлення (View): це частина, що відповідає за відображення даних користувачеві та отримання вказівок від нього. У веб-застосунку це може бути клієнтська сторона, яка рендерить сторінки або взаємодіє з API. У проекті представлення буде відображене не прямо, а через інтерфейс Swagger.
3. Контролер (Controller): це шар між моделлю та представленням. Контролери приймають запити, викликають методи сервісів, формують відповідь та відправляють її до користувача. Рівень контролеру є основним рівнем, який реалізовує обробку запитів користувачів та взаємодію з сервісами проекту для того, щоб запит оброблявся за визначеним програмістом сценарієм.

Особливості цієї архітектури:

Переваги:

- розділення відповідальності: кожна частина системи має свою відповідальність, що полегшує розробку, тестування та зміни;
- масштабованість: забезпечує зручність у розширенні та зміні окремих частин програми;
- відокремлення логіки і представлення: це дозволяє змінювати вигляд без зміни бізнес-логіки.

Недоліки:

- складність: для менших проектів це може бути зайвою складністю;
- потреба у координації: велика система, розбита на багато частин, потребує гарної координації між ними.

3.3 Внутрішнє проектування

3.3.1 Вибір мови програмування

Для розробки проекту було обрано мову програмування C#.

Широкі можливості розвитку: C# є однією з основних мов програмування для платформ .NET і володіє широким спектром можливостей для розробки різноманітних застосунків.

Підтримка технологій .NET: мова C# є основною мовою програмування для платформ .NET, включаючи ASP.NET та ASP.NET Core. Її вибір у цьому контексті є логічним і забезпечує сумісність з екосистемою цих технологій.

Широкі ресурси та спільнота: C# має велику спільноту розробників та багато ресурсів для навчання та розвитку. Це забезпечує доступ до великої кількості матеріалів, документації та підтримки.

Модерна та ефективна мова: C# є сучасною мовою програмування з багатьма сучасними функціями, такими як асинхронність, властивості, делегати та LINQ, що сприяє покращенню продуктивності та зручності розробки.

3.3.2 Технологічна платформа

Для розробки проекту було обрано середовище розробки Visual Studio.

Інтегроване середовище розробки (IDE): Visual Studio надає повноцінне та потужне середовище для розробки .NET-проектів. Його функціонал, такий як підтримка мови C#, налагоджувачі, інструменти для тестування, дебагу, аналізу коду та інші, робить його ідеальним вибором для розробки проектів в середовищі .NET.

Підтримка ASP.NET та ASP.NET Core: Visual Studio має широкі можливості для розробки як традиційних ASP.NET, так і новітніх ASP.NET Core.

Воно забезпечує набір інструментів для розробки, тестування та відлагодження проектів обох платформ.

3.3.3 Ієрархія та взаємодія класів системи

Для ілюстрації взаємодії класів у проектах, розробимо діаграми класів проектів для виконання запитів до проектів на обох платформах (рис. 3.5 – 3.11).

Важливо зауважити, що діаграми класів для обох проектів ідентичні, тому дублювання інформації на цьому етапі є зайвим. Буде створено єдиний варіант діаграм для ілюстрації спільної структури класів обох платформ.

Крім того, приведемо діаграму класів (рис. 3.5), яка відображає взаємодію класів у проекті, відповідальному за відправку запитів до обох платформ, а також для проведення тестування швидкості відповіді від них.

Створені діаграми класів допоможуть візуалізувати взаємодію із системою та об'єднати основні класи, які забезпечують функціональність проектів.

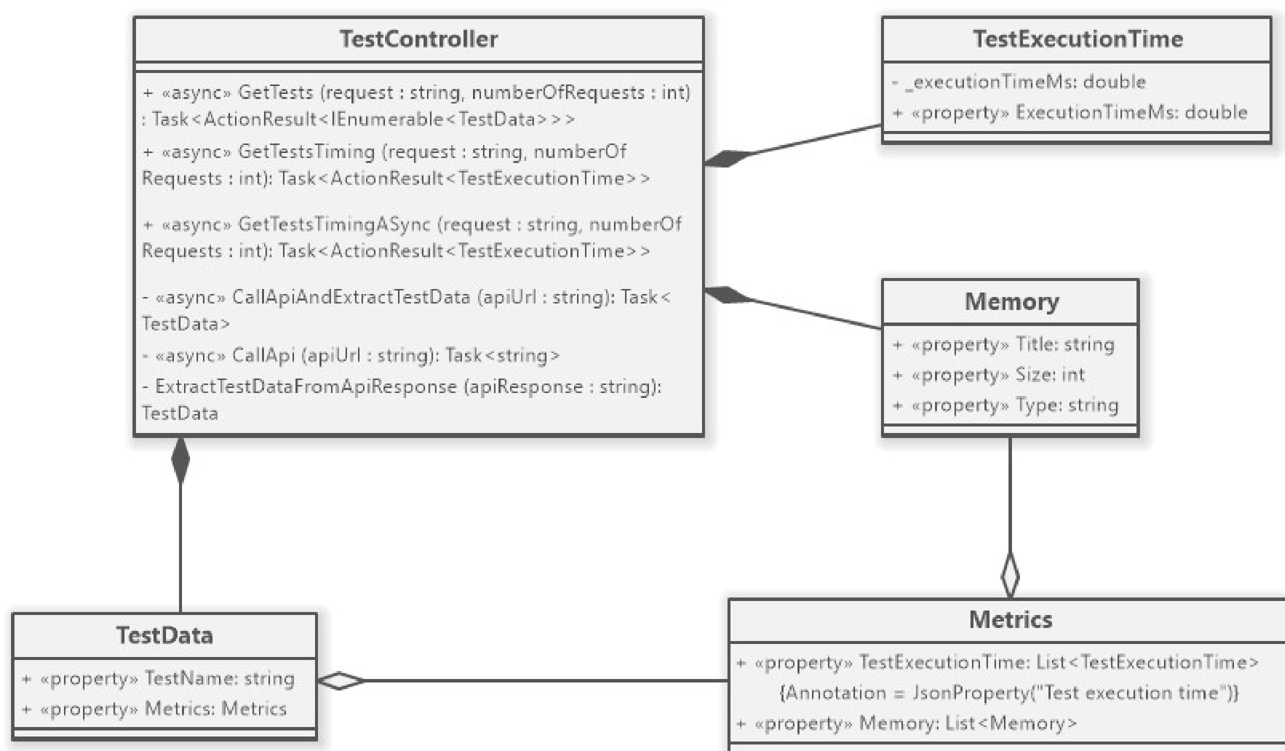

Рисунок 3.5 – Діаграма класів, проекту для виконання запитів до інших проектів

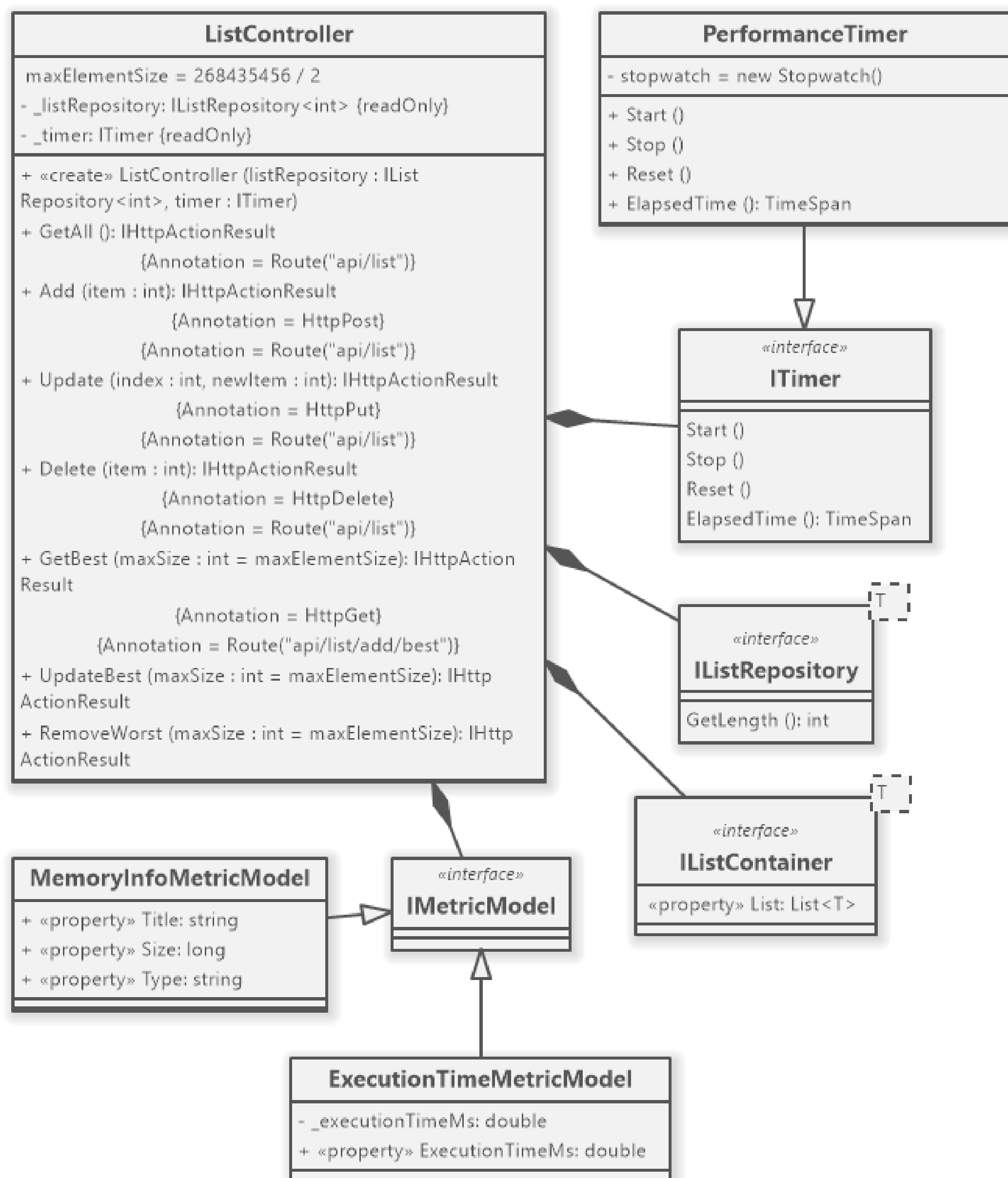

Рисунок 3.6 – Діаграма класів, взаємодія класів сервісів, моделей та даних з контролером колекції «список»

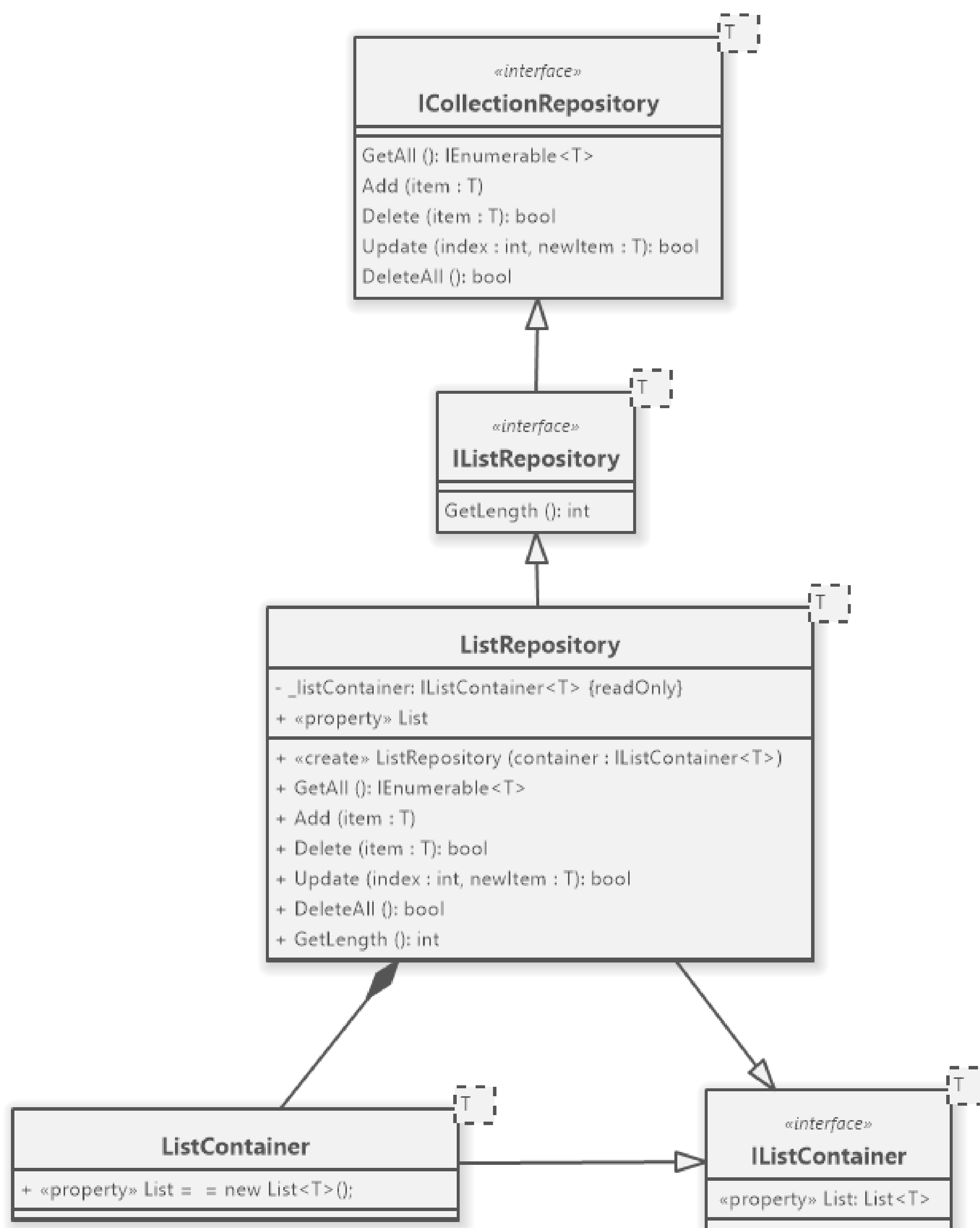

Рисунок 3.7 – Діаграма класів, яка відображає структуру класів рівня даних для колекції «список»

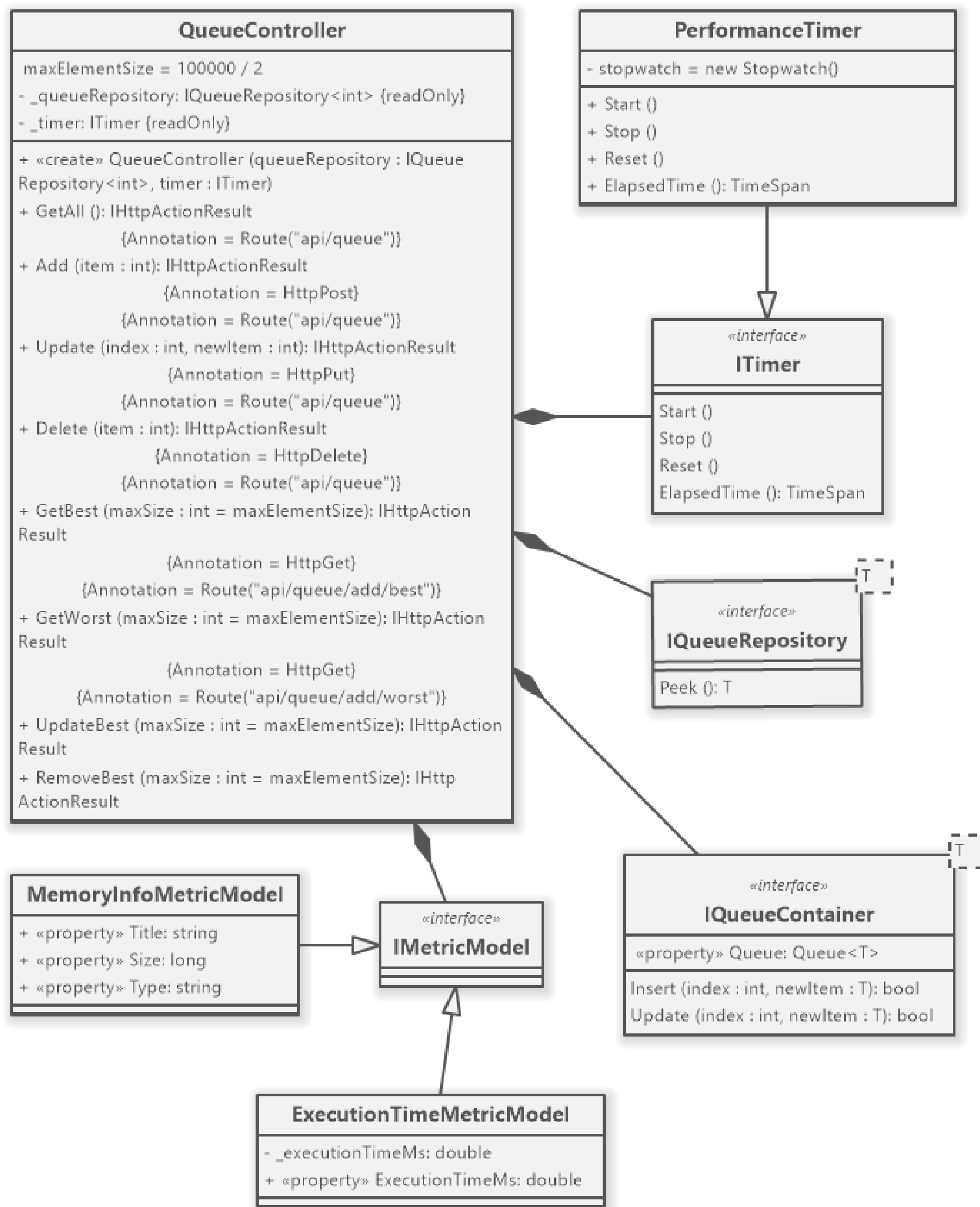

Рисунок 3.8 – Діаграма класів, взаємодія класів сервісів, моделей та даних з контролером колекції «черга»

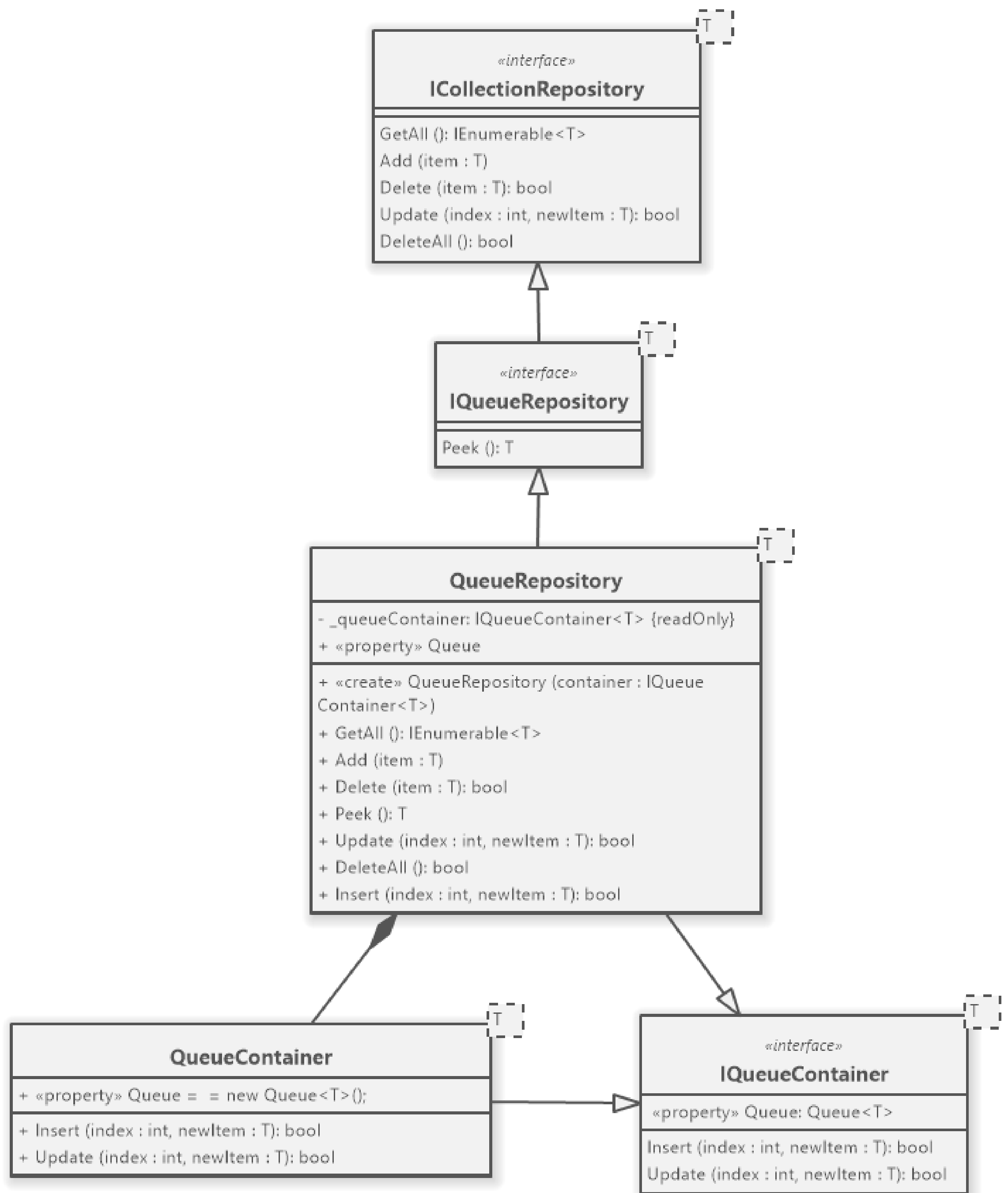

Рисунок 3.9 – Діаграма класів, яка відображає структуру класів рівня даних для колекції «черга»

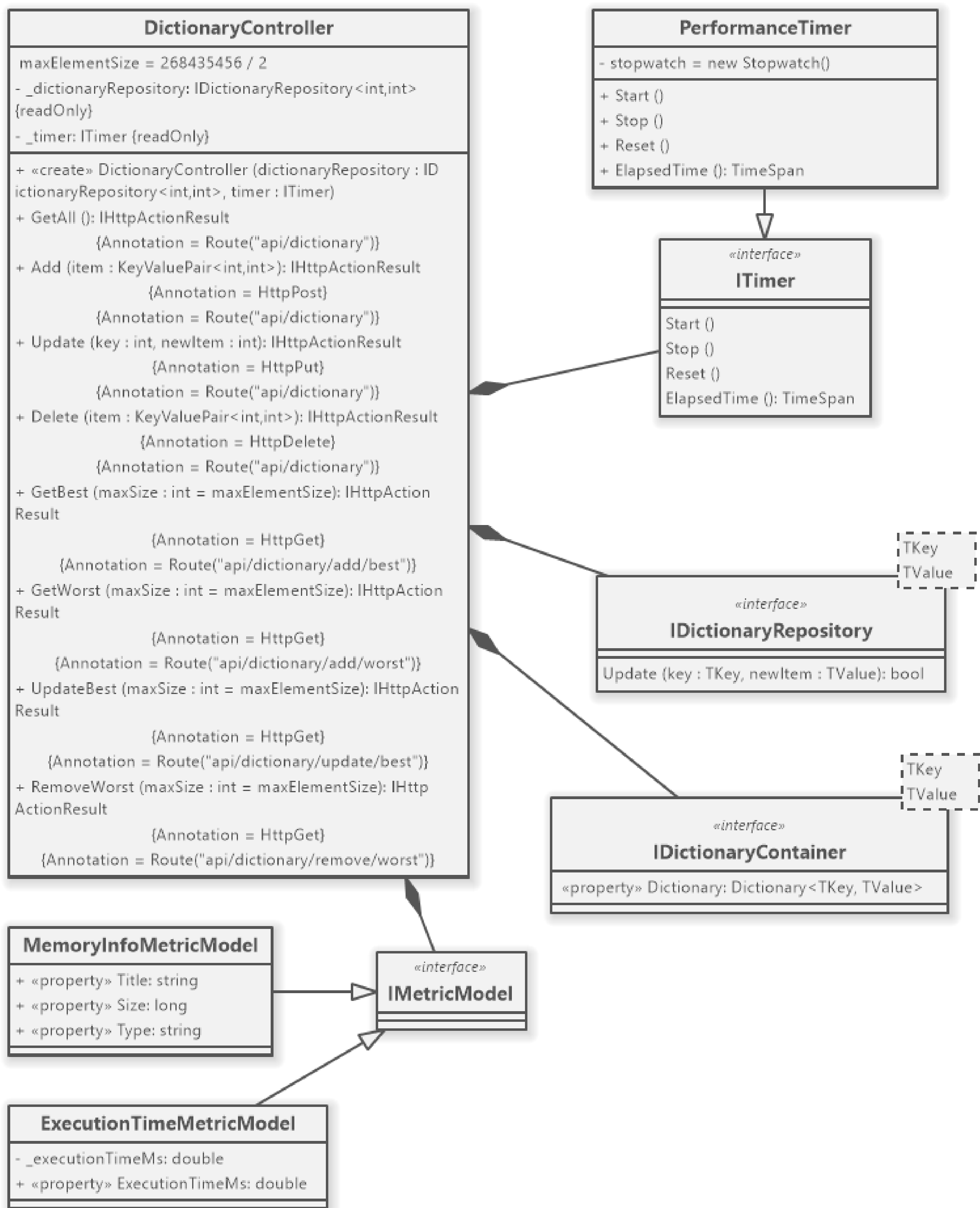

Рисунок 3.10 – Діаграма класів, взаємодія класів сервісів, моделей та даних з контролером колекції «словник»

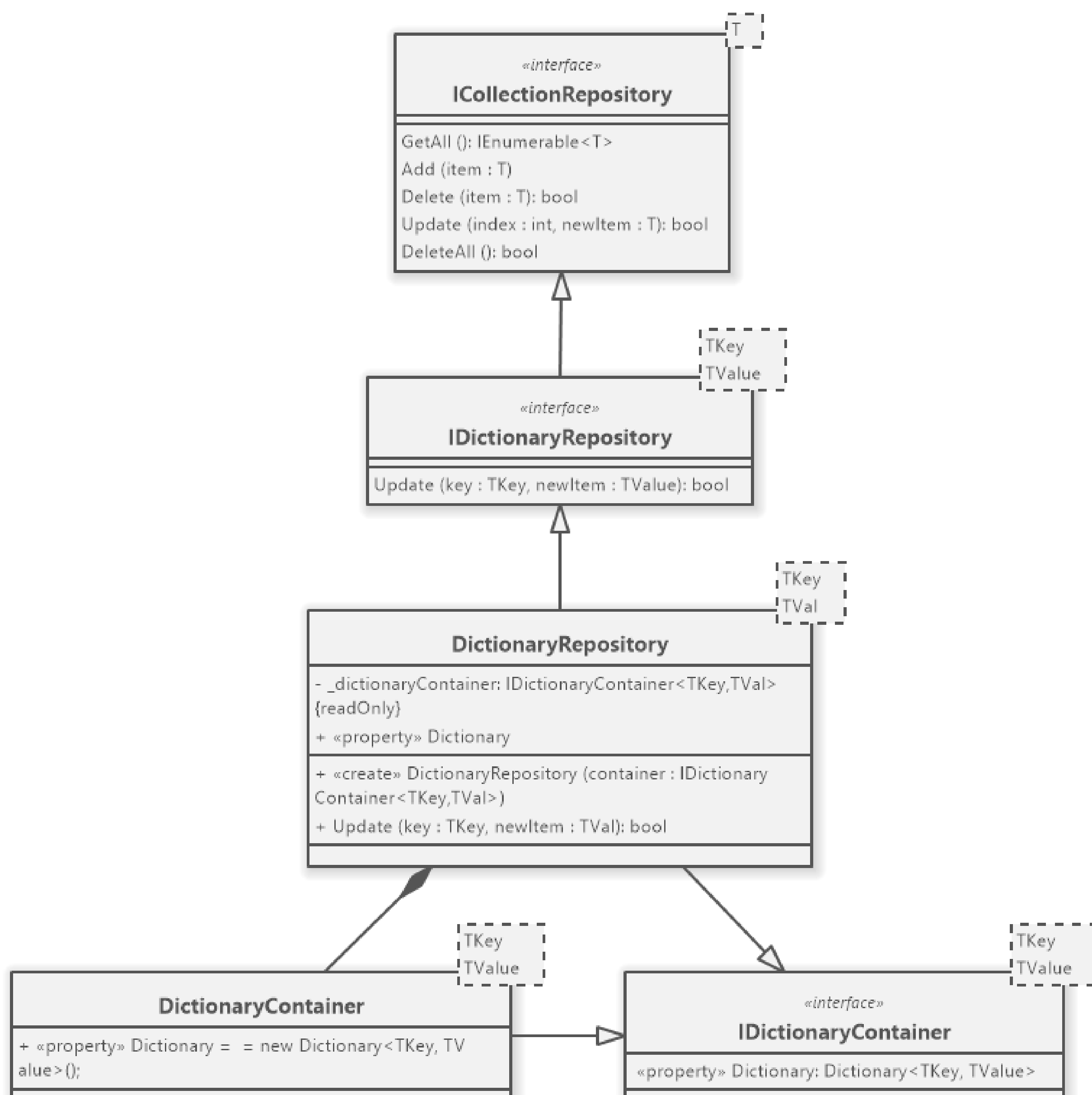

Рисунок 3.11 – Діаграма класів, яка відображає структуру класів рівня даних для колекції «словник»

3.3.4 Використані принципи проектування

Single Responsibility Principle (SOLID): кожен клас та компонент програми має одну конкретну відповідальність. Наприклад, ListController відповідає за операції зі списками, а PerformanceTimer за вимірювання часу виконання операцій. Це дозволяє зберегти структуру програми чистою та добре організованою.

Open-Closed Principle (SOLID): програма легко розширюється для дослідження інших типів колекцій та методів вимірювання продуктивності без змінення існуючого коду. Нові класи можуть бути додані, а інтерфейси реалізовані без впливу на існуючу функціональність.

KISS (Keep It Simple, Stupid): програма має просту та зрозумілу структуру. Кожен компонент має чітко визначену відповідність та завдання. Зберігання коду мінімалістичним та легко зрозумілим сприяє його підтримці та розширенню.

DRY (Don't Repeat Yourself): програма слідує принципу DRY, оскільки логіка операцій з колекціями винесена до репозиторіїв, які можуть бути використані для різних типів колекцій без дублювання коду. Також, інструменти вимірювання продуктивності використовуються у багатьох частинах програми без повторення їх логіки.

Інші принципи SOLID: інтерфейси та абстракції використовуються для розділення конкретної реалізації від клієнтського коду. Наприклад, IListRepository визначає контракти для операцій над списками, і різні реалізації репозиторіїв можуть відповідати цим контрактам.

3.3.5 Використані шаблони проектування

Модель-Представлення-Контролер (MVC):

- опис: MVC розділяє архітектуру додатка на три основні компоненти - модель, представлення і контролер. Модель відповідає за дані та бізнес-логіку, представлення за відображення інтерфейсу, а контролер за обробку вхідних подій та взаємодію з моделлю та видом[3];
- застосування: контролер ListController взаємодіє з моделлю (ListRepository) та представленням (HTTP дії), визначаючи логіку та обробляючи HTTP-запити.

Репозиторій (Repository):

- опис: патерн репозиторію визначає інтерфейс для доступу до колекції об'єктів, а його реалізація виконує конкретну роботу з даними[3];

– застосування: інтерфейси IListRepository та його реалізація ListRepository використовують підхід репозиторію для роботи з колекцією.

Фабричний метод (Factory Method):

– опис: патерн фабричного методу надає інтерфейс для створення об'єктів у суперкласі, але залишає підкласам можливість змінювати тип створюваних об'єктів[3];
– застосування: контролер ListController вказує на використання фабричного методу та ін'єкції залежностей для створення об'єктів.

Ітератор (Iterator):

– опис: ітератор дозволяє послідовно перебирати елементи колекції, не розкриваючи її внутрішньої структури[3];
– застосування: ітерація по елементах колекцій.

Одиночка (Singleton):

– опис: одиночка гарантує, що клас має лише один екземпляр та надає глобальну точку доступу до цього екземпляра[3];
– застосування: єдиний екземпляр ListContainer створюється у класі ListRepository.

Стратегія (Strategy):

– опис: стратегія визначає сімейство алгоритмів, і робить їх взаємозамінюваними. Вона дозволяє вибирати алгоритм в час виконання[3];
– застосування: реалізація інтерфейсу ITimer для вимірювання часу може слугувати стратегією вимірювання часу.

Шаблонний метод (Template Method):

– опис: шаблонний метод визначає загальну структуру алгоритму, а кроки для реалізації залишає підкласам;
– застосування: методи в контролері (GetBest, UpdateBest, RemoveWorst) мають загальну структуру, залишаючи кроки для реалізації в підкласах.

Фасад (Facade):

- опис: фасад надає простий інтерфейс для складної системи об'єктів, полегшуючи використання цієї системи[3];
- застосування: клас MemoryInfoProvider може слугувати фасадом, який надає простий інтерфейс для отримання різних метрик пам'яті.

Компоновщик (Composite):

- опис: компоновщик дозволяє об'єднувати об'єкти в деревоподібній структурі для представлення частино-цілого взаємодії[3];
- застосування: PerformanceTestModel містить колекцію різних метрик, які можна розглядати як «листки» компоновщика.

Команда (Command):

- опис: команда перетворює запити або операції в об'єкти, що дозволяє параметризувати клієнта з запитами, обчислювати, чергувати та підтримувати скасування операцій[3];
- застосування: методи в контролері (GetAll, Add, Update, Delete, GetBest, UpdateBest, RemoveWorst) можна розглядати як команди, які виконують певні дії.

3.4 Тестування та налагодження програми

3.4.1 Метод тестування граничними умовами

Тестування граничних умов є важливою частиною тестової стратегії для перевірки коректності та стабільності системи у крайніх сценаріях використання. Граничні умови можуть включати обробку великих обсягів даних, найвищі та найнижчі значення параметрів, а також невалідні вхідні дані. Такий підхід дозволяє виявити можливі проблеми та забезпечити стабільність системи в різних умовах.

Таблиця 3.1 – Створені тести

Тестовий сценарій	Вхідні дані	Очікуваний результат	Результат
Додавання елементу з максимальною величиною	item = 536870912	BadRequest	BadRequest
Виклик GetAll при порожньому списку	Немає вхідних даних	Повернення порожнього списку	Повернення порожнього списку
Оновлення елемента з негативним індексом	index = -1, newItem = 42	NotFound	NotFound
Видалення неіснуючого елемента	item = 100	NotFound	NotFound

3.4.2 Налагодження програми

В процесі розробки програмного забезпечення, однією з важливих складових є налагодження, яка дозволяє виявляти та виправляти помилки в коді. Одним з ефективних методів налагодження є «Послідовне налагодження» або «Step-by-Step Debugging». Цей підхід надає можливість розглядати виконання програми крок за кроком, аналізувати значення змінних та виявляти місця, де виникають неполадки.

При використанні методу «Послідовне налагодження», важливо обирати точки в коді, де ймовірність наявності помилок найвища. Це можуть бути складні фрагменти коду, методи, які викликаються в різних сценаріях, чи ті місця, де ви маєте підозри щодо неправильної поведінки системи.

Переваги «Послідовного налагодження»:

– детальний аналіз: дозволяє докладно розглядати виконання коду, крок за кроком, для з'ясування причин помилок;
– зручність: надає можливість контролювати виконання програми на кожному етапі, щоб виявити аномалії;
– контекст: допомагає зосередитися на конкретних ділянках коду, де ймовірність помилок велика.

Використання «Послідовного налагодження» в коді:

– встановлення точок зупину в обраних модулях чи методах коду;
– запуск програми у режимі налагодження;

– перехід від однієї точки зупину до іншої за допомогою команд «Next Step» або «Continue»;
– контроль значень змінних та об'єктів на кожному кроці;
– виявлення та виправлення помилок у вихідному коді.

Обрання «Послідовного налагодження» в процесі розробки дозволяє забезпечити якість коду та ефективно вирішувати потенційні проблеми, забезпечуючи стабільність та вірогідність успішного виконання програмного продукту.

Висновки до розділу 3

Цей розділ описує важливі кроки розробки та планування проекту.

Формалізація задачі була проведена за допомогою діаграм прецедентів, що чітко визначило функціональні можливості системи та взаємодію користувачів з нею.

Базова архітектура системи, побудована на принципах Model-View-Controller (MVC), визначає структуру програми, дозволяючи відокремити логіку, представлення та управління даними. Це надає гнучкість та можливість подальшого розширення системи.

Внутрішнє проектування включає вибір мови програмування (C#), технологічної платформи (Visual Studio), та ієрархії класів системи. Принципи SOLID та інші принципи проектування, такі як «Don't repeat yourself» та «Keep It Simple, Stupid», допомагають створити чистий та ефективний код.

Тестування системи визначено як важливий етап, і використання методу тестування граничними умовами дозволяє виявити та виправити можливі проблеми. Налагодження програми виконується за допомогою послідовного налагодження, що полегшує виявлення помилок у коді.

Загальний підхід до проектування та розробки системи дотримується сучасних практик у сфері розробки програмного забезпечення. Це створює підґрунтя для успішного впровадження проекту для дослідження продуктивності платформ ASP.NET та ASP.NET Core.

4 ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ПЛАТФОРМ

4.1 Підготовка до експерименту

В реалістичних умовах важливо визначити, як система веде себе при роботі з різними обсягами даних та різними типами операцій.

При підготовці до експерименту варто визначити сценарії експерименту для подальшого його виконання.

Таблиця 4.1 – Тестові сценарії структур даних

№	Структура даних	Дія	Кількість елементів	Кількість запитів
1	Словник	Add Best	100	100
2	Словник	Add Best	10,000	100
3	Словник	Add Worst	100	100
4	Словник	Add Worst	10,000	100
5	Словник	Update Best	100	100
6	Словник	Update Best	10,000	100
7	Словник	Remove Worst	100	100
8	Словник	Remove Worst	10,000	100
9	Список	Add Best	100	100
10	Список	Add Best	1,000	100
11	Список	Update Best	100	100
12	Список	Update Best	1,000	100
13	Список	Remove Worst	100	100
14	Список	Remove Worst	1,000	100
15	Черга	Add Best	100	100
16	Черга	Add Best	10,000	100
17	Черга	Add Worst	100	100
18	Черга	Add Worst	1,000	100
19	Черга	Update Best	100	100
20	Черга	Update Best	1,000	100
21	Черга	Remove Best	100	100
22	Черга	Remove Best	1,000	100

Ці тестові сценарії допоможуть дослідити реакцію системи на різні сценарії введення даних та визначити продуктивність для кожної операції та структури даних.

Метою цього експерименту є дослідження швидкості обробки найпростішого запиту на отримання даних для обох платформ - ASP.NET Core та ASP.NET. Для цього розглядаються два варіанти виконання: асинхронний та синхронний. Один і той же запит виконується багато разів підряд (100, 1000, 10000 та 100000 разів), що дозволяє створити імітацію навантаження системи та проаналізувати стан системи при різних рівнях навантаження.

Синхронний варіант виконання: у синхронному варіанті запити обробляються послідовно, один за одним. Кожен наступний запит чекає завершення попереднього перед виконанням. Це означає, що весь процес виконання зупиняється, доки поточний запит не завершить своє виконання.

Асинхронний варіант виконання: у випадку асинхронного виконання запити обробляються паралельно. Коли запит встає на очікування результатів, інші запити можуть продовжити своє виконання. Це дозволяє системі більш ефективно використовувати ресурси та знижує час очікування для клієнтів.

Цей експеримент дозволить визначити, як система веде себе при великій кількості однотипних запитів у синхронному та асинхронному режимах.

4.2 Проведення експерименту з дослідження колекції «Словник»

Оскільки дослідження показують приблизно однакові показники відносно графіків порівняння, то буде відображено лише графіки з дослідженням для великої кількості елементів (рис. 4.1 – 4.16).

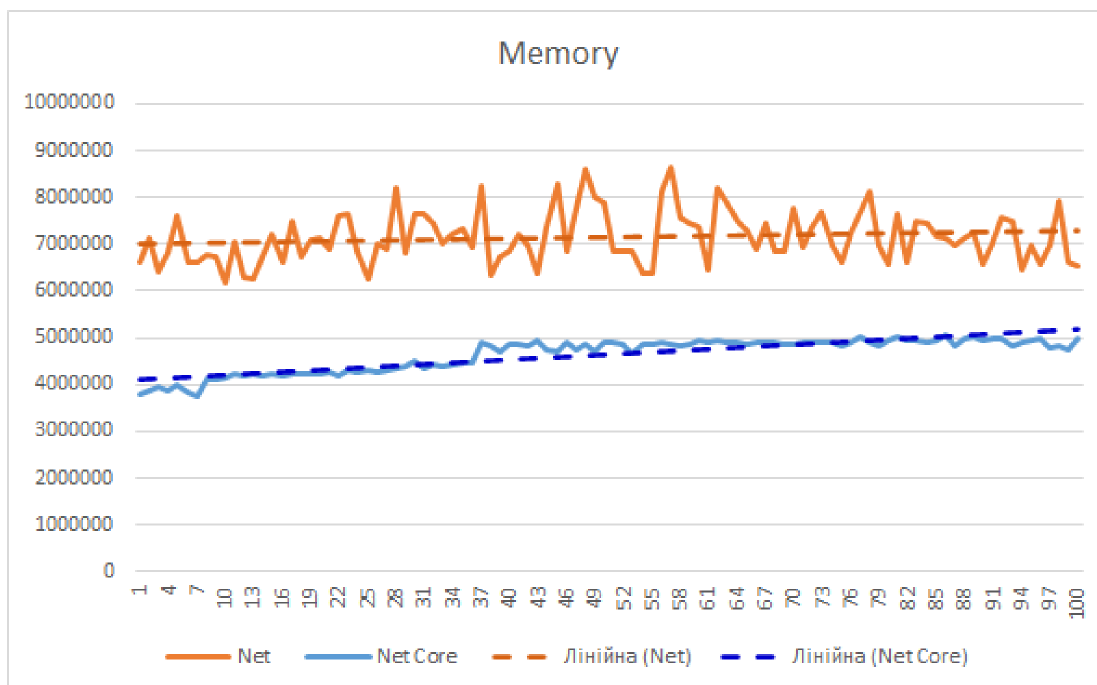

Рисунок 4.1 – графіки використання пам’яті збірником сміття add best

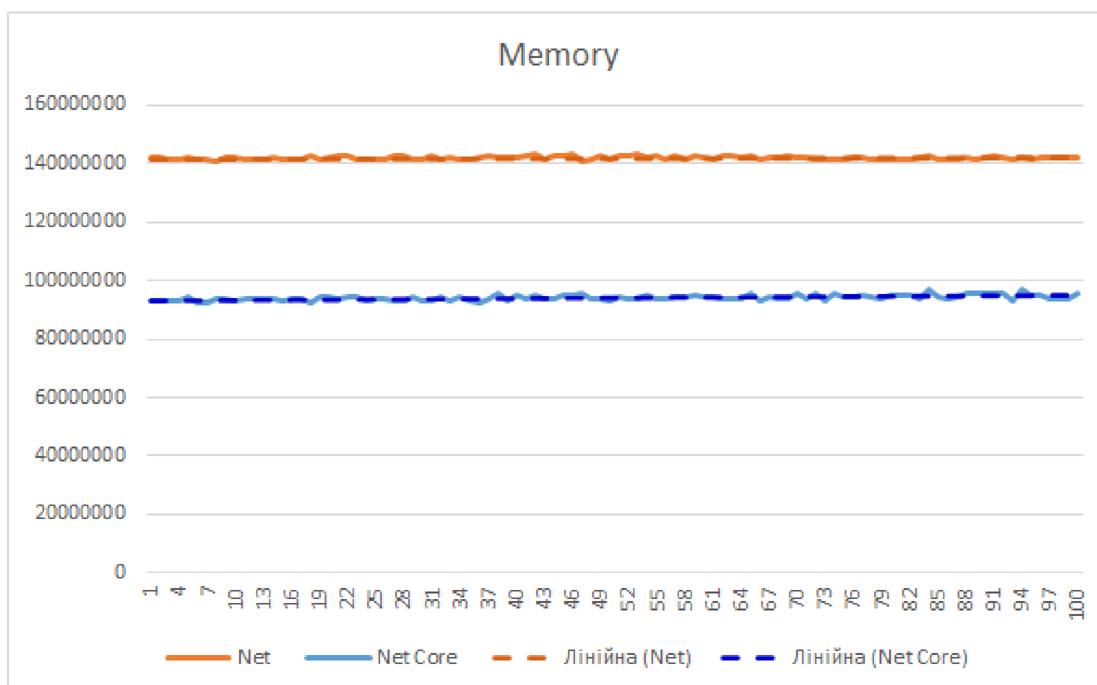

Рисунок 4.2 – графіки використання пам’яті процесу перед тестування add best

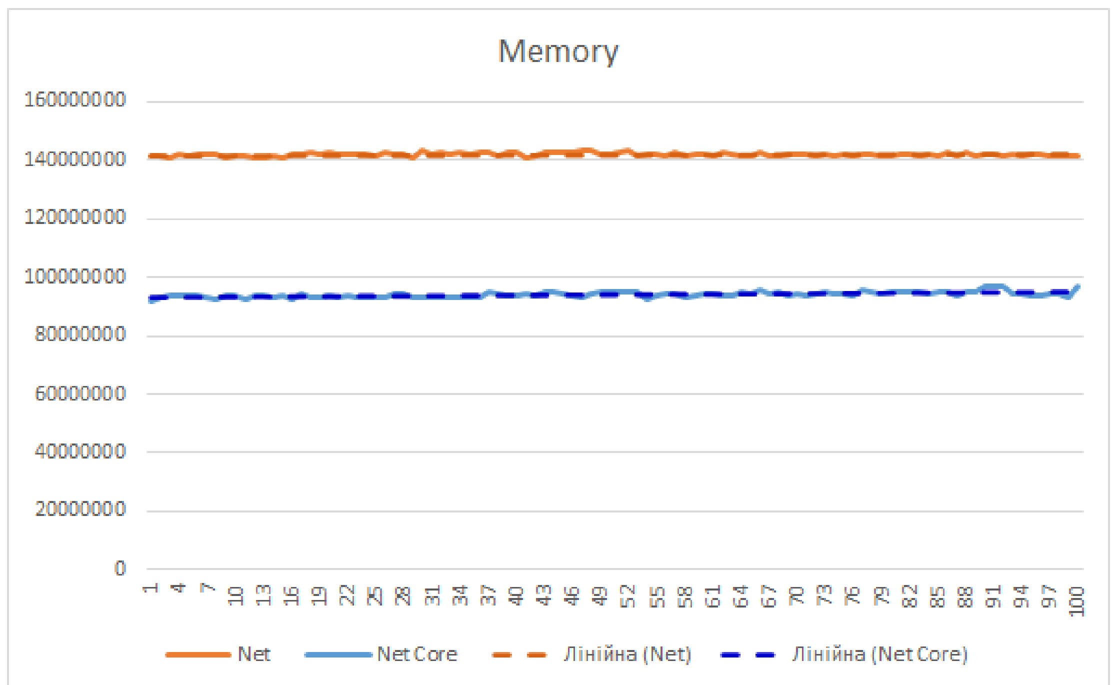

Рисунок 4.3 – графіки використання пам’яті процесу після тестування add best

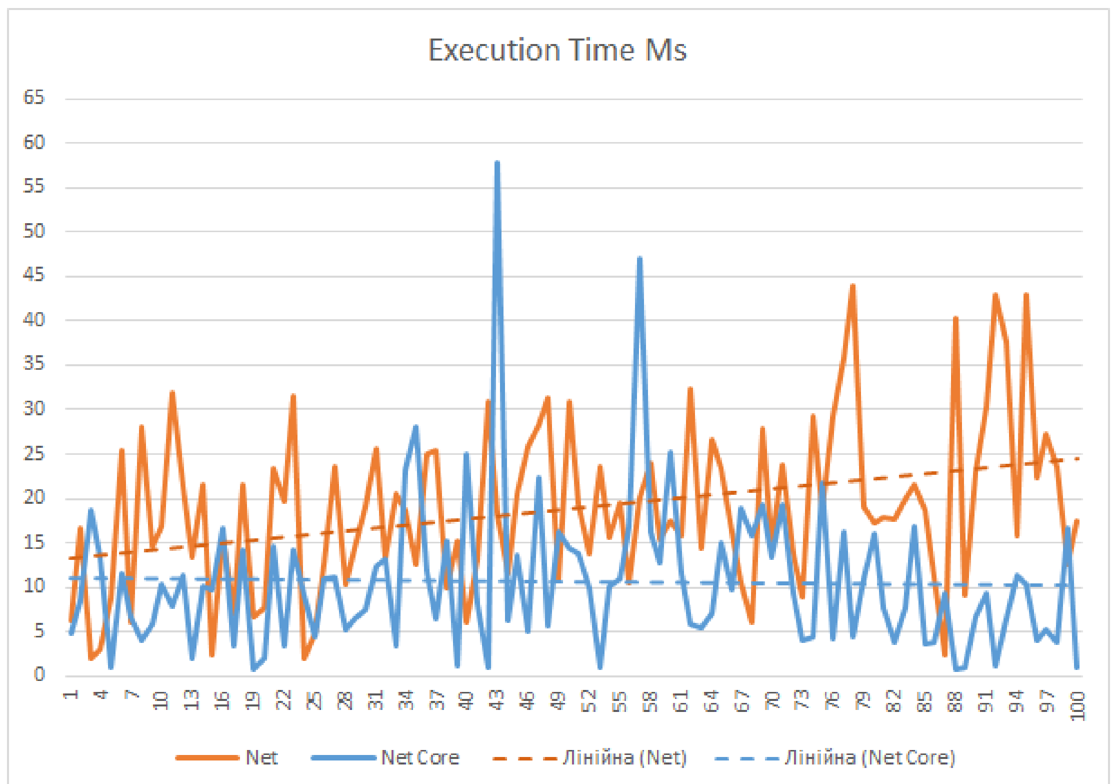

Рисунок 4.4 – графіки швидкості обробки запитів add best

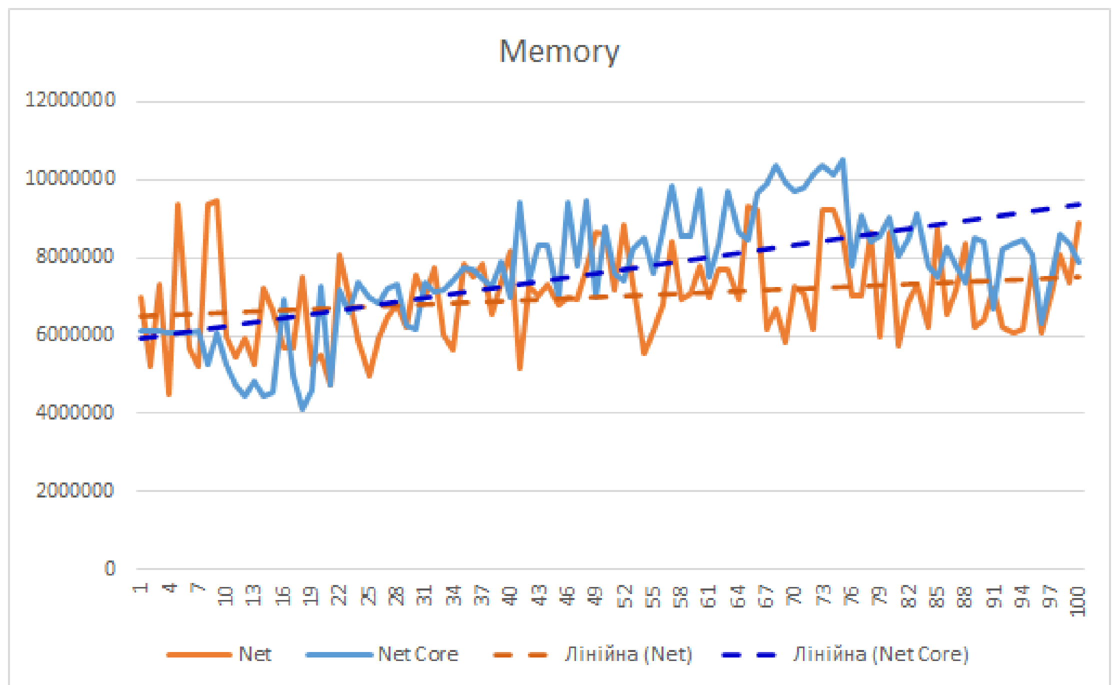

Рисунок 4.5 – графіки використання пам'яті збірником сміття add worst

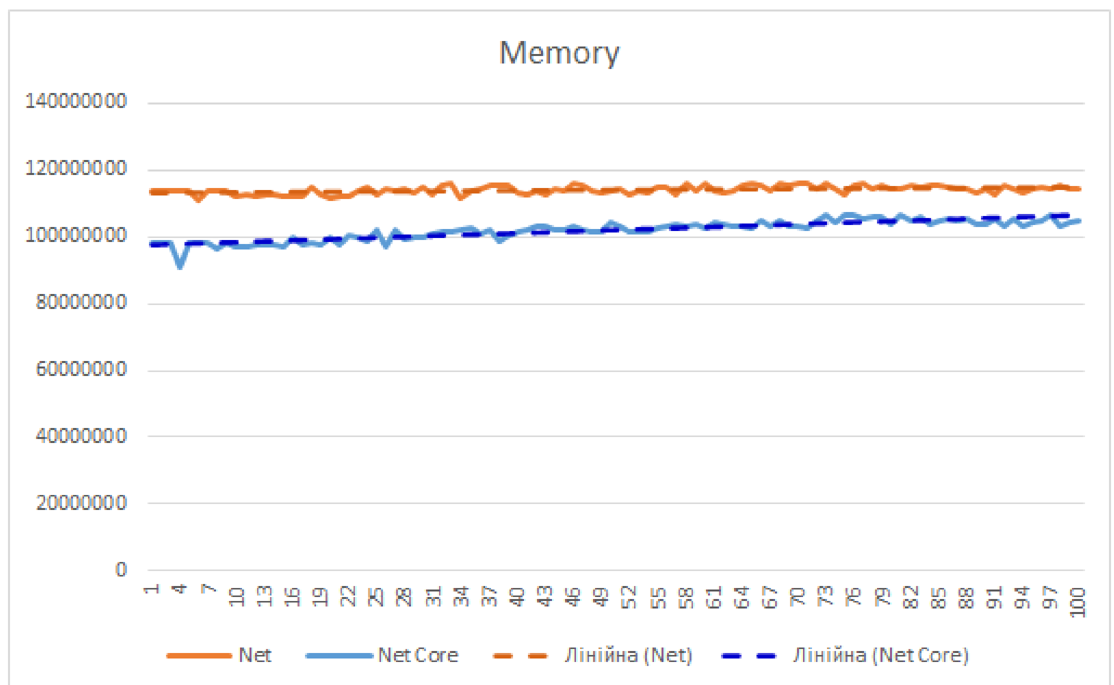

Рисунок 4.6 – графіки використання пам'яті процесу перед тестування add worst

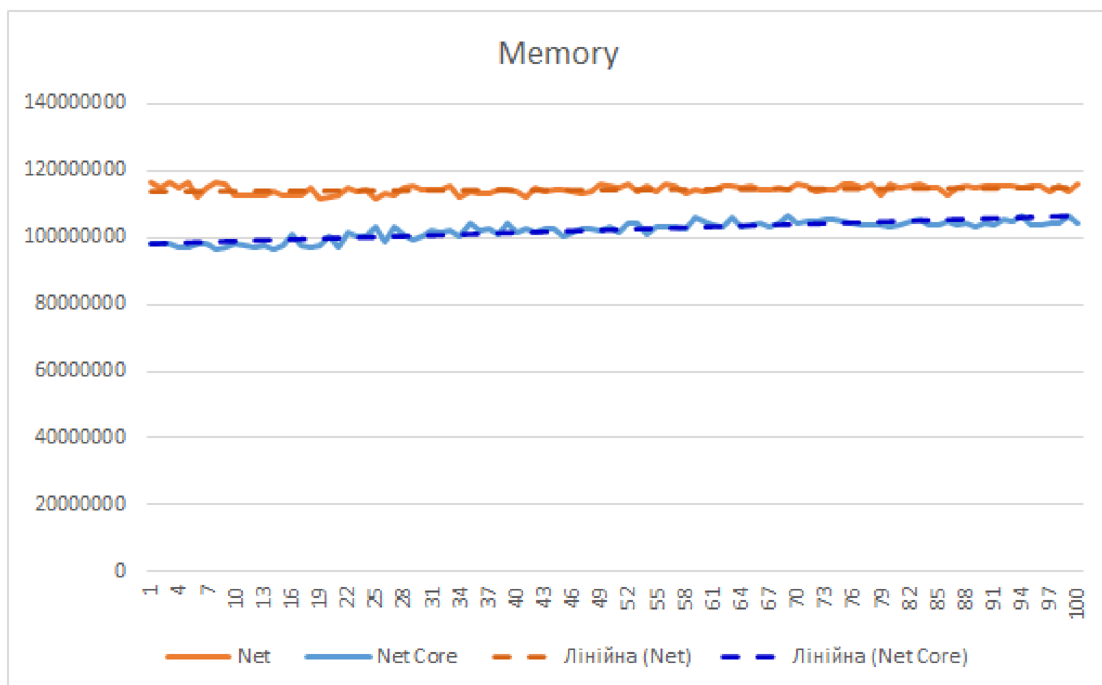

Рисунок 4.7 – графіки використання пам’яті процесу після тестування add worst

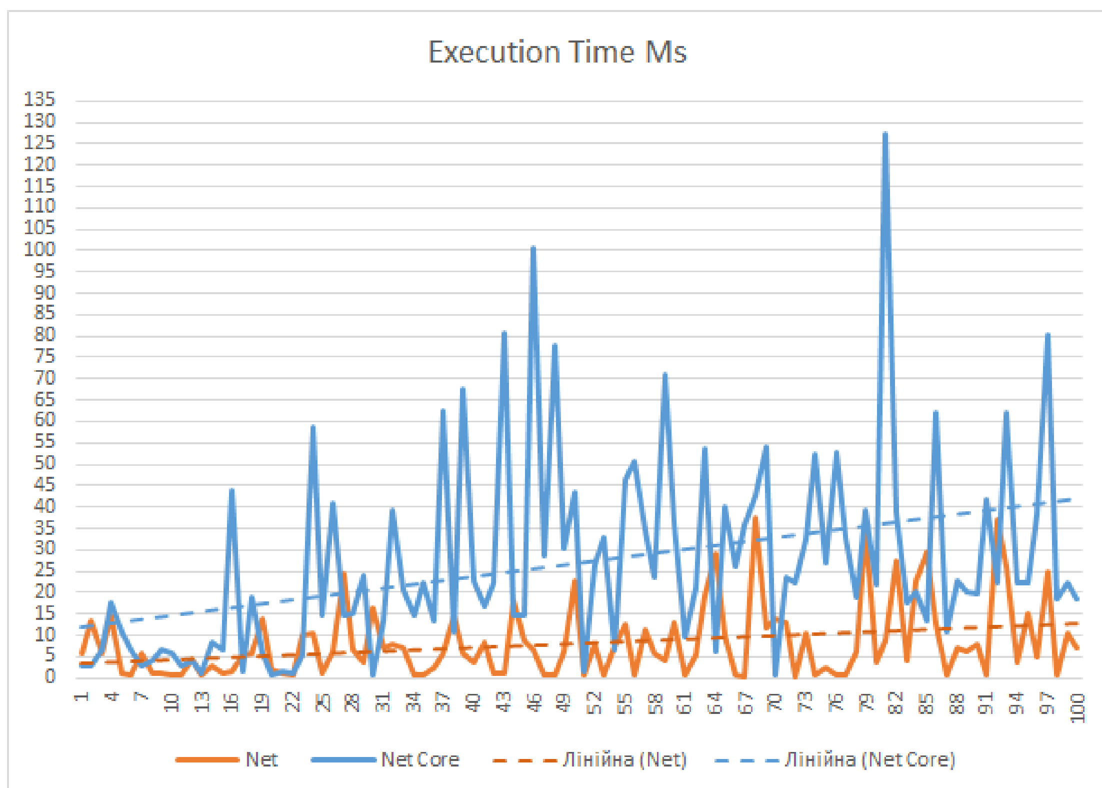

Рисунок 4.8 – графіки швидкості обробки запитів add worst

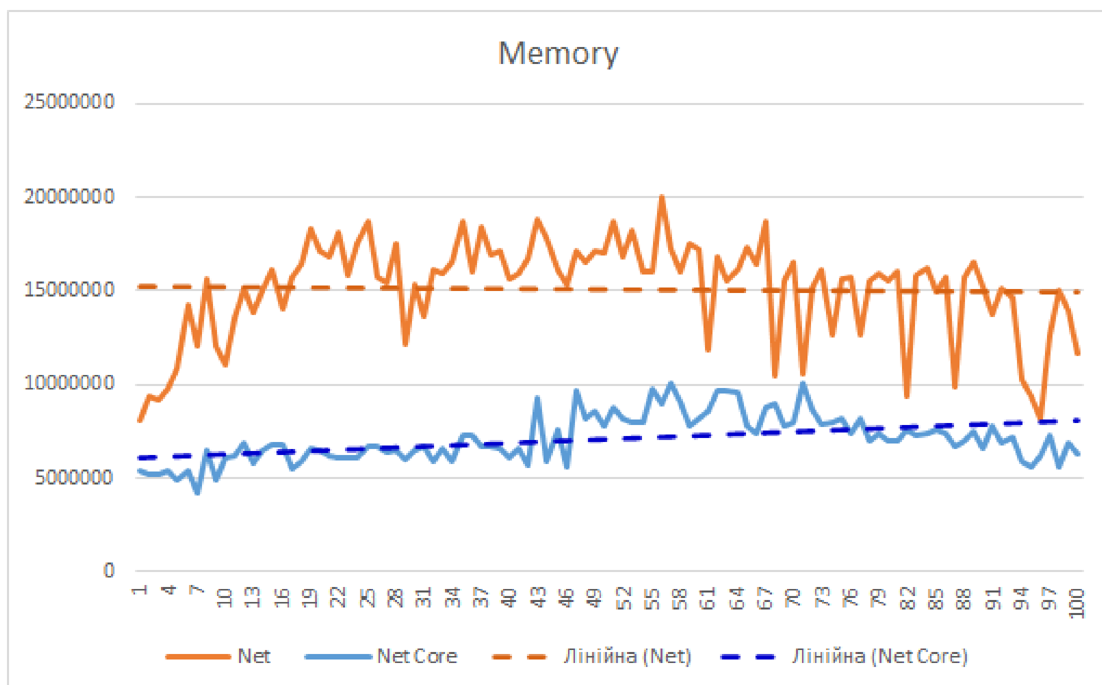

Рисунок 4.9 – графіки використання пам’яті збірником сміття remove worst

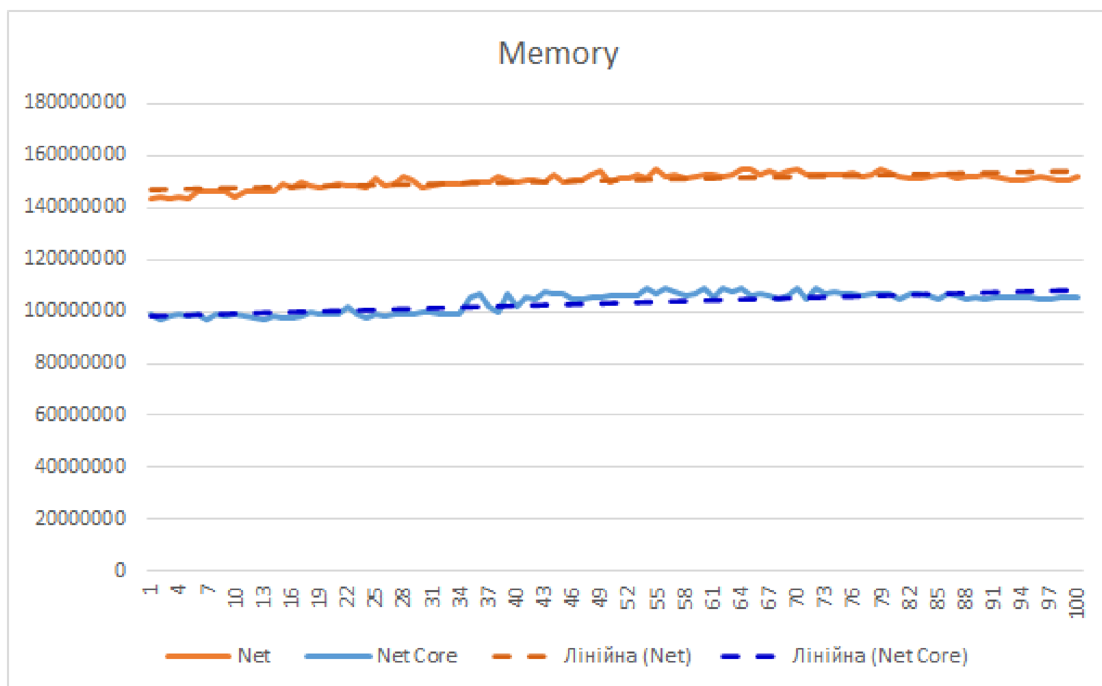

Рисунок 4.10 – графіки використання пам’яті процесу перед тестування remove worst

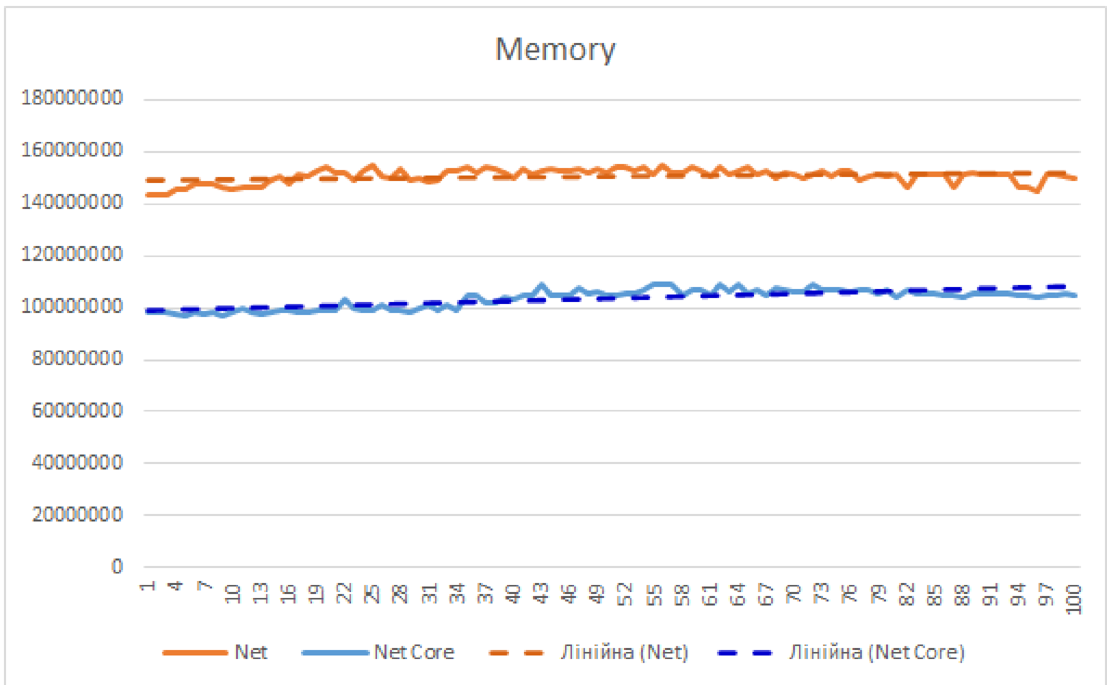

Рисунок 4.11 – графіки використання пам'яті процесу після тестування remove worst

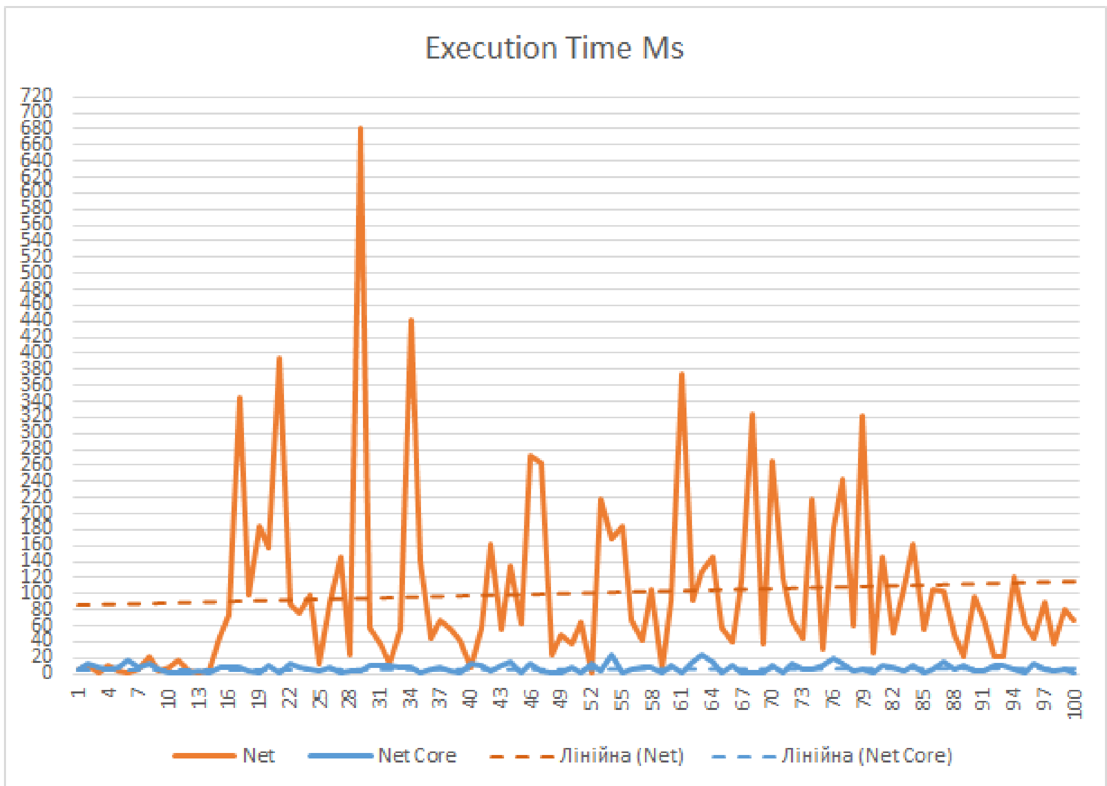

Рисунок 4.12 – графіки швидкості обробки запитів remove worst

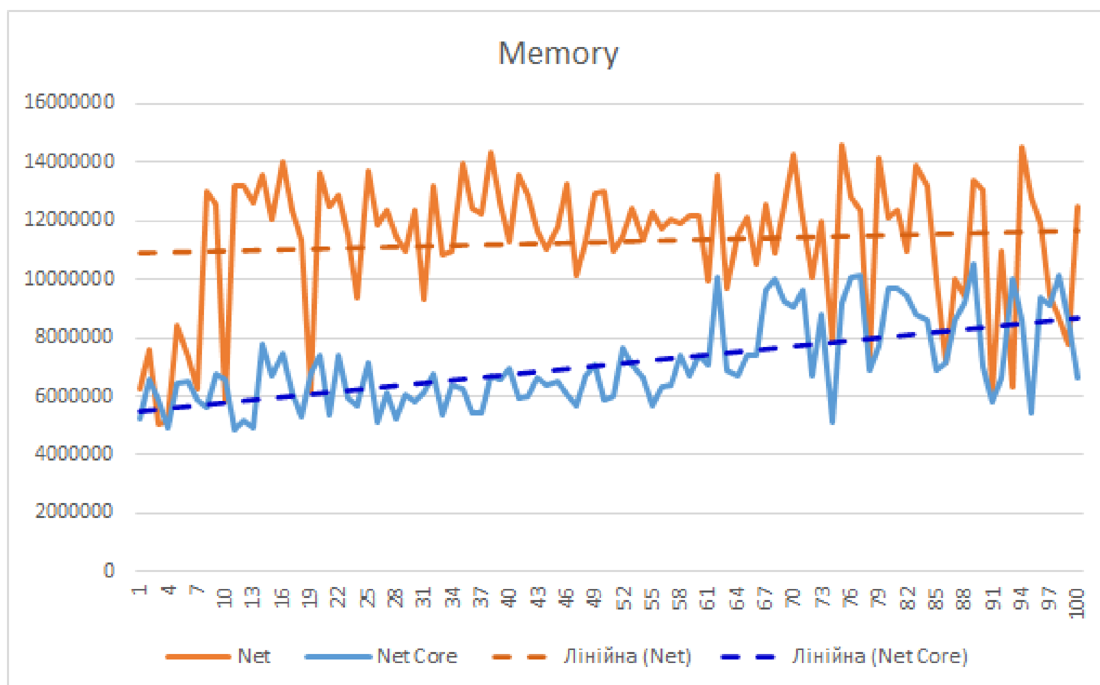

Рисунок 4.13 – графіки використання пам’яті збірником сміття update best

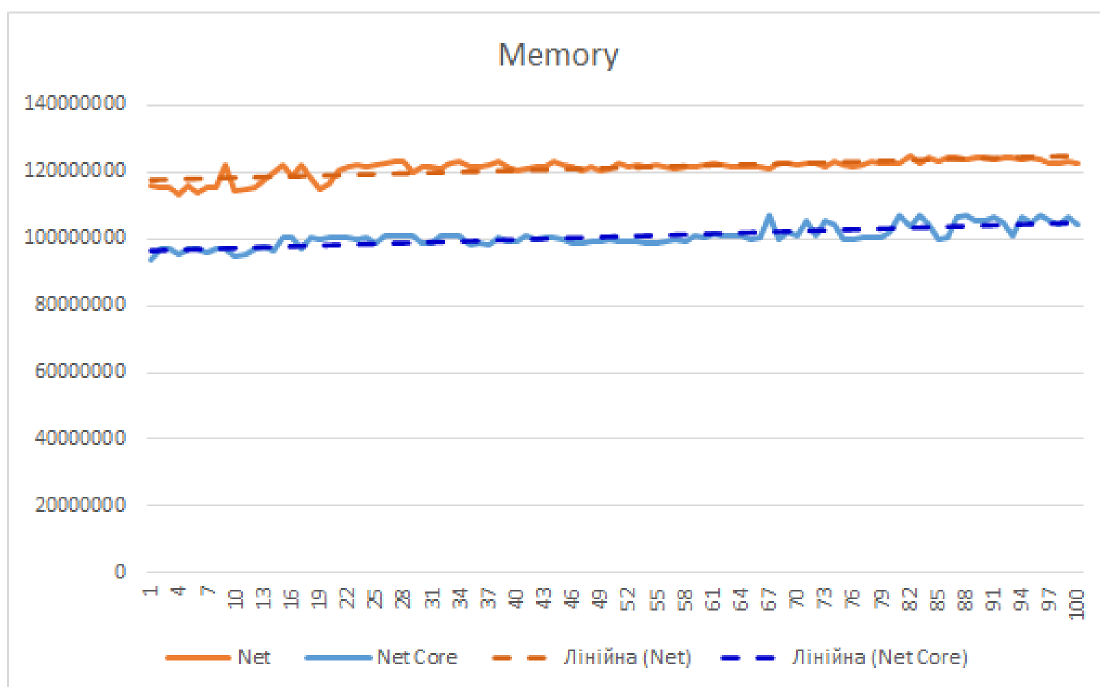

Рисунок 4.14 – графіки використання пам’яті процесу перед тестування update best

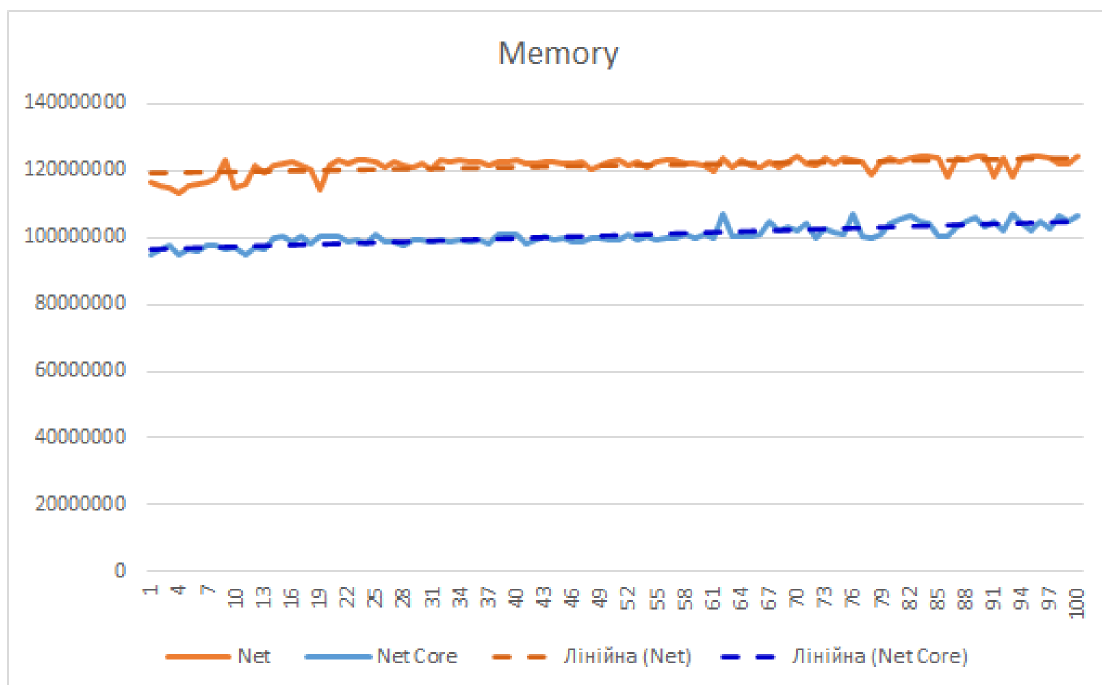

Рисунок 4.15 – графіки використання пам'яті процесу після тестування update best

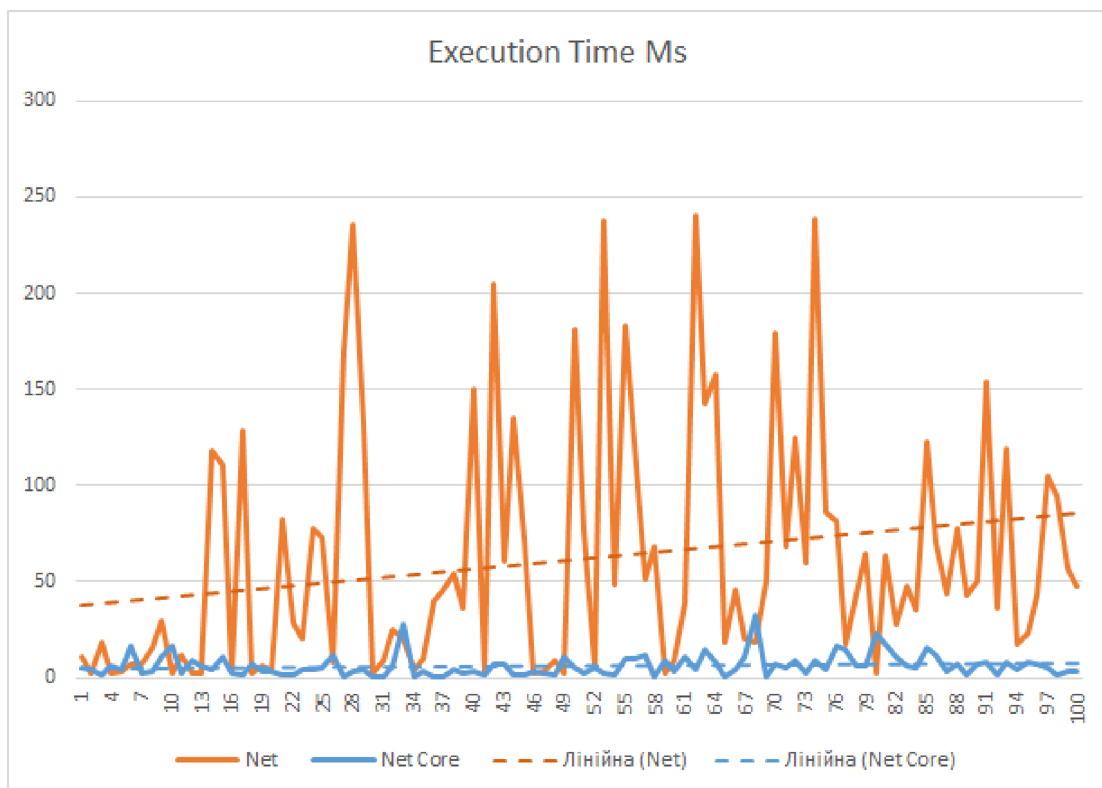

Рисунок 4.16 – графіки швидкості обробки запитів update best

4.3 Проведення експерименту з дослідження колекції «Список»

Це дослідження також показує приблизно однакові показники відносно графіків порівняння, тому буде відображено лише графіки з дослідженням для великої кількості елементів (рис. 4.17 – 4.28).

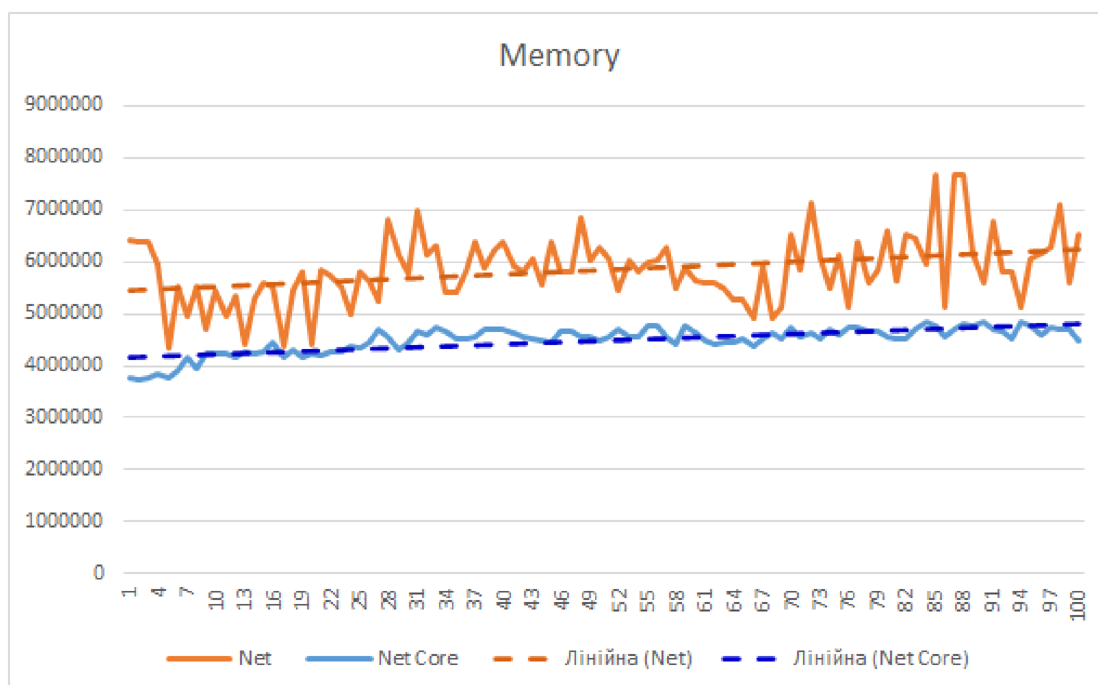

Рисунок 4.17 – графіки використання пам’яті збірником сміття add best

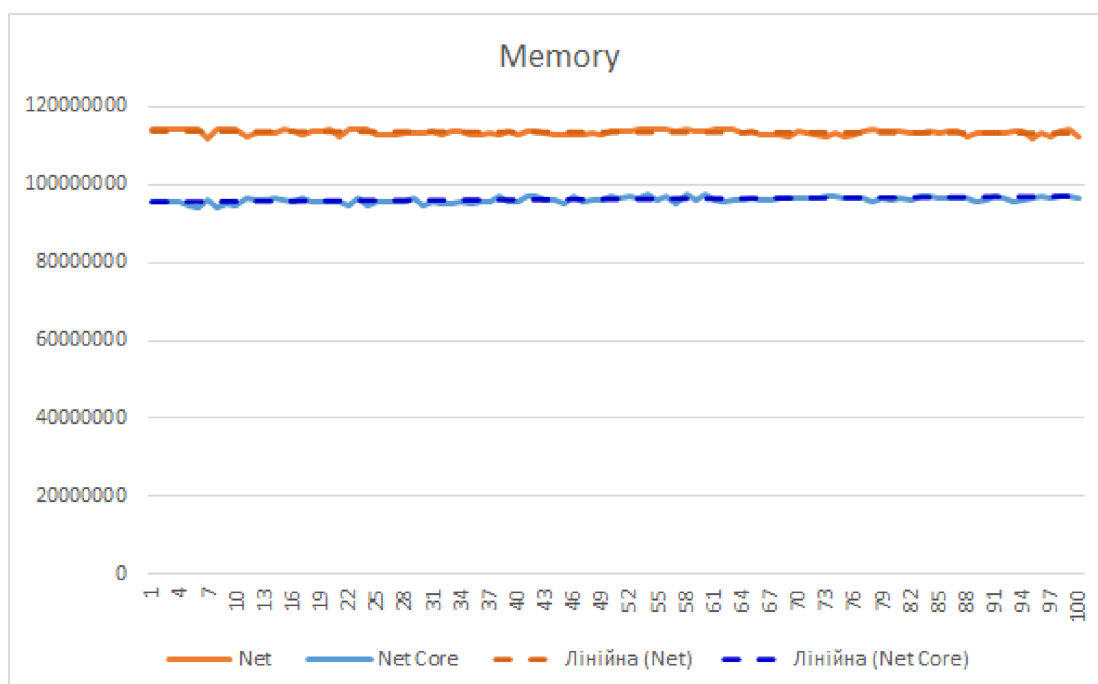

Рисунок 4.18 – графіки використання пам’яті процесу перед тестування add best

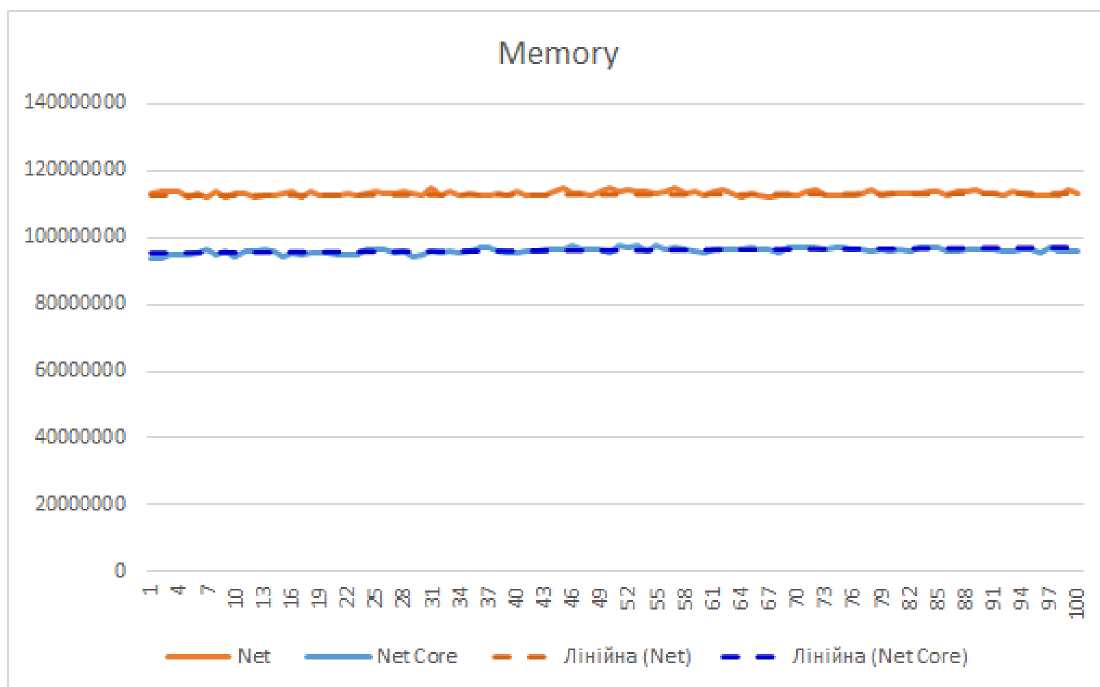

Рисунок 4.19 – графіки використання пам’яті процесу після тестування add best

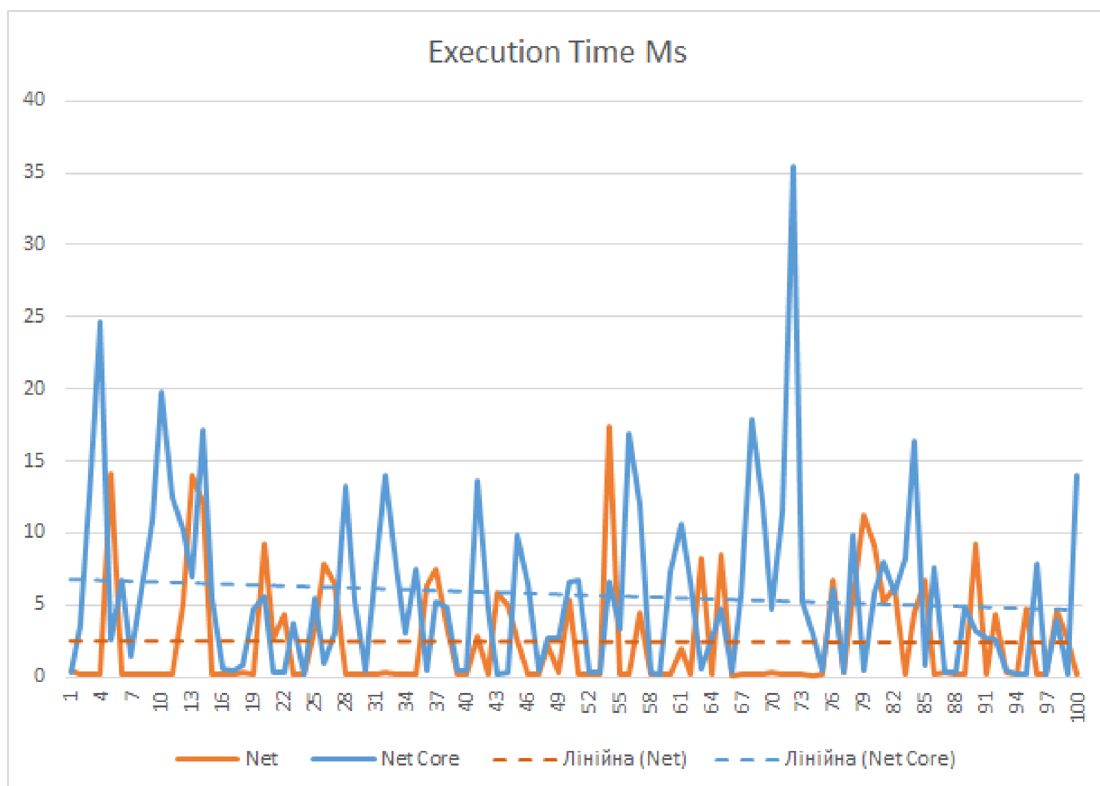

Рисунок 4.20 – графіки швидкості обробки запитів add best

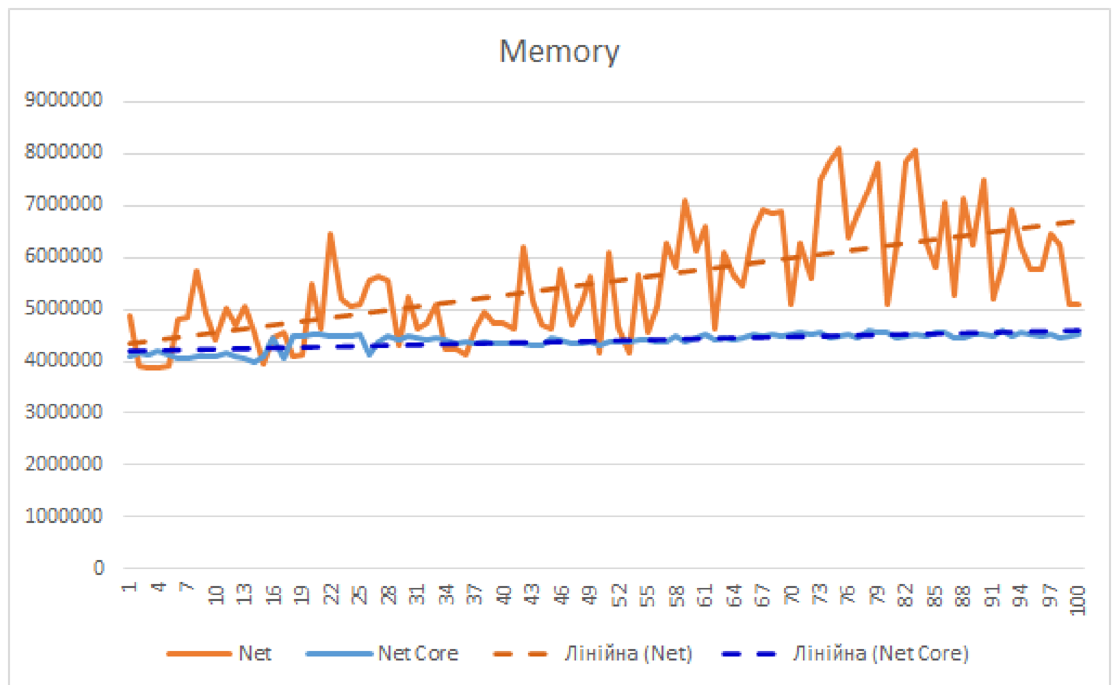

Рисунок 4.21 – графіки використання пам’яті збірником сміття remove worst

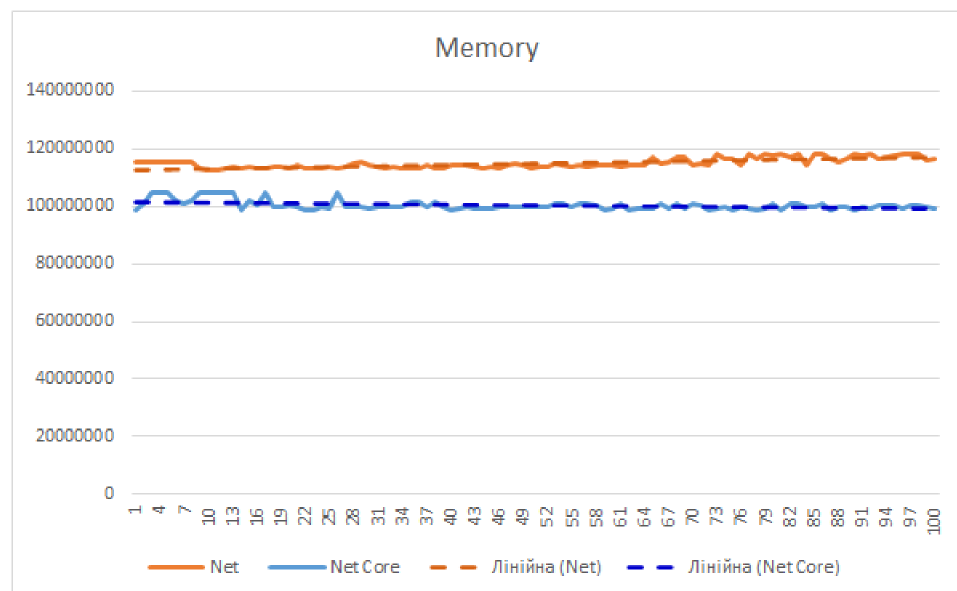

Рисунок 4.22 – графіки використання пам’яті процесу перед тестування remove worst

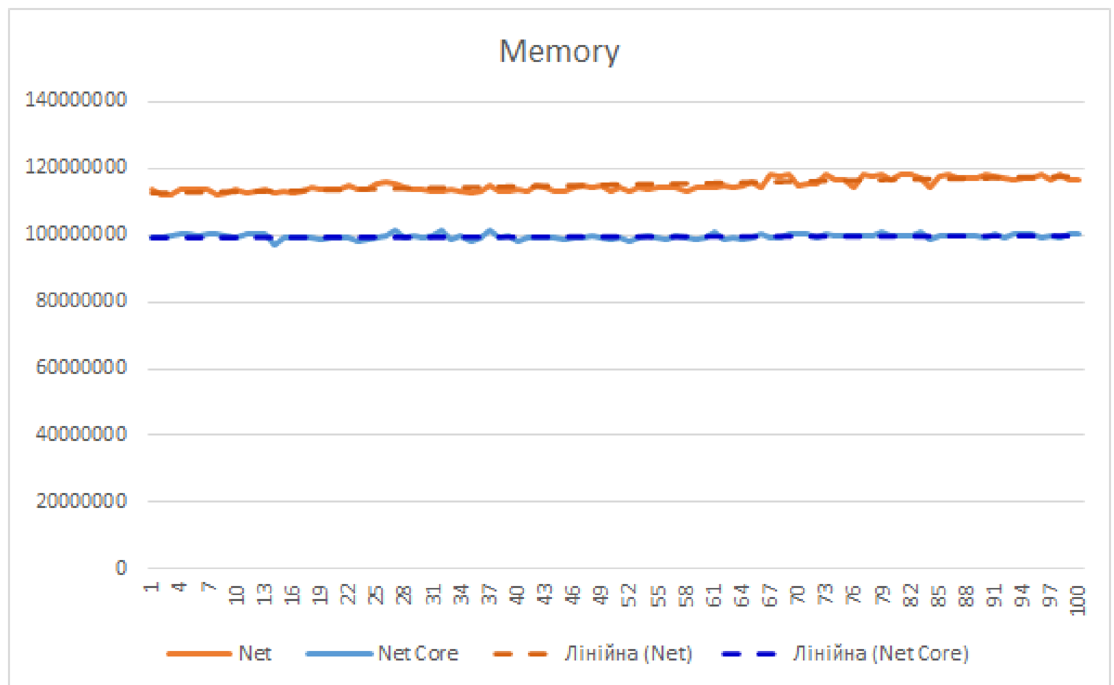

Рисунок 4.23 – графіки використання пам’яті процесу після тестування remove worst

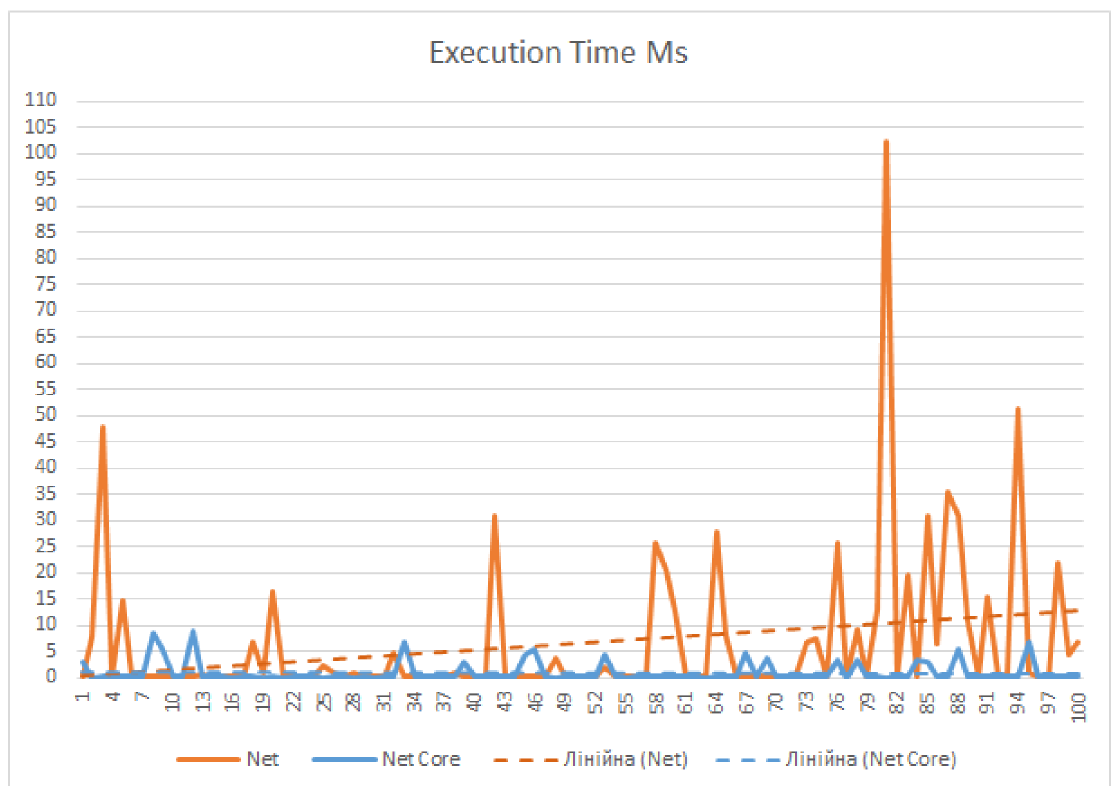

Рисунок 4.24 – графіки швидкості обробки запитів remove worst

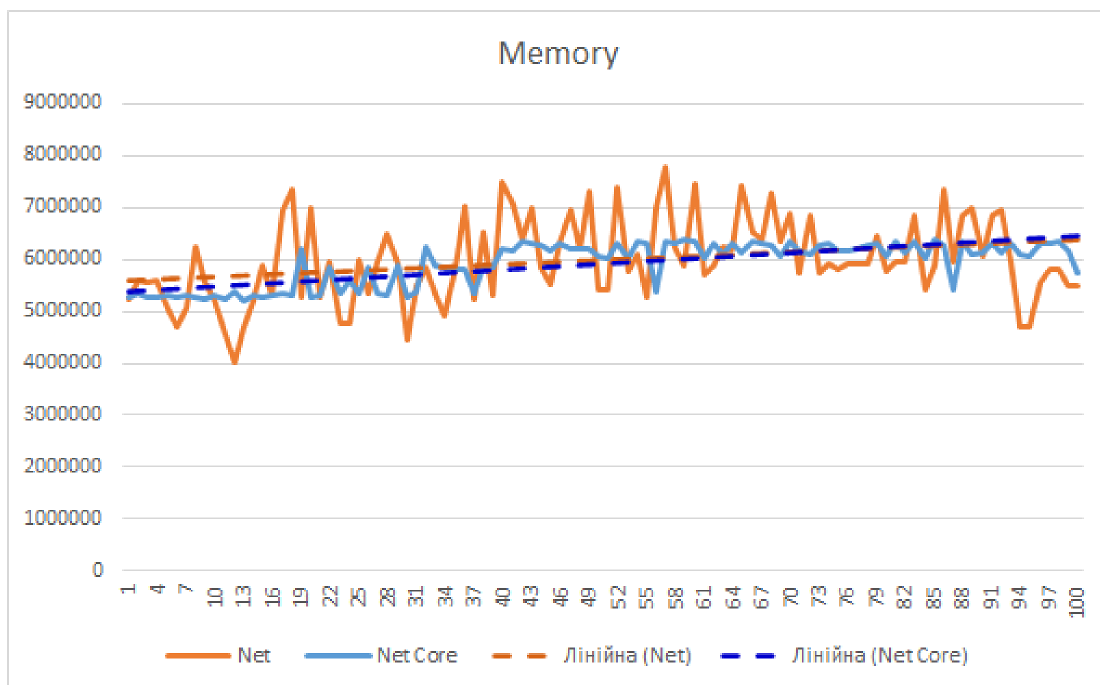

Рисунок 4.25 – графіки використання пам’яті збірником сміття update best

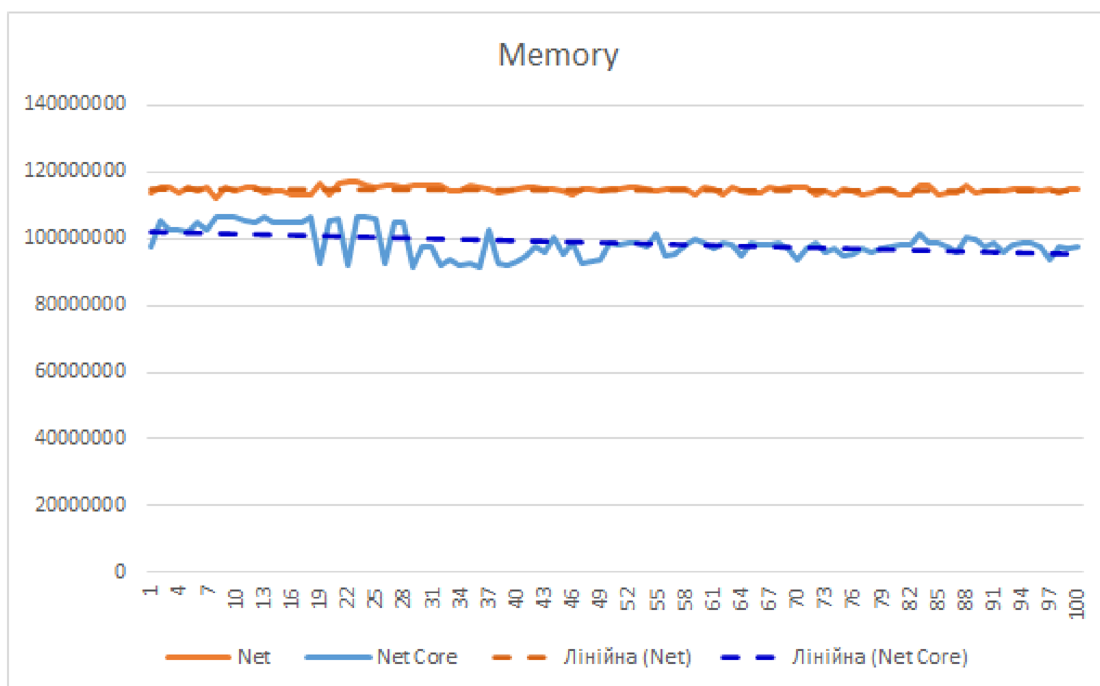

Рисунок 4.26 – графіки використання пам’яті процесу перед тестування update best

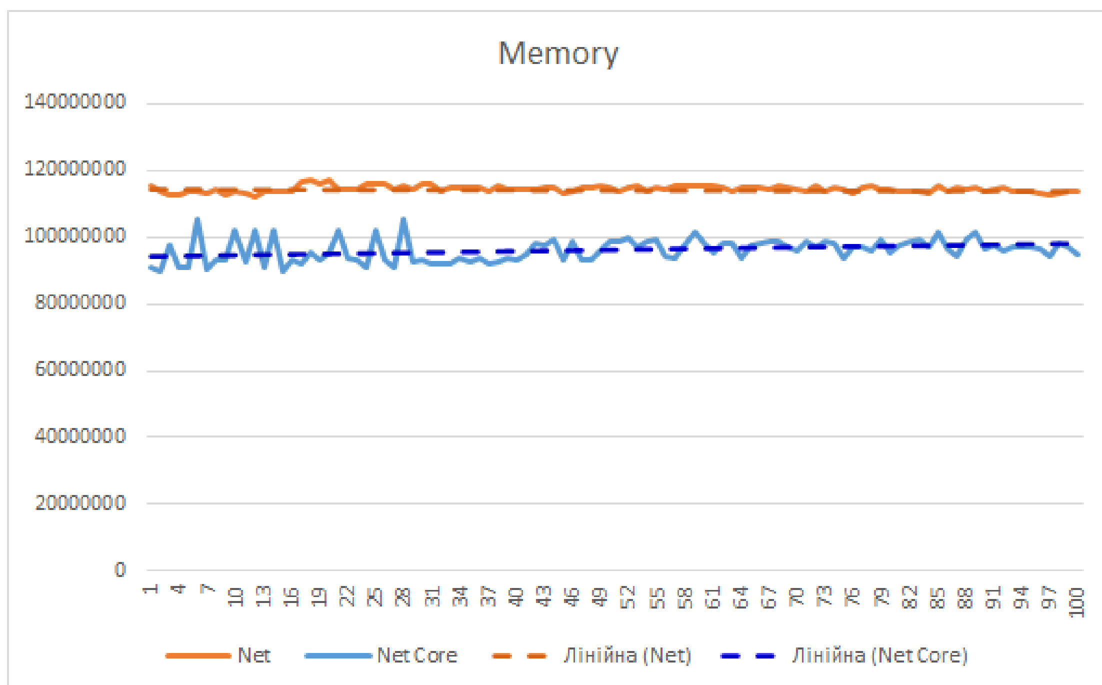

Рисунок 4.27 – графіки використання пам’яті процесу після тестування update best

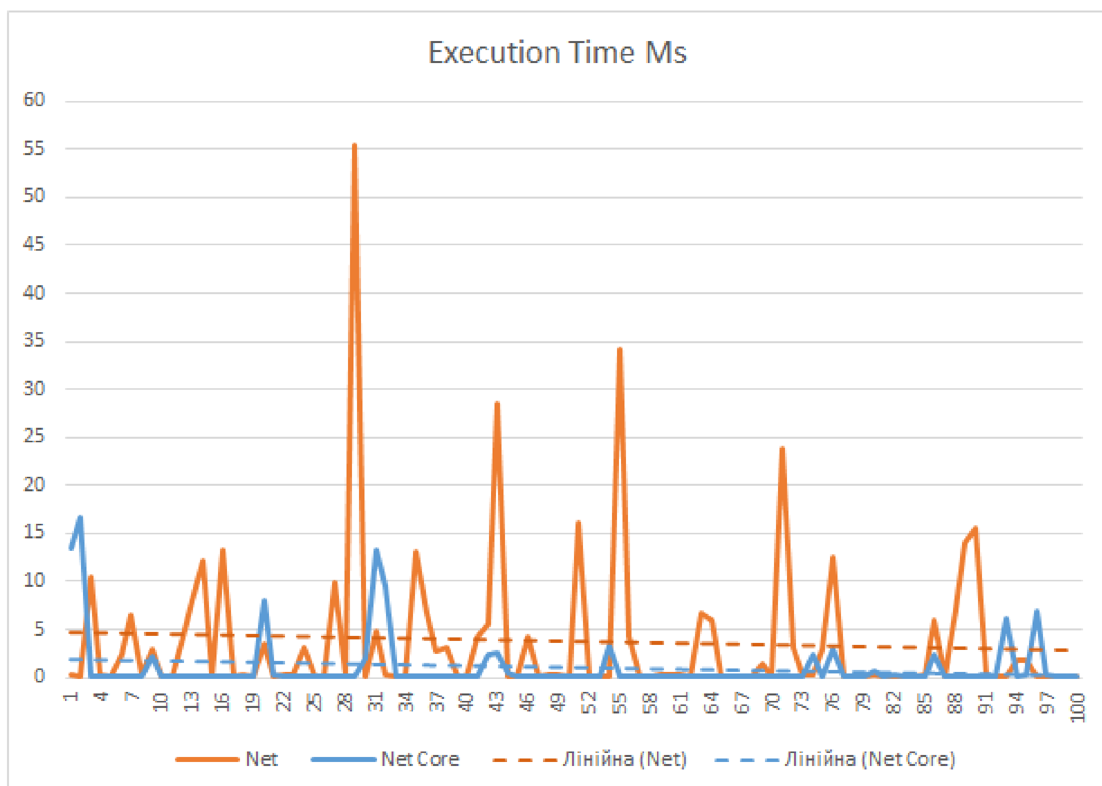

Рисунок 4.28 – графіки швидкості обробки запитів update best

4.4 Проведення експерименту з дослідження колекції «Черга»

Це дослідження також показує приблизно однакові показники відносно графіків порівняння, тому буде відображено лише графіки з дослідженням для великої кількості елементів (рис. 4.29 – 4.44).

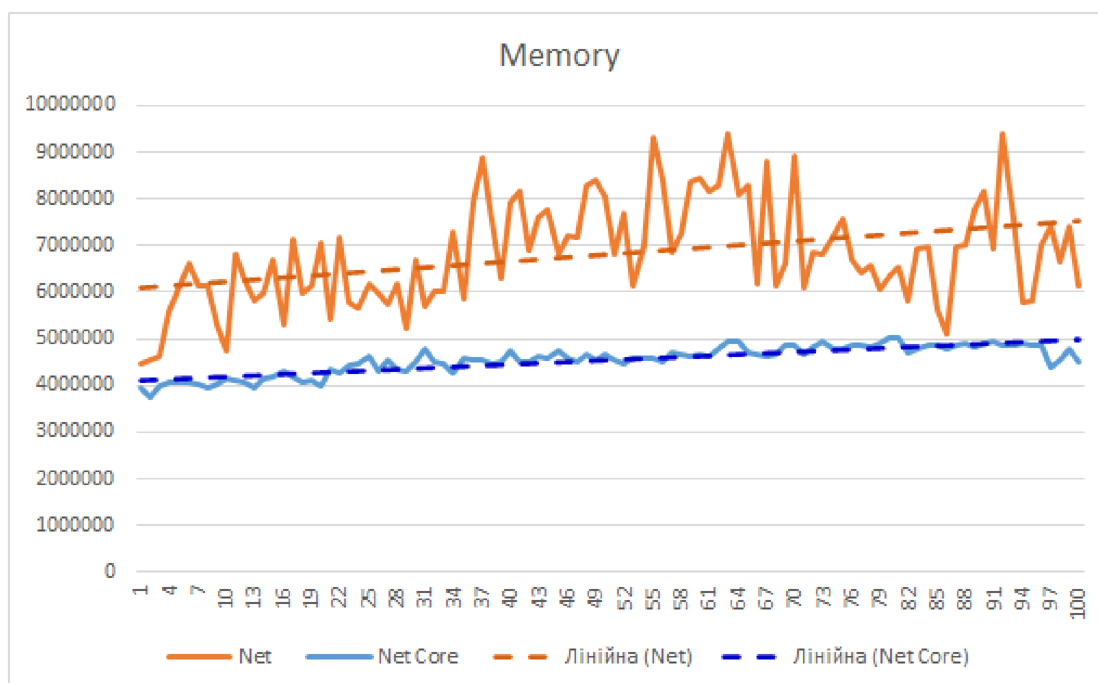

Рисунок 4.29 – графіки використання пам’яті збірником сміття add best

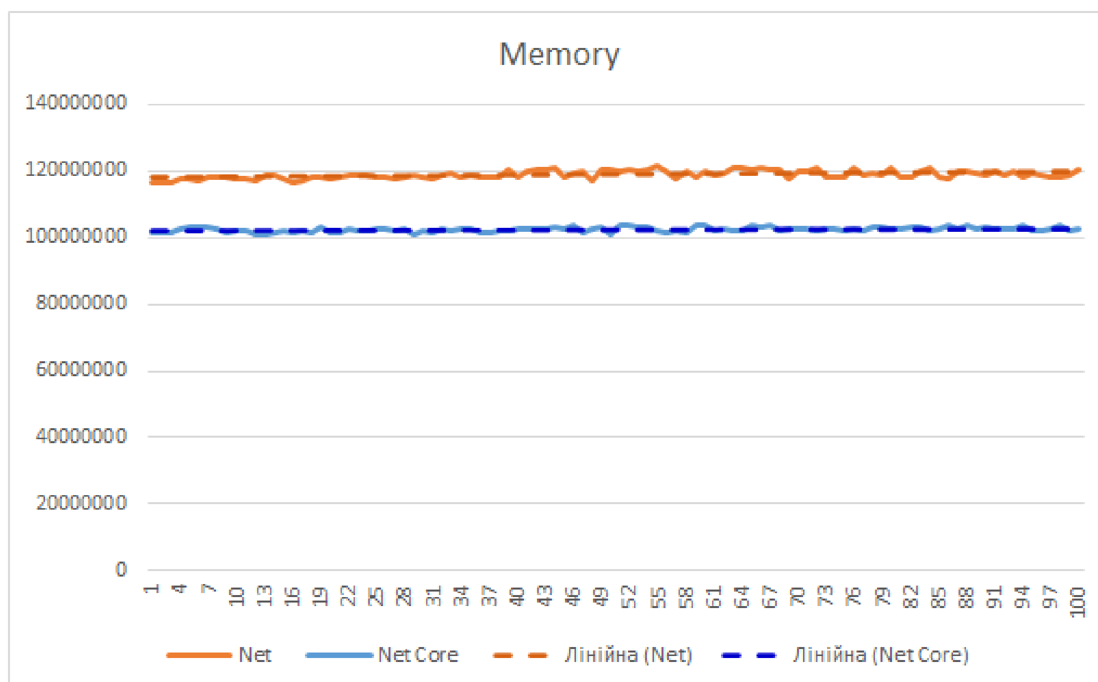

Рисунок 4.30 – графіки використання пам’яті процесу перед тестування add best

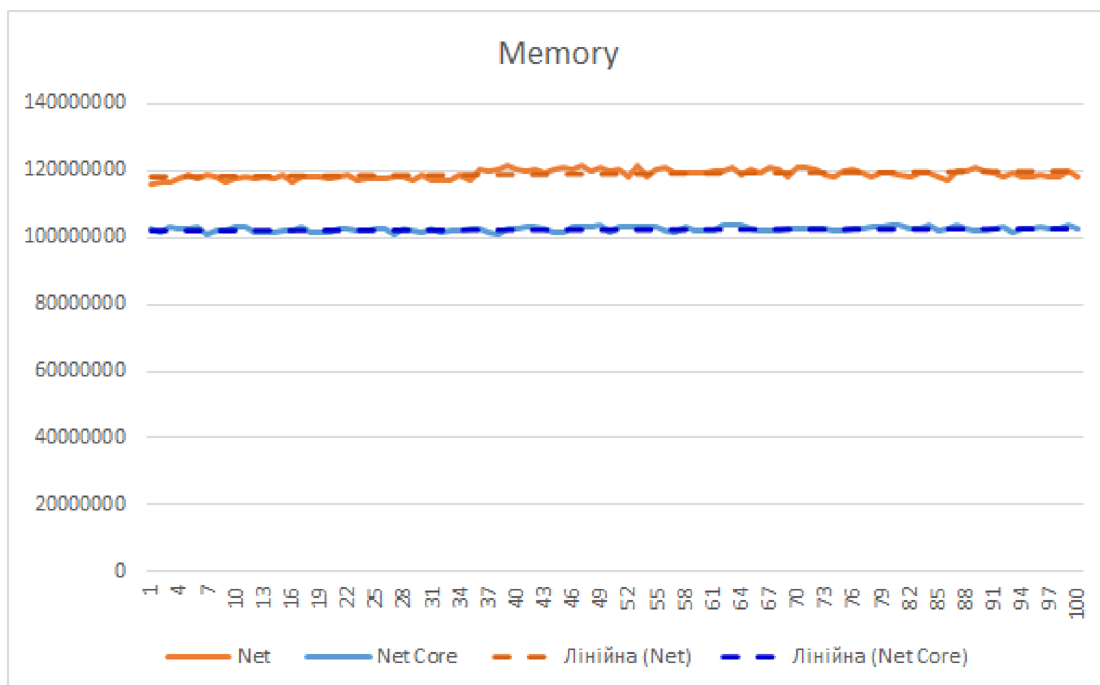

Рисунок 4.31 – графіки використання пам’яті процесу після тестування add best

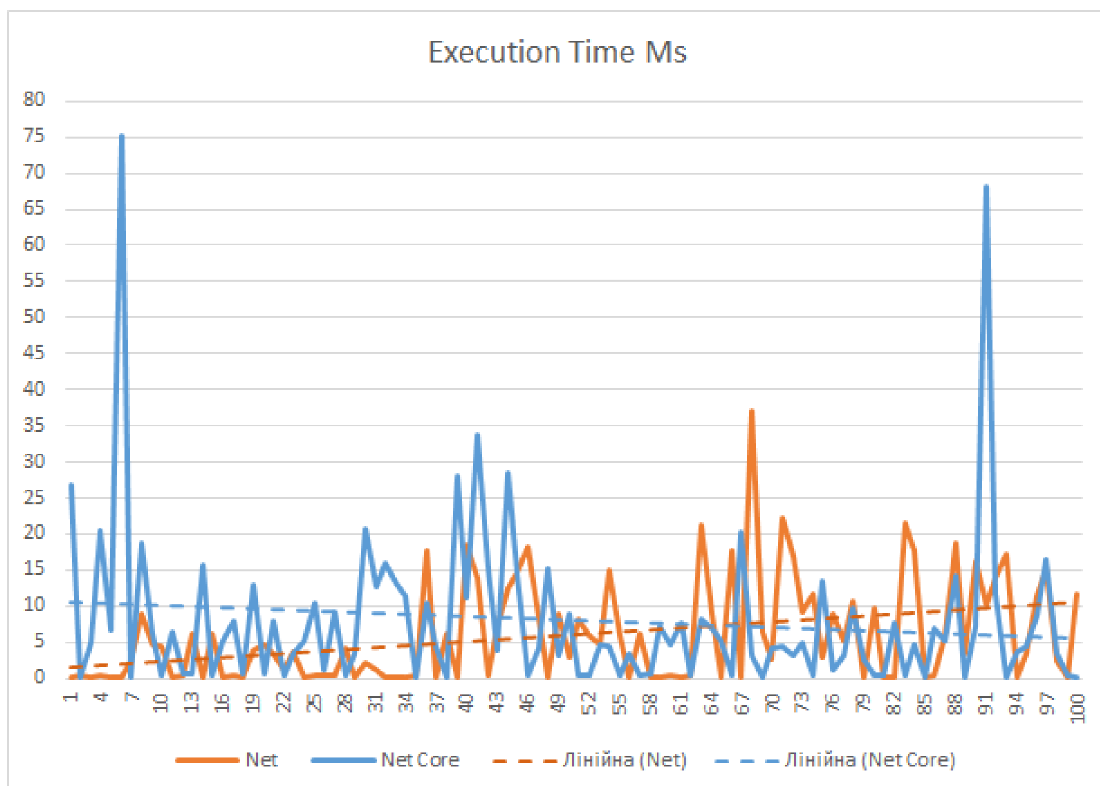

Рисунок 4.32 – графіки швидкості обробки запитів add best

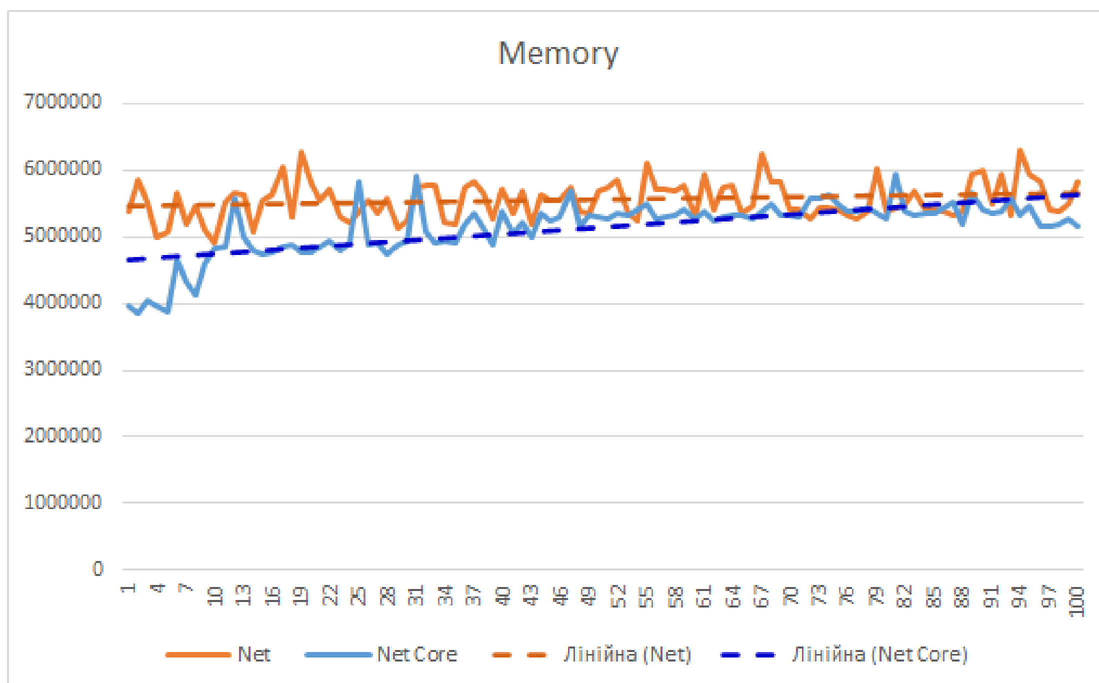

Рисунок 4.33 – графіки використання пам’яті збірником сміття add worst

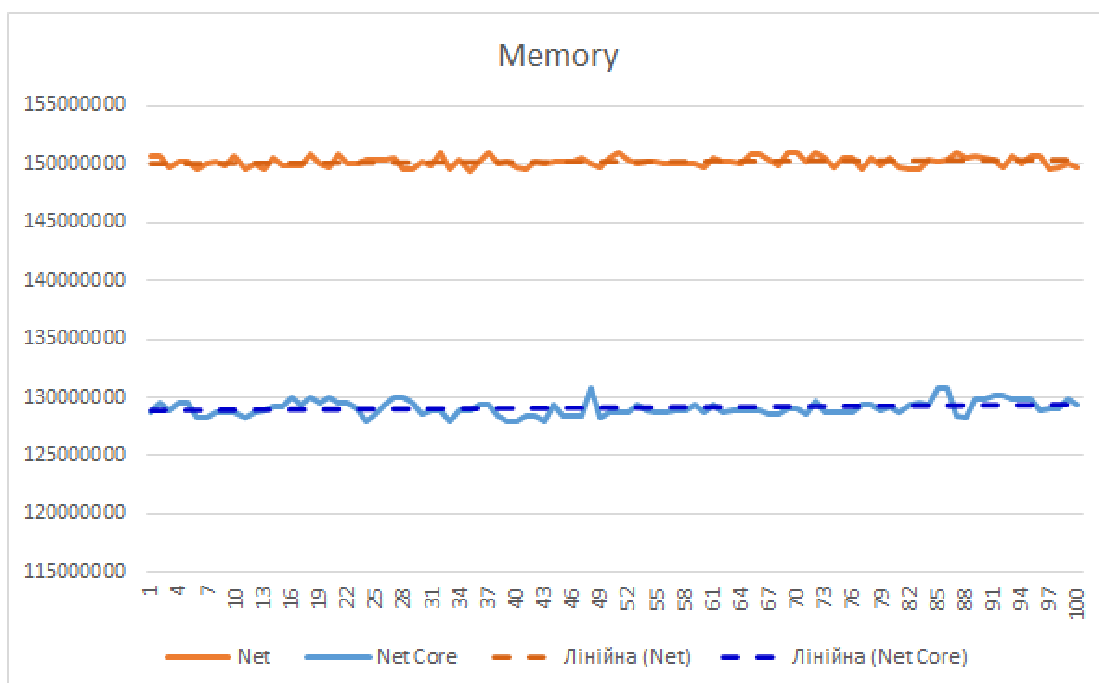

Рисунок 4.34 – графіки використання пам’яті процесу перед тестування add worst

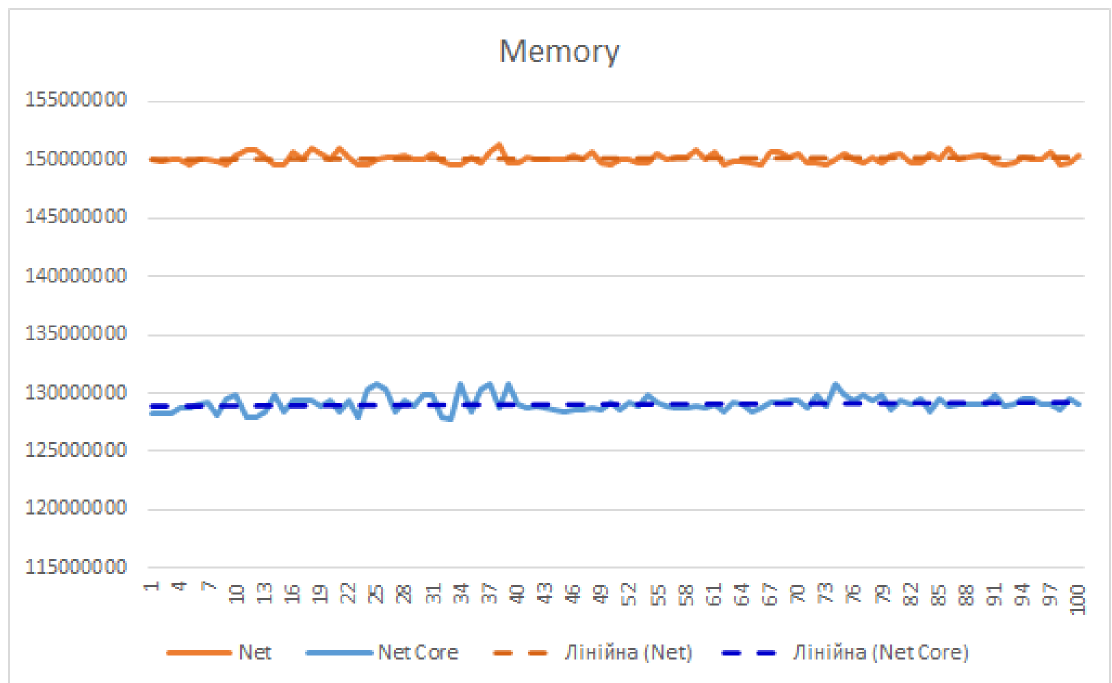

Рисунок 4.35 – графіки використання пам’яті процесу після тестування add worst

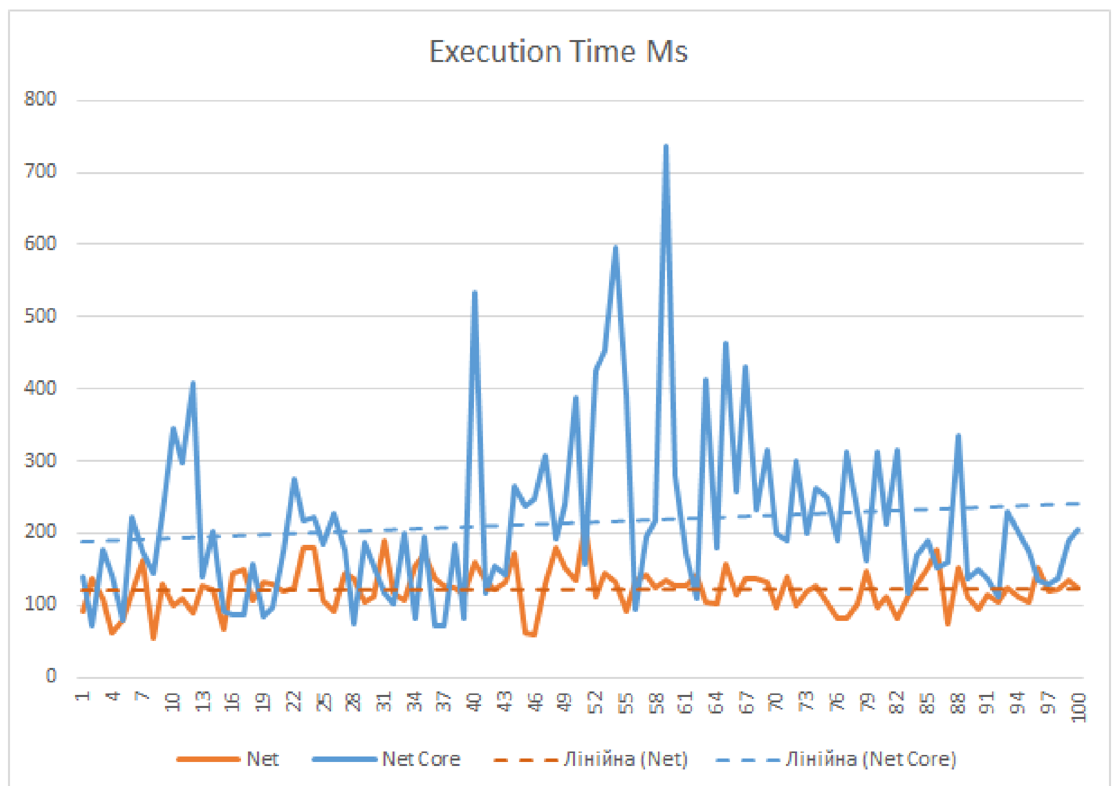

Рисунок 4.36 – графіки швидкості обробки запитів add worst

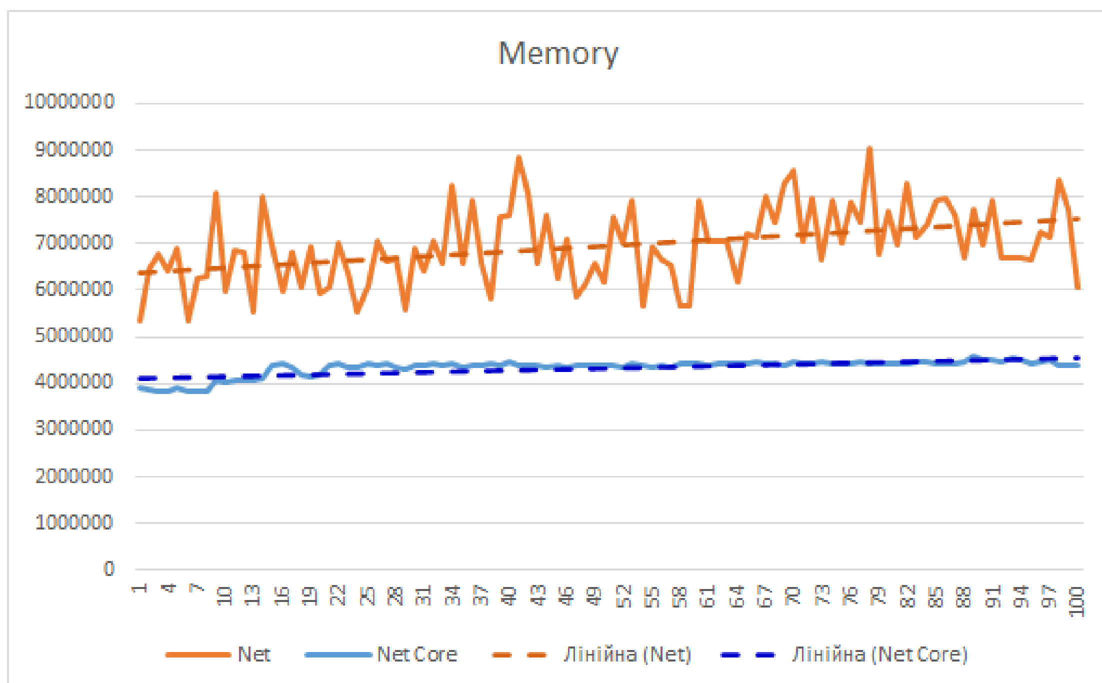

Рисунок 4.37 – графіки використання пам’яті збірником сміття remove best

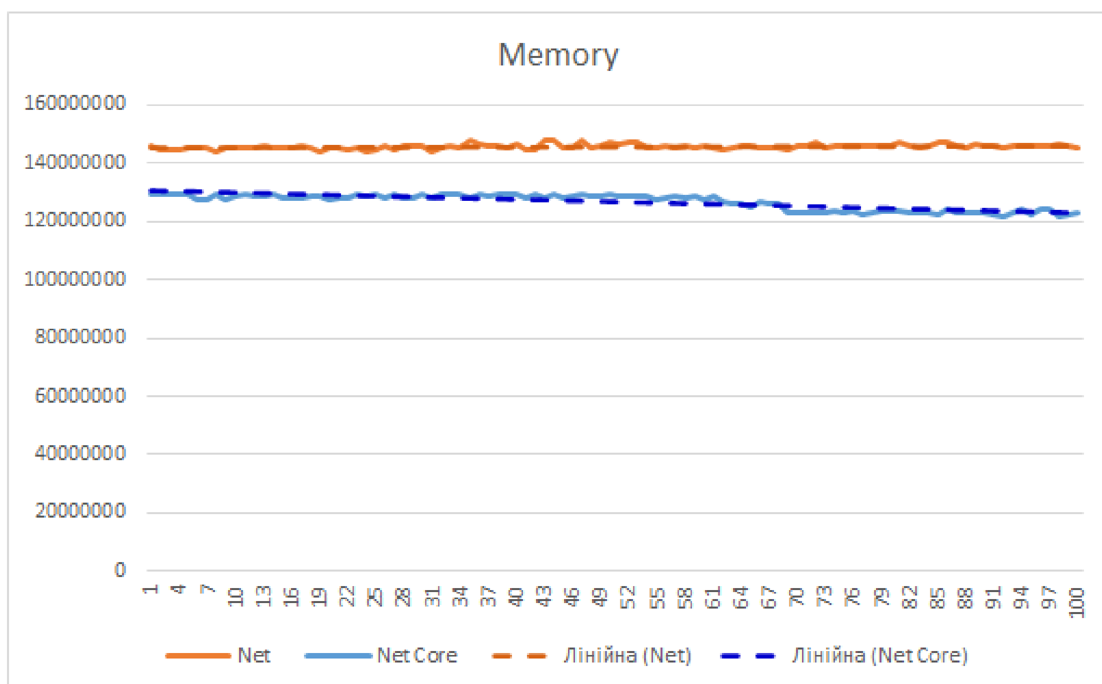

Рисунок 4.38 – графіки використання пам’яті процесу перед тестування remove best

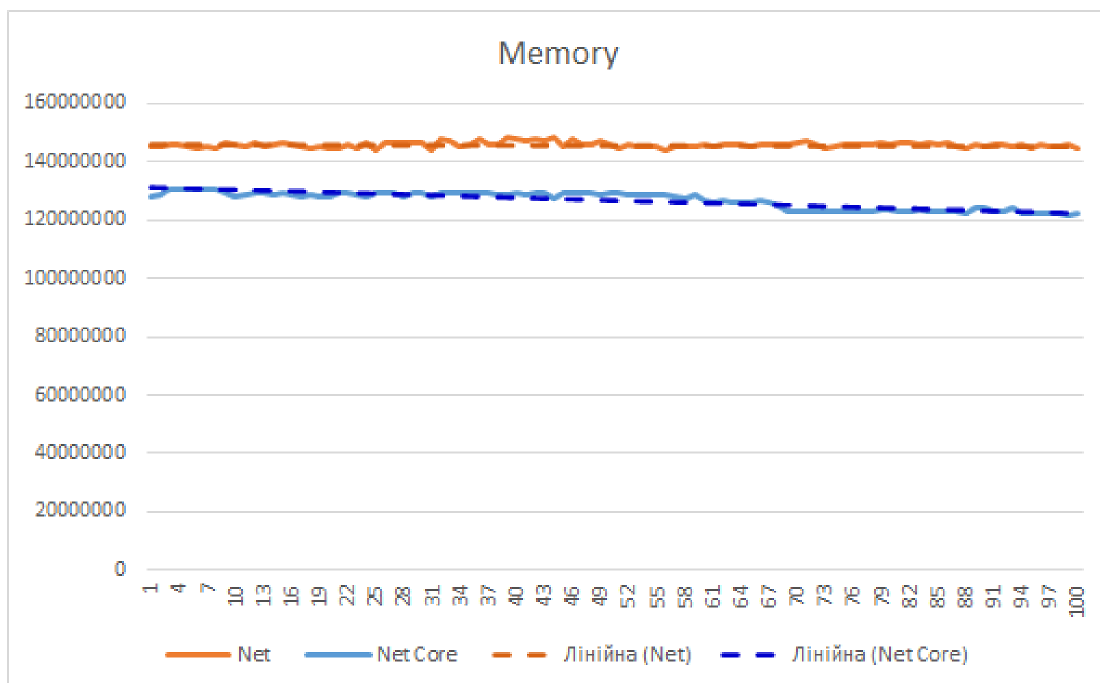

Рисунок 4.39 – графіки використання пам’яті процесу після тестування remove best

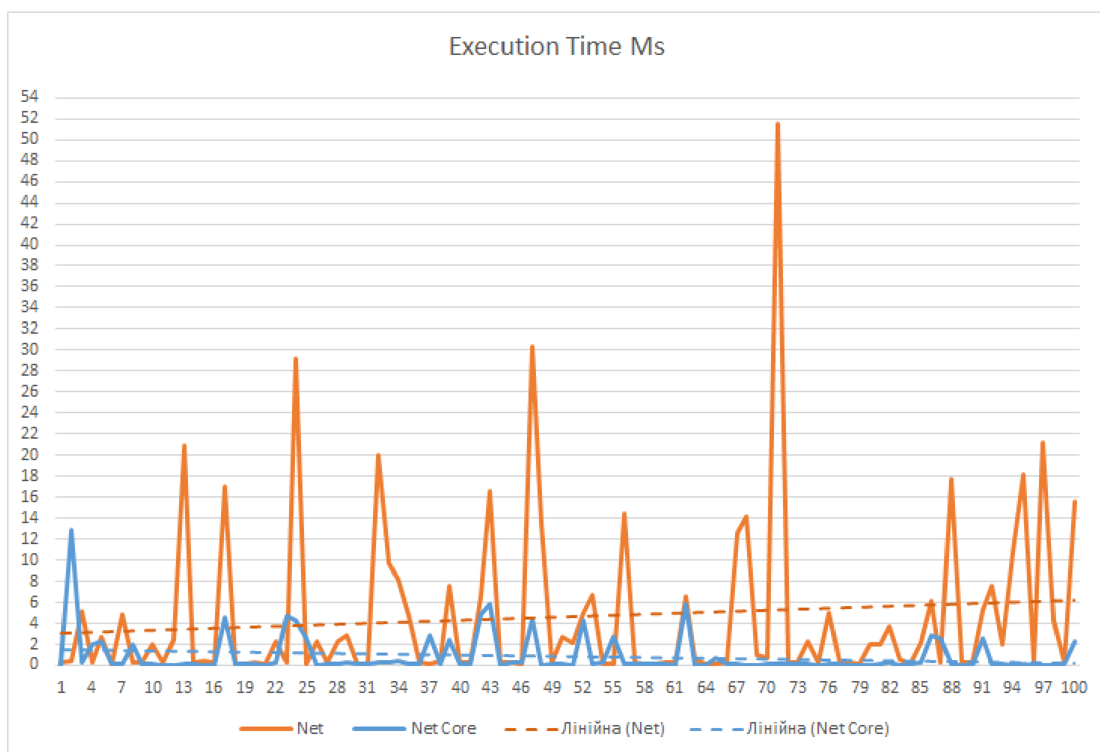

Рисунок 4.40 – графіки швидкості обробки запитів remove best

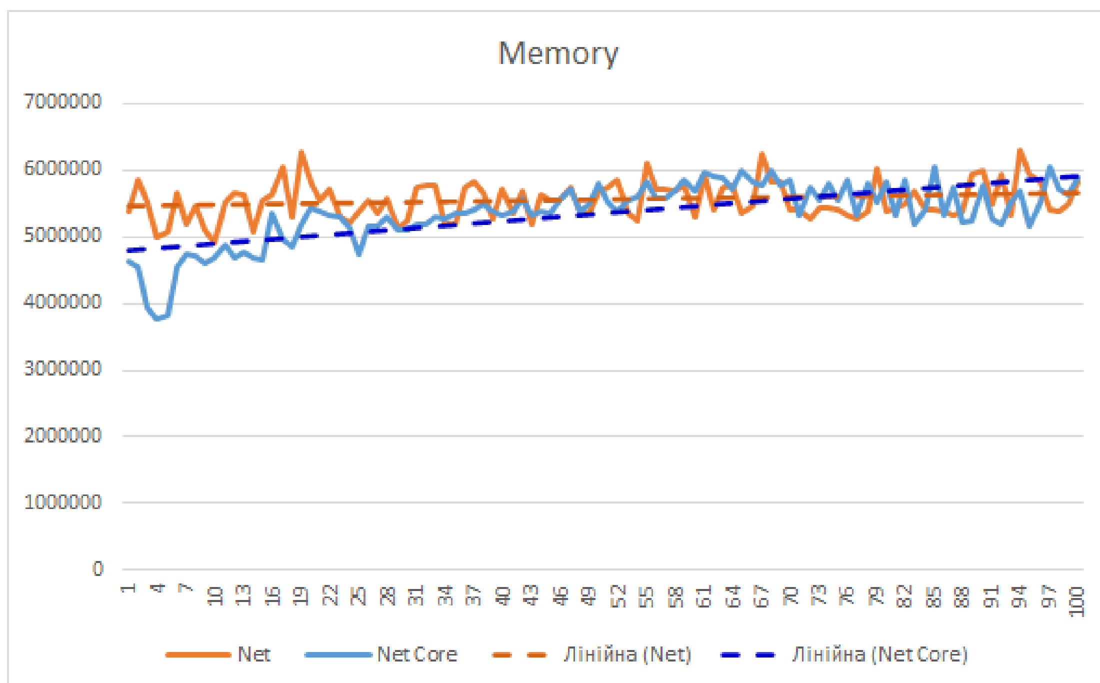

Рисунок 4.41 – графіки використання пам’яті збірником сміття update best

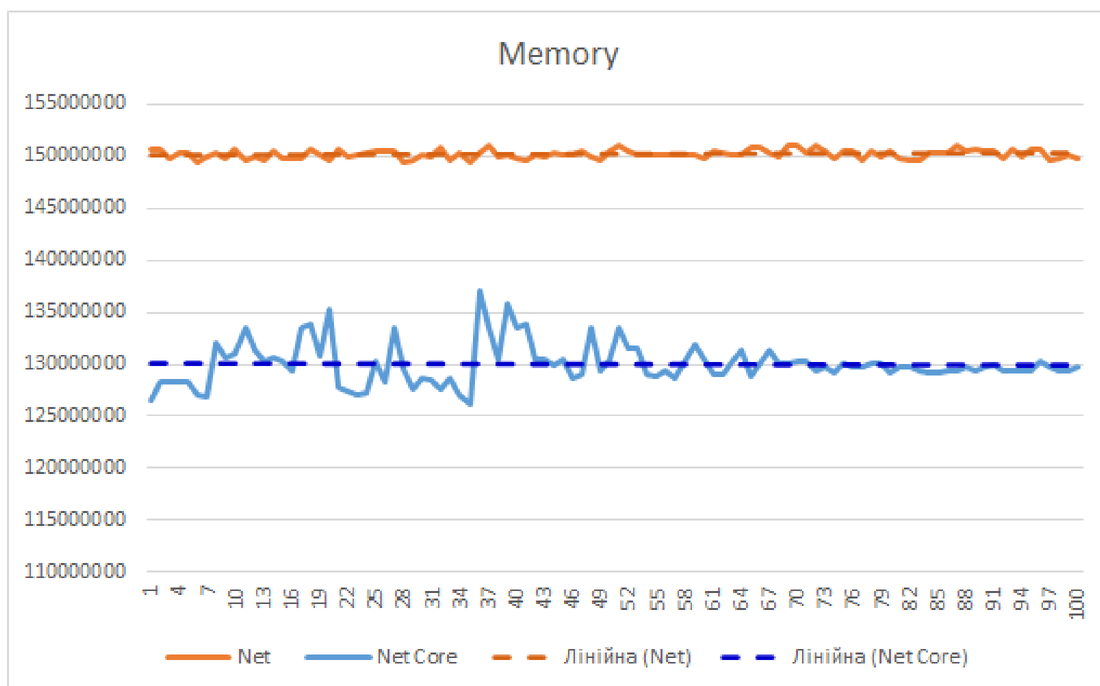

Рисунок 4.42 – графіки використання пам’яті процесу перед тестування update best

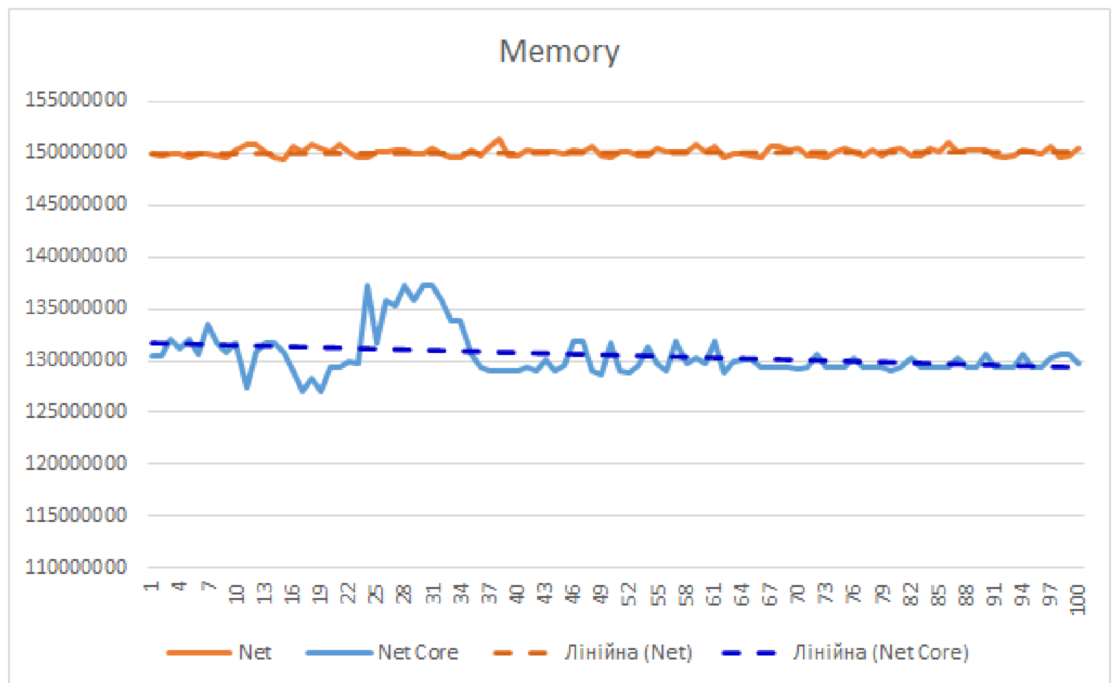

Рисунок 4.43 – графіки використання пам’яті процесу після тестування update best

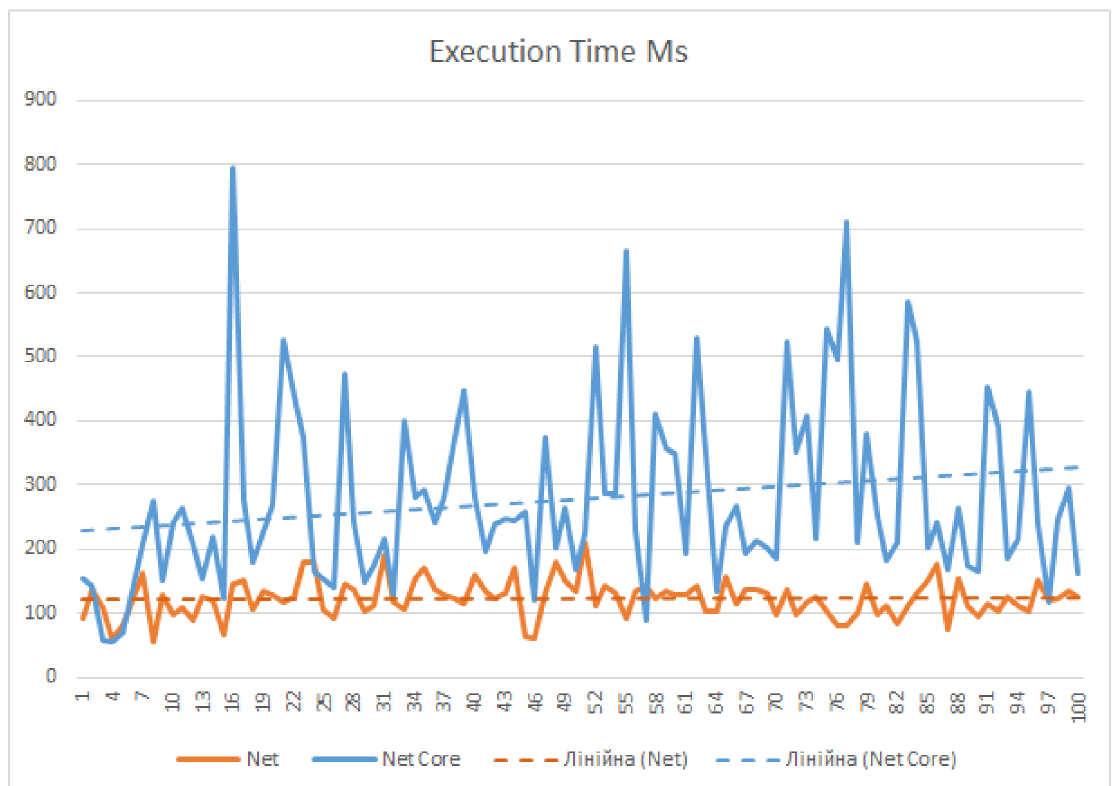

Рисунок 4.44 – графіки швидкості обробки запитів update best

4.5 Проведення експерименту з дослідження синхронного режиму відправки запиту

Таблиця 4.2 – Тестові сценарії синхронного режиму

Кількість запитів	**ASP.NET Core (ms)**	**ASP.NET (ms)**
100	90.48	280.74
1000	595.83	3100.58
10000	5630.41	38892.53
100000	41985.56	718788.21

4.6 Проведення експерименту з дослідження асинхронного режиму відправки запиту

Таблиця 4.3 – Тестові сценарії асинхронного режиму

Кількість запитів	**ASP.NET Core (ms)**	**ASP.NET (ms)**
100	50.335	110.93
1000	440.9913	1266.0468
10000	4428.4904	13829.9423
100000	35536.0521	159831.9476

4.7 Результати експериментів

Створимо таблиці (табл. 4.4 – 4.14) в яких буде вказано пояснення щодо результатів дослідження. Проаналізуємо кожну колекцію окремо за кожним показником, а саме: збірник сміття, обсяг пам’яті процесу перед тестуванням та після тестування. Також проаналізуємо швидкість обробки запитів з урахуванням особливостей організації роботи методів колекцій.

Щодо порівняння швидкості виконання синхронних та асинхронних запитів проаналізуємо таблиці 4.2 – 4.3, в яких записані часові показники виконання найпростішого з можливих запитів на отримання порожнього списку елементів. За результатами визначимо наскільки швидше чи повільніше виконуються запити на платформах і додатково перевіримо, які запити виконуються швидше, що дозволить відповісти на питання, які типи методів краще використовувати при розробці проектів.

Сценарій Add Best – додання елементу в порожній словник.

Таблиця 4.4 – Результати дослідження колекції «словник» дії Add Best

Показник	Результати
Збірник сміття	За проведеними дослідженнями лінія тренду показує, що ASP.NET Core витрачає набагато менше пам'яті ніж версія ASP.NET. Проте при збільшенні кількості запитів обсяг сміття стає більшим швидше ніж у ASP.NET.
Процес перед тестуванням	ASP.NET Core займає майже вдвічі менше пам'яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає майже вдвічі менше пам'яті на виконання поставленої задачі.
Швидкість обробки запитів	На початку експерименту обидві платформи виконували запити майже з однаковою швидкістю, але далі ASP.NET почав витрачати набагато більше часу.

Сценарій Add Worst – додання елементу в словник зі збільшенням значення ключа.

Таблиця 4.5 – Результати дослідження колекції «словник» дії Add Worst

Показник	Результати
Збірник сміття	На початку дослідження ASP.NET Core займав менше пам'яті, але далі пам'ять ставала все більшою й навіть стала більше ніж у ASP.NET.
Процес перед тестуванням	ASP.NET Core займає трохи менше пам'яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає трохи менше пам'яті на виконання поставленої задачі.
Швидкість обробки запитів	В цьому дослідженні ASP.NET Core показав великі затримки при обробці запитів в порівнянні з його попередником

Сценарій Remove Worst – видалення елементу зі словника від найбільшого значення ключа до найменшого.

Таблиця 4.6 – Результати дослідження колекції «словник» дії Remove Worst

Показник	**Результати**
Збірник сміття	Зберігається тенденція збільшення обсягу пам’яті від кількості запитів для ASP.NET Core, але в цей раз обсяг пам’яті набагато менший ніж у ASP.NET.
Процес перед тестуванням	ASP.NET Core займає трохи менше пам’яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає трохи менше пам’яті на виконання поставленої задачі.
Швидкість обробки запитів	В цьому дослідженні ASP.NET показав занадто великі затримки при обробці запитів в порівнянні з його новішою версією

Сценарій Update Best – оновлення елементів послідовно зі збільшенням значення.

Таблиця 4.7 – Результати дослідження колекції «словник» дії Update Best

Показник	**Результати**
Збірник сміття	На початку дослідження ASP.NET Core займав менше пам’яті, але далі пам’ять ставала все більшою.
Процес перед тестуванням	ASP.NET Core займає трохи менше пам’яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає трохи менше пам’яті на виконання поставленої задачі.
Швидкість обробки запитів	В цьому дослідженні ASP.NET показав великі затримки при обробці запитів в порівнянні з його попередником

Отже підбиваючи підсумки дослідження можна сказати, що загалом ASP.NET Core є кращим вибором за всіма критеріями в порівнянні зі своїм попередником, особливо за критерієм використання пам’яті.

Сценарій Add Best – додання елементу в кінець списку.

Таблиця 4.8 – Результати дослідження колекції «список» дії Add Best

Показник	Результати
Збірник сміття	Збірники сміття працюють майже в однаковому ритмі, за виключенням того, що сміття у ASP.NET Core менше.
Процес перед тестуванням	ASP.NET Core займає менше пам'яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає менше пам'яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET Core в цьому дослідженні показав велику затримку при обробці запитів в порівнянні з попередником

Сценарій Remove Worst – видалення зі списку передостаннього елементу.

Таблиця 4.9 – Результати дослідження колекції «список» дії Remove Worst

Показник	Результати
Збірник сміття	Сміття у ASP.NET Core менше.
Процес перед тестуванням	ASP.NET Core займає менше пам'яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає менше пам'яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET в цьому дослідженні показав велику затримку при обробці запитів в порівнянні з новішою версією

Сценарій Update Best – оновлення першого елементу списку.

Таблиця 4.10 – Результати дослідження колекції «список» дії Update Best

Показник	Результати
Збірник сміття	Працюють майже однаково.
Процес перед тестуванням	ASP.NET Core займає менше пам'яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає менше пам'яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET в цьому дослідженні показав велику затримку при обробці запитів в порівнянні з новішою версією

Отже як і зі словником майже всі показники у ASP.NET Core щодо списку є кращими й перевага залишається за ним.

Сценарій Add Best – додає елементи до порожньої черги.

Таблиця 4.11 – Результати дослідження колекції «черга» дії Add Best

Показник	Результати
Збірник сміття	Сміття у ASP.NET Core менше.
Процес перед тестуванням	ASP.NET Core займає менше пам’яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає менше пам’яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET в цьому дослідженні показав більш швидку роботу при обробці запитів в порівнянні з новішою версією.

Сценарій Add Worst – додає елементи до черги, яка вже містить елементи, вставляючи нові елементи в середину черги, що призводить до пересортування.

Таблиця 4.12 – Результати дослідження колекції «черга» дії Add Worst

Показник	Результати
Збірник сміття	Сміття у ASP.NET Core менше, але при збільшені кількості запитів обсяг сміття стає все більше.
Процес перед тестуванням	ASP.NET Core займає набагато менше пам’яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає набагато менше пам’яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET в цьому дослідженні показав більш швидку роботу при обробці запитів в порівнянні з новішою версією.

Сценарій Remove Best – видаляє кожен елемент з черги.

Таблиця 4.13 – Результати дослідження колекції «черга» дії Remove Best

Показник	Результати
Збірник сміття	Сміття у ASP.NET Core менше і воно майже стабільно не збільшується на відмінну від платформи ASP.NET.

Продовження таблиці 4.13

Показник	Результати
Процес перед тестуванням	ASP.NET Core займає менше пам’яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає менше пам’яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET Core в цьому дослідженні показав більш швидку роботу при обробці запитів в порівнянні з ASP.Net.

Сценарій Update Best – оновлює кожен елемент в черзі.

Таблиця 4.14 – Результати дослідження колекції «черга» дії Update Best

Показник	Результати
Збірник сміття	Збірник сміття працює майже однаково.
Процес перед тестуванням	ASP.NET Core займає набагато менше пам’яті на виконання поставленої задачі.
Процес після тестуванням	ASP.NET Core займає набагато менше пам’яті на виконання поставленої задачі.
Швидкість обробки запитів	ASP.NET в цьому дослідженні показав більш швидку роботу при обробці запитів в порівнянні з новішою версією.

Дослідження показало, що швидкість обробки й обсяги використання пам’яті у ASP.NET Core здебільшого є більш оптимізованими ніж у ASP.NET.

Отже всі 3 колекції майже у всіх показниках для платформи ASP.NET Core є кращими ніж для платформи ASP.NET, тому робимо висновок, що перевага при виборі платформи для створення проекту віддається ASP.NET Core.

Враховуючи визначені показники можна зазначити, що обрання платформи ASP.NET Core збереже багато коштів власнику проекту, якщо додаток планується розгортати у хмарному сховищі і загалом сервер буде працювати швидше на цій платформі.

Перейдемо до дослідження синхронного та асинхронного режиму відправки запитів. Проаналізувавши результати дослідження робимо такі висновки:

– ASP.NET Core виявився значно ефективнішим у порівнянні з ASP.NET в обох режимах – синхронному та асинхронному;
– у синхронному режимі з ростом кількості запитів спостерігається збільшення часу виконання для обох платформ. Однак, у випадку ASP.NET, цей приріст є значно більшим;
– у випадку асинхронного режиму, хоча час виконання також зростає зі збільшенням кількості запитів, ASP.NET Core все ще показує кращі результати, зберігаючи високу продуктивність порівняно з ASP.NET;
– швидкість обробки запитів у асинхронного режимі є більшою ніж при синхронному режимі, тому варто зазначити, що при можливості програмісти повинні використовувати саме асинхронні методи у своїй роботі, це дозволить поліпшити час відгуку серверу на запити клієнта і відповідно дозволить швидше завантажувати інформацію кінцевому користувачу.

Отже, використання асинхронного програмування може значно покращити продуктивність в обох платформах, але ASP.NET Core загалом показує кращу ефективність в порівнянні з платформою ASP.NET.

ВИСНОВКИ

Дослідження та порівняння платформ ASP.NET та ASP.NET Core стали важливим етапом для визначення оптимальної технології для розробки проектів. У досліджені було перевірено різноманітні аспекти продуктивності, такі як швидкість обробки, ефективність використання пам'яті, а також аналіз синхронного та асинхронного виконання запитів.

Розглянемо кожен аспект окремо та зробимо висновки, спираючись на отримані результати.

Вибір платформи: дослідження показало, що як швидкість обробки, так і обсяги використання пам'яті у ASP.NET Core зазвичай є більш оптимізованими, ніж у ASP.NET. Враховуючи це, усі три колекції даних демонструють кращі показники ефективності для платформи ASP.NET Core. Зроблено висновок, що при виборі платформи для створення проекту слід віддати перевагу ASP.NET Core.

Економія коштів і продуктивність: враховуючи визначені показники, можна визначити, що використання ASP.NET Core може заощадити власникові проекту значну суму коштів, особливо якщо додаток розробляється для розгортання в хмарному середовищі. Враховуючи загальні результати, система буде працювати швидше на платформі ASP.NET Core.

Дослідження синхронного та асинхронного режимів: дослідження синхронного та асинхронного виконання запитів показало, що ASP.NET Core виявився значно ефективнішим порівняно з ASP.NET у обох режимах.

Час виконання та кількість запитів: у синхронному режимі з ростом кількості запитів час виконання збільшується для обох платформ, але приріст для ASP.NET є більш значущим. У випадку асинхронного режиму ASP.NET Core показує кращі результати, зберігаючи високу продуктивність порівняно з ASP.NET.

Використання асинхронного програмування: швидкість обробки запитів у асинхронному режимі є значно вищою, ніж у синхронному. Рекомендується

використовувати асинхронні методи, що дозволяє поліпшити час відгуку серверу на запити клієнта.

Висновок щодо асинхронного програмування: використання асинхронного програмування може значно покращити продуктивність в обох платформах. Однак ASP.NET Core виявився більш ефективним у порівнянні з платформою ASP.NET навіть у випадку асинхронного виконання.

Загальним висновком є те, що обрання ASP.NET Core для розробки проекту надає великі переваги в порівнянні з платформою ASP.NET, а саме: покращення продуктивності, ефективності використання ресурсів та економії коштів для власників проектів, особливо при використанні у хмарному середовищі.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. https://devblogs.microsoft.com/dotnet/performance-improvements-in-aspnet-core-7
2. https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_5
3. Shvets, A. Dive Into DESIGN PATTERNS [Текст]. – NY: Refactoring.Guru, 2022. – 410 с.
4. Freeman, A. Pro ASP.NET CORE MVC [Текст]. – NY.: Apress, 2016. – 1018 с.
5. Freeman, A. Pro ASP.NET CORE 6. Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages [Текст]. – NY.: Apress, 2022. – 1286 с.
6. Jeffrey, R. CLR via C# (Developer Reference) [Текст]. – NY.: Microsoft Press, 2012. – 1443 с.
7. Freeman, A. Pro jQuery 2.0 (Expert's Voice in Web Development) [Текст]. – NY.: Apress, 2013. – 1651 с.

ДОДАТКИ

Технічне завдання
ДОДАТОК А

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET ТА ASP NET CORE

Технічне завдання
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1344-01-ЛЗ

Завідувач кафедри КІТ
_________Вадим ГОРЯЧКІН

Керівник розробки
_________Олександр ІВАНОВ

Виконавець
_________Олександр ГЕТМАНЕНКО

Нормоконтролер
_________Світлана ВОЛКОВА

2024

ЗАТВЕРДЖЕНО
44165850.1344-01-ЛЗ

АНАЛІЗ ШВИДКОДІЙНОСТІ
ПЛАТФОРМ ASP NET ТА ASP NET
CORE

Технічне завдання

Листів 15

2024

АНОТАЦІЯ

Документ 44165850.1344-01 «АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET ТА ASP NET CORE. Технічне завдання».

У даному документі представлені призначення та область застосування програмного продукту, основні вимоги до виконання проекту.

44165850.1344-01

ЗМІСТ

ВСТУП 4
1 ПІДСТАВИ ДЛЯ РОЗРОБКИ 5
2 ПРИЗНАЧЕННЯ РОЗРОБКИ 6
3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ 7
3.1 Вимоги до функціональних характеристик 7
3.2 Вимоги до вхідних даних 8
3.3 Вимоги до вихідних даних 9
3.4 Вимоги до надійності 12
3.5 Умови експлуатації 12
3.6 Вимоги до складу та параметрів технічних засобів 13
3.7 Вимоги до інформаційної та програмної сумісності 14
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 15

ВСТУП

Сфера веб-розробки продовжує постійно розвиватись, з'являється все більше технологій веб-розробки, однією з таких є .NET, зокрема Asp.net та Asp.net Core, враховуючі, що ці дві технології є новими, особливо Asp.net Core, порівняння їх швидкодійності є надзвичайно важливим для розробників та користувачів.

Швидкість роботи будь-якого додатку є важливою характеристикою програмного забезпечення. Якщо веб-додаток працює повільно, це зменшує ефективність роботи з ним й збільшує ймовірність втрати користувачів. Теж саме стосується й сфери розробки, в залежності від обраної технології розробник може витрати багато часу для створення свого додатку, зручність написання коду, швидкість роботи обраної платформи та її можливості є одними з ключових факторів розробки.

Порівняння швидкодійності платформ Asp.net та Asp.net Core може допомогти розробникам обрати, яка з цих платформ є більш оптимальною для виконання поставленого завдання.

Окрім того, зростаюча кількість веб-додатків, що працюють в режимі реального часу, підвищує значимість швидкодійності. У таких веб-додатках необхідно забезпечити максимальну швидкість відгуку системи на запити користувачів.

1 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від 05.12.2022 р. «Про призначення керівників та затвердження магістерських робіт», затверджений ректором Українського державного університету науки і технологій проф. Пшіньком О. М.

Тема роботи – «АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET TA ASP NET CORE».

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

Функціональне призначення:

1. Вимірювання продуктивності: програма розроблена для вимірювання швидкодійності операцій додавання, оновлення та видалення елементів в різних типах колекцій, таких як списки, черги та словники. Функціонал для вимірювання часу виконання операцій та обсягу використаної пам'яті дозволяє отримати об'єктивні дані про продуктивність платформ Asp.net та Asp.net Core.
2. Аналіз продуктивності при великих обсягах даних: програма надає можливість проводити дослідження продуктивності на різних обсягах даних, що допомагає визначити вплив розміру даних на ефективність операцій. Такий аналіз є важливим для розробників, які працюють з великими обсягами даних.

Експлуатаційне призначення:

1. Використання для досліджень: програма призначена для використання в дослідницьких цілях, зокрема для аналізу продуктивності платформ Asp.net та Asp.net Core. Вона може бути корисною для аналітиків, розробників, тестувальників та інших спеціалістів, які цікавляться порівнянням ефективності цих платформ.
2. Підтримка розширення: програма має модульну структуру та дозволяє легко додавати нові типи колекцій та методи вимірювання продуктивності без змінення існуючого коду. Це дозволяє розробникам розширювати її функціональність для аналізу інших аспектів продуктивності.
3. Порівняння платформ Asp.net та Asp.net Core: програма надає можливість порівнювати швидкодійність операцій між різними версіями платформ Asp.net та Asp.net Core. Це допомагає розробникам та аналітикам приймати обґрунтовані рішення щодо вибору платформи для розробки веб-додатків.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Функціональні вимоги системи:

- Вимірювання часу виконання операцій: програма повинна здійснювати вимірювання часу виконання операцій додавання, оновлення та видалення елементів в різних типах колекцій. Це вимірювання дозволяє отримати інформацію про час, який потрібно для виконання цих операцій на платформах Asp.net та Asp.net Core.
- Вимірювання обсягу використаної пам'яті: програма має проводити вимірювання обсягу використаної пам'яті під час виконання операцій. Це допомагає визначити, скільки пам'яті використовується при виконанні кожної операції та загалом при використанні різних типів колекцій.
- Аналіз продуктивності при великих обсягах даних: програма повинна підтримувати можливість проведення досліджень при різних обсягах даних. Це означає, що користувач може вибирати розмір та обсяги даних для аналізу продуктивності.
- Легкість розширення: програма повинна бути легко розширюваною для додавання нових типів колекцій та методів вимірювання продуктивності без змінення існуючого коду. Це забезпечує гнучкість та можливість адаптації до нових потреб.
- Вивід результатів в зрозумілому форматі: програма має надавати результати вимірювань продуктивності у зрозумілому форматі, наприклад, у вигляді структурованих об'єктів формату JSON або звітів. Це допомагає користувачам аналізувати та інтерпретувати результати досліджень у інших програмах.
- Можливість очищення колекцій: програма має включати можливість очищення колекцій після виконання тестів, щоб забезпечити коректність результатів дослідження.

Обмеження системи та додаткові можливості:

– забезпечення валідації введеної інформації та повідомлення про некоректність введених даних при запиті не через інтерфейс Swagger.

3.2 Вимоги до вхідних даних

Таблиця 3.1 – Вхідні дані для Dictionary API

Endpoint	**Parameters**	**Request Body (application/json)**
GET /api/dictionary	No parameters	-
POST /api/dictionary	No parameters	{"key": 0, "value": 0}
PUT /api/dictionary	key (integer), newItem (integer)	-
DELETE /api/dictionary	No parameters	{"key": 0, "value": 0}
GET /api/dictionary/add/best	maxSize (integer, default: 134217728)	-
GET /api/dictionary/add/worst	maxSize (integer, default: 134217728)	-
GET /api/dictionary/update/best	maxSize (integer, default: 134217728)	-
GET /api/dictionary/remove/worst	maxSize (integer, default: 134217728)	-

Таблиця 3.2 – Вхідні дані для List API

Endpoint	**Parameters**	**Request Body (application/json)**
GET /api/list	No parameters	-
POST /api/list	item (integer)	-
PUT /api/list	index (integer), newItem (integer)	-
DELETE /api/list	item (integer)	-
GET /api/list/add/best	maxSize (integer, default: 134217728)	-
GET /api/list/update/best	maxSize (integer, default: 134217728)	-
GET /api/list/remove/worst	maxSize (integer, default: 134217728)	-

Таблиця 3.3 – Вхідні дані для Queue API

Endpoint	Parameters	Request Body (application/json)
GET /api/queue	No parameters	-
POST /api/queue	item (integer)	-
PUT /api/queue	index (integer), newItem (integer)	-
DELETE /api/queue	item (integer)	-
GET /api/queue/add/best	maxSize (integer, default: 50000)	-
GET /api/queue/add/worst	maxSize (integer, default: 50000)	-
GET /api/queue/update/best	maxSize (integer, default: 50000)	-
GET /api/queue/remove/best	maxSize (integer, default: 50000)	-

Таблиця 3.4 – Вхідні дані для Test API

Endpoint	Parameters	Request Body (application/json)
GET /api/Test/api-tester	request (string, query) numberOfRequests (integer, query)	-
GET /api/Test/api-tester-timing	request (string, query) numberOfRequests (integer, query)	-
GET /api/Test/api-tester-timing-async	request (string, query) numberOfRequests (integer, query)	-

3.3 Вимоги до вихідних даних

Таблиця 3.5 – Вихідні дані для Dictionary API

Endpoint	Response Code	Response Body (application/json)
GET /api/dictionary	200	[]
POST /api/dictionary	200	-
PUT /api/dictionary	200	-
DELETE /api/dictionary	200	-

Продовження таблиці 3.5

Endpoint	Response Code	Response Body (application/json)
GET /api/dictionary/add/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/dictionary/add/worst	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/dictionary/update/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/dictionary/remove/worst	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}

Таблиця 3.6 – Вихідні дані для List API

Endpoint	Response Code	Response Body (application/json)
GET /api/list	200	-
POST /api/list	200	-
PUT /api/list	200	-
DELETE /api/list	200	-
GET /api/list/add/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/list/update/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}

Продовження таблиці 3.6

Endpoint	Response Code	Response Body (application/json)
GET /api/list/remove/worst	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}

Таблиця 3.7 – Вихідні дані для Queue API

Endpoint	Response Code	Response Body (application/json)
GET /api/queue	200	-
POST /api/queue	200	-
PUT /api/queue	200	-
DELETE /api/queue	200	-
GET /api/queue/add/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/queue/add/worst	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/queue/update/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}
GET /api/queue/remove/best	200	{"testName": "string", "metrics": {"additionalProp1": ["string"], "additionalProp2": ["string"], "additionalProp3": ["string"]}}

Таблиця 3.8 – Вихідні дані для Test API

Endpoint	**Response Code**	**Response Body**
GET /api/Test/api-tester	200	[{"testName": "string", "metrics": {"testExecutionTime": [{"executionTimeMs": 0}], "memory": [{"title": "string", "size": 0, "type": "string"}]}}]
GET /api/Test/api-tester-timing	200	{"executionTimeMs": 0}
GET /api/Test/api-tester-timing-async	200	{"executionTimeMs": 0}

3.4 Вимоги до надійності

Вимоги до надійності наступні:

– контроль ведених даних у полів введення: необхідно забезпечити механізм контролю та валідації введених даних в полі введення. Це допоможе уникнути некоректних даних та можливих помилок при їх обробці;

– повідомлення про стан роботи та результати операцій: при оновленні даних, система повинна забезпечувати відображення відповідних повідомлень, що інформують користувача про стан роботи програми та результати виконаних операцій. Це дозволить користувачеві бути в курсі процесу та виявляти можливі проблеми;

– архівна копія тексту програми: слід забезпечити збереження архівної копії тексту програми. Це дозволить відновити вихідний код у разі втрати або пошкодження основної версії програми, забезпечуючи можливість швидкої відновлення роботи системи.

3.5 Умови експлуатації

Програмний продукт повинен використовуватись у приміщеннях, що відповідають умовам роботи електронно-обчислювальних машин (ЕОМ). Це

означає, що приміщення повинні мати відповідні кліматичні, санітарні та гігієнічні умови, відповідно до вимог, визначених у ДНАОП 0.00-1.31-99 (див. табл. 3.9).

Таблиця 3.9 – Кліматичні умови

Пора року	Категорія робіт згідно з ГОСТ 12.01-005-88	Температура повітря, град. С	Відносна вологість повітря, %	Швидкість руху повітря, м/с
		Оптимальна	Оптимальна	Оптимальна
Холодна	Легка-1-а	22-24	40-60	0,1
	Легка-1-б	21-23	40-60	0,1
Тепла	Легка-1-а	23-25	40-60	0,1
	Легка-1-б	22-24	40-60	0,2

Користувач, який працює з програмним продуктом, повинен мати навички роботи з персональним комп'ютером та бути ознайомленим з документацією до програми та процесом роботи з веб-інтерфейсом Swagger. Це допоможе забезпечити ефективне та безперебійне використання програмного продукту, оскільки користувач буде знати, як правильно взаємодіяти з програмою та виконувати необхідні дії.

3.6 Вимоги до складу та параметрів технічних засобів

Для використання розроблюваного програмного продукту вимагається наступне обладнання:

- мінімум 16 гігабайт оперативної пам'яті;
- браузер , що підтримується програмним продуктом;
- процесор: 32 або 64 розрядний процесор з тактовою частотою 2.8-4.1 ГГц або вище, який підтримує набір інструкцій SSE2;
- мінімум 256 гігабайт простору на жорсткому диску;
- засоби контролю, такі як миша, віддалений робочий стіл, клавіатура;
- монітор.

3.7 Вимоги до інформаційної та програмної сумісності

Програмний продукт, що розробляється, має підтримку для наступних операційних систем:

- Windows 7 та наступні версії (включно) - це охоплює широкий спектр операційних систем Windows, включаючи Windows 7, Windows 8 і Windows 10.

Для розробки програмного продукту використовується середовище розробки Visual Studio 2022 Professional, яке забезпечує розширені можливості для розробки програм на різних платформах.

У дослідженні використовуються платформи ASP.NET Framework 4.8 та ASP.NET Core Framework 7 (ASP.NET Core 7).

Для інтерфейсу API платформи ASP.NET Framework 4.8 використовуються бібліотеки Swashbuckle 5.6.0 та Swashbuckle.Core 5.6.0.

Для роботи механізму «ін’єкції залежності» у платформи ASP.NET Framework 4.8 використовується бібліотеки Unity 5.11.1 та Unity.WebAPI 5.4.0.

Для інтерфейсу API платформи ASP.NET Core 7 використовуються бібліотеки Swashbuckle.AspNetCore 6.5.0 та Microsoft.AspNetCore.OpenApi 7.0.7.

Також, для користувачів програмного продукту необхідна наявність інтернет-браузера для доступу до функціональності програми через веб-інтерфейс.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Richter, J. CLR via C# (Developer Reference) [Текст] : – Redmond, WA.: Microsoft Press, 2018. – 896 с.
2. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю.М.Івченко, В.І.Шинкаренко, В.Г.Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В.Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В.Лазаряна, 2009. – 38 с.

ДОДАТОК Б

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET ТА ASP NET CORE

Керівництво користувача
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1344-01 ІЗ 01-ЛЗ

Завідувач кафедри КІТ
_________Вадим ГОРЯЧКІН

Керівник розробки
_________Олександр ІВАНОВ

Виконавець
_________Олександр ГЕТМАНЕНКО

Нормоконтролер
_________Світлана ВОЛКОВА

2024

ЗАТВЕРДЖЕНО
44165850.1344-01 ІЗ 01-ЛЗ

АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET ТА ASP NET CORE

Керівництво користувача

Листів 12

2024

44165850.1344-01 ІЗ 01

АНОТАЦІЯ

Документ 44165850.1344-01 ІЗ 01 «АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET ТА ASP NET CORE. Керівництво користувача».

У даному документі описані обов'язкові рекомендації з користування програмою. Програма написана на мові C#. Об'єм пам'яті, що займають програми комплексу складає від 16 ГБ. Конфігурація комп'ютер або ноутбук. Комплекс функціонує в середовищі всіх видів операційних систем сімейства Windows починаючи з версії Windows 7.

ЗМІСТ

1 ВСТУП 4

2 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ... 5

3 ОПИС ОПЕРАЦІЙ 7

3.1 Робота з Ендпоінтом /api/dictionary.. 7

3.2 Робота з Ендпоінтом /api/list 7

3.3 Робота з Ендпоінтом /api/queue 8

3.4 Робота з Ендпоінтом /api/test 8

3.5 Виконання досліду з використанням api-tester 9

3.6 Виконання досліду з використанням api-tester-timing та api-tester-timing-async 10

4 АВАРІЙНІ СИТУАЦІЇ 11

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 12

1 ВСТУП

Сфера веб-розробки продовжує постійно розвиватись, з'являється все більше технологій веб-розробки, однією з таких є .NET, зокрема Asp.net та Asp.net Core, враховуючі, що ці дві технології є новими, особливо Asp.net Core, порівняння їх швидкодійності є надзвичайно важливим для розробників та користувачів.

Швидкість роботи будь-якого додатку є важливою характеристикою програмного забезпечення. Якщо веб-додаток працює повільно, це зменшує ефективність роботи з ним й збільшує ймовірність втрати користувачів. Теж саме стосується й сфери розробки, в залежності від обраної технології розробник може витрати багато часу для створення свого додатку, зручність написання коду, швидкість роботи обраної платформи та її можливості є одними з ключових факторів розробки.

Порівняння швидкодійності платформ Asp.net та Asp.net Core може допомогти розробникам обрати, яка з цих платформ є більш оптимальною для виконання поставленого завдання.

Окрім того, зростаюча кількість веб-додатків, що працюють в режимі реального часу, підвищує значимість швидкодійності. У таких веб-додатках необхідно забезпечити максимальну швидкість відгуку системи на запити користувачів.

2 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

Функціональне призначення:

1) Вимірювання продуктивності: програма розроблена для вимірювання швидкодійності операцій додавання, оновлення та видалення елементів в різних типах колекцій, таких як списки, черги та словники. Функціонал для вимірювання часу виконання операцій та обсягу використаної пам'яті дозволяє отримати об'єктивні дані про продуктивність платформ Asp.net та Asp.net Core.
2) Аналіз продуктивності при великих обсягах даних: програма надає можливість проводити дослідження продуктивності на різних обсягах даних, що допомагає визначити вплив розміру даних на ефективність операцій. Такий аналіз є важливим для розробників, які працюють з великими обсягами даних.

Експлуатаційне призначення:

1) Використання для досліджень: програма призначена для використання в дослідницьких цілях, зокрема для аналізу продуктивності платформ Asp.net та Asp.net Core. Вона може бути корисною для аналітиків, розробників, тестувальників та інших спеціалістів, які цікавляться порівнянням ефективності цих платформ.
2) Підтримка розширення: програма має модульну структуру та дозволяє легко додавати нові типи колекцій та методи вимірювання продуктивності без змінення існуючого коду. Це дозволяє розробникам розширювати її функціональність для аналізу інших аспектів продуктивності.

Порівняння платформ Asp.net та Asp.net Core: програма надає можливість порівнювати швидкодійність операцій між різними версіями платформ Asp.net та Asp.net Core. Це допомагає розробникам та аналітикам приймати обґрунтовані рішення щодо вибору платформи для розробки веб-додатків.

Для використання розроблюваного програмного продукту вимагається наступне обладнання:

- мінімум 16 гігабайт оперативної пам'яті;
- браузер , що підтримується програмним продуктом;
- процесор: 32 або 64 розрядний процесор з тактовою частотою 2.8-4.1 ГГц або вище, який підтримує набір інструкцій SSE2;
- мінімум 256 гігабайт простору на жорсткому диску;
- засоби контролю, такі як миша, віддалений робочий стіл, клавіатура;
- монітор.

3 ОПИС ОПЕРАЦІЙ

3.1 Робота з Ендпоінтом /api/dictionary

Відкриття Swagger-інтерфейсу: відкрийте веб-браузер та перейдіть за адресою /api/swagger.

Обираємо «Dictionary» API:

– у списку доступних API знайдіть «Dictionary» та оберіть його;
– з доступних ендпоінтів, які починаються з /api/ dictionary оберіть необхідний;
– детально ознайомтеся із параметрами запиту, прикладами відповідей та схемою запиту.

Використання «Try it out»:

– використайте кнопку «Try it out» для введення параметрів та виконання запиту;
– перевірте відповідь та переконайтеся, що результати відповідають очікуваним.

3.2 Робота з Ендпоінтом /api/list

Відкриття Swagger-інтерфейсу: відкрийте веб-браузер та перейдіть за адресою /api/swagger.

Обираємо «List» API:

– у списку доступних API знайдіть «List» та оберіть його;
– з доступних ендпоінтів, які починаються з /api/list оберіть необхідний;
– детально ознайомтеся із параметрами запиту, прикладами відповідей та схемою запиту.

Використання «Try it out»:

– використайте кнопку «Try it out» для введення параметрів та виконання запиту;
– перевірте відповідь та переконайтеся, що результати відповідають очікуваним.

3.3 Робота з Ендпоінтом /api/queue

Відкриття Swagger-інтерфейсу: відкрийте веб-браузер та перейдіть за адресою /api/swagger.

Обираємо «Queue» API:

– у списку доступних API знайдіть «Queue» та оберіть його;
– з доступних ендпоінтів, які починаються з /api/ queue оберіть необхідний;
– детально ознайомтеся із параметрами запиту, прикладами відповідей та схемою запиту.

Використання «Try it out»:

– використайте кнопку «Try it out» для введення параметрів та виконання запиту;
– перевірте відповідь та переконайтеся, що результати відповідають очікуваним.

3.4 Робота з Ендпоінтом /api/test

Відкриття Swagger-інтерфейсу: відкрийте веб-браузер та перейдіть за адресою /api/swagger.

Обираємо «Test» API:

– у списку доступних API знайдіть «Test» та оберіть його;
– з доступних ендпоінтів оберіть необхідний /api/test/api-tester, /api/test/api-tester-timing або /api/test/api-tester-timing-async;
– детально ознайомтеся із параметрами запиту, прикладами відповідей та схемою запиту.

Використання «Try it out»:

– використайте кнопку «Try it out» для введення параметрів та виконання запиту;
– перевірте відповідь та переконайтеся, що результати відповідають очікуваним.

3.5 Виконання досліду з використанням api-tester

1) Відкриття Swagger-інтерфейсу: відкрийте веб-браузер та перейдіть за адресою /swagger (або /api/swagger, в залежності від налаштувань).
2) У списку доступних API знайдіть «Test» та оберіть його.
3) З доступних ендпоінтів оберіть /api/test/api-tester та натисність «Try it out».
4) Далі у полі request введіть один з запитів:
   – https://localhost:44361/api/dictionary/add/best;
   – https://localhost:44361/api/dictionary/add/worst;
   – https://localhost:44361/api/dictionary/update/best;
   – https://localhost:44361/api/dictionary/remove/worst;
   – https://localhost:44361/api/list/add/best;
   – https://localhost:44361/api/list/update/best;
   – https://localhost:44361/api/list/remove/worst;
   – https://localhost:44361/api/queue/add/best;
   – https://localhost:44361/api/queue/add/worst;
   – https://localhost:44361/api/queue/update/best;
   – https://localhost:44361/api/queue/remove/best;
   – https://localhost:7198/api/dictionary/add/best;
   – https://localhost:7198/api/dictionary/add/worst;
   – https://localhost:7198/api/dictionary/update/best;
   – https://localhost:7198/api/dictionary/remove/worst;
   – https://localhost:7198/api/list/add/best;
   – https://localhost:7198/api/list/update/best;
   – https://localhost:7198/api/list/remove/worst;
   – https://localhost:7198/api/queue/add/best;
   – https://localhost:7198/api/queue/add/worst;
   – https://localhost:7198/api/queue/update/best;
   – https://localhost:7198/api/queue/remove/best.

5) У полі numberOfRequests введіть кількість запитів, яку хочете виконати.
6) Натисніть кнопку Execute.
7) Дочекайтесь завершення запиту.
8) В результаті вам буде видано список пройдених тестів у форматі JSON, який далі ви можете експортувати до файлу JSON та використовувати в подальших дослідженнях.

3.6 Виконання досліду з використанням api-tester-timing та api-tester-timing-async

1) Відкриття Swagger-інтерфейсу: відкрийте веб-браузер та перейдіть за адресою /swagger (або /api/swagger, в залежності від налаштувань).
2) У списку доступних API знайдіть «Test» та оберіть його.
3) З доступних ендпоінтів оберіть /api/test/api-tester-timing чи /api/test/api-tester-timing-asyc та натисність «Try it out».
4) Далі у полі request введіть необхідний вам запит, наприклад: https://localhost:44361/api/list.
5) У полі numberOfRequests введіть кількість запитів, яку хочете виконати.
6) Натисніть кнопку Execute.
7) Дочекайтесь завершення запиту.
8) В результаті вам буде видано значення executionTimeMs у форматі JSON, який далі ви можете експортувати до файлу JSON та використовувати в подальших дослідженнях.

Ця інструкція надає детальні кроки для взаємодії з ендпоінтами API через Swagger-інтерфейс.

4 АВАРІЙНІ СИТУАЦІЇ

Програмний збій. Якщо у додатку виникне програмний збій під час виконання запиту, то необхідно повторити останню дію з тими же параметрами або за потреби змінити вхідні параметри запиту, якщо це не допомогло, то необхідно перезапустити додаток.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Richter, J. CLR via C# (Developer Reference) [Текст]: – Redmond, WA.: Microsoft Press, 2018. – 896 с.
2. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю.М.Івченко, В.І.Шинкаренко, В.Г.Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В.Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В.Лазаряна, 2009. – 38 с.

Текст програми
ДОДАТОК В

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

АНАЛІЗ ШВИДКОДІЙНОСТІ
ПЛАТФОРМ ASP NET ТА ASP NET
CORE

Текст програми
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1344-01 12 01-ЛЗ

Завідувач кафедри КІТ
_________Вадим ГОРЯЧКІН

Керівник розробки
_________Олександр
ІВАНОВ

Виконавець
_________Олександр
ГЕТМАНЕНКО

Нормоконтролер
_________Світлана
ВОЛКОВА

2024

ЗАТВЕРДЖЕНО
44165850.1344-01 12 01-ЛЗ

АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET ТА ASP NET CORE

Текст програми

Листів 49

2024

44165850.1344-01 12 01

АНОТАЦІЯ

Документ 44165850.1344-01 12 01 «АНАЛІЗ ШВИДКОДІЙНОСТІ ПЛАТФОРМ ASP NET TA ASP NET CORE. Текст програми».

У даному документі представлені схеми взаємодії модулів програми та текст програми. Програма написана на мові C#. Об'єм пам'яті, що займають програми комплексу складає від 16 ГБ. Конфігурація комп'ютер або ноутбук. Комплекс функціонує в середовищі всіх видів операційних систем сімейства Windows починаючи з версії Windows 7.

ЗМІСТ

1 СХЕМА ВЗАЄМОДІЇ МОДУЛІВ 5
2 ТЕКСТ ПРОГРАМИ 12
2.1 Модуль DictionaryController........ 12
2.2 Модуль ListController........ 20
2.3 Модуль QueueController 26
2.4 Модуль IDictionaryContainer........ 34
2.5 Модуль DictionaryContainer 35
2.6 Модуль ListContainer 35
2.7 Модуль IListContainer........ 35
2.8 Модуль IQueueContainer........ 35
2.9 Модуль QueueContainer 36
2.10 Модуль IDictionaryRepository 37
2.11 Модуль DictionaryRepository 37
2.12 Модуль IListRepository 39
2.13 Модуль ListRepository 39
2.14 Модуль IQueueRepository........ 40
2.15 Модуль QueueRepository 41
2.16 Модуль ICollectionRepository........ 42
2.17 Модуль ExecutionTimeMetricModel 43
2.18 Модуль GlobalEnums 43
2.19 Модуль IMetricModel........ 43
2.20 Модуль MemoryInfoMetricModel........ 44
2.21 Модуль PerformanceTestModel 44
2.22 Модуль MemoryInfoProvider........ 44
2.23 Модуль ITimer 45
2.24 Модуль PerformanceTimer........ 45
2.25 Модуль TestController........ 46
2.26 Модуль TestData........ 49

2.27 Модуль Metrics 49
2.28 Модуль TestExecutionTime 49
2.29 Модуль Memory 49

1 СХЕМА ВЗАЄМОДІЇ МОДУЛІВ

Оскільки у проектах кожен модуль програми представляє з себе клас, то для наочного відображення схеми взаємодії модулів програми використаємо діаграми класів (рис. 1.1 – 1.7). Це дозволить найбільш повно описати процес їх взаємодії між собою.

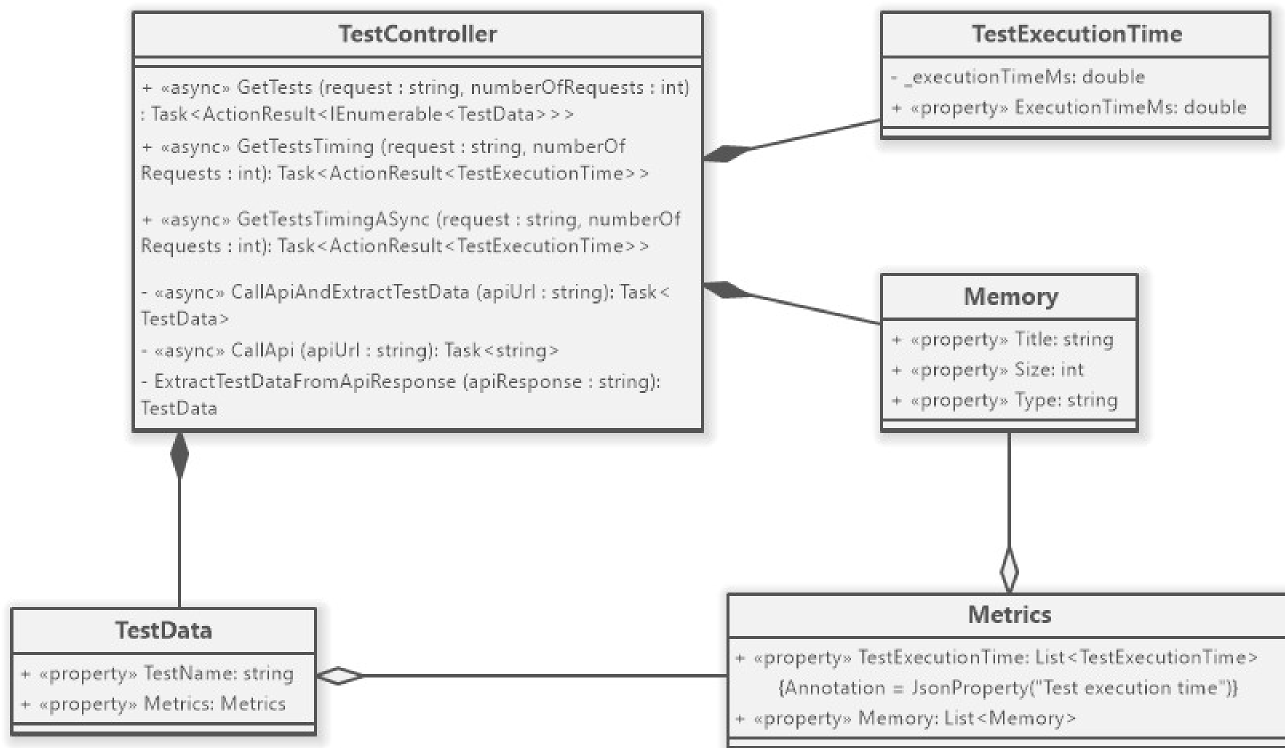

Рисунок 1.1 – Діаграма класів, проекту для виконання запитів до інших проектів

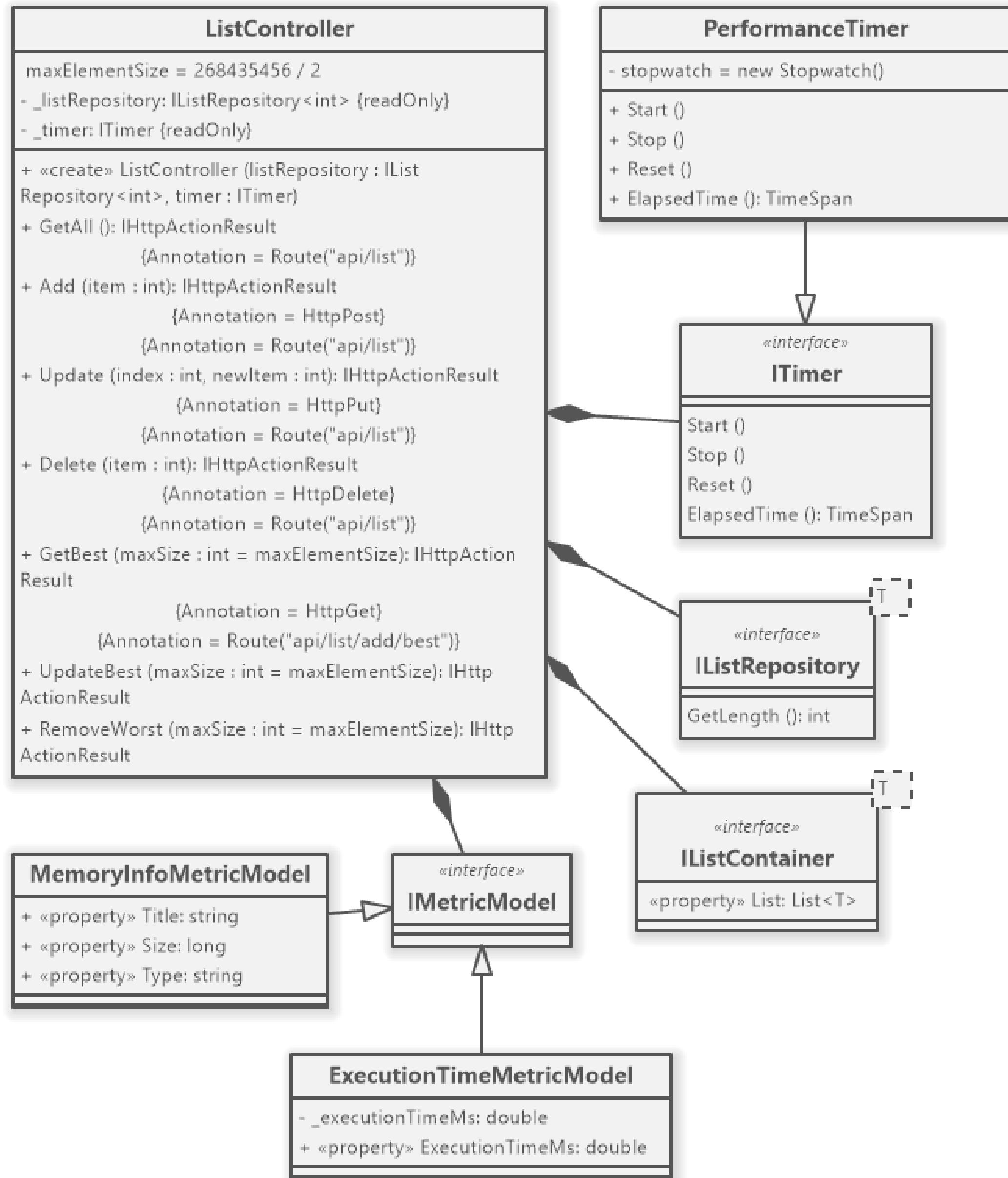

Рисунок 1.2 – Діаграма класів, взаємодія класів сервісів, моделей та даних з контролером колекції «список»

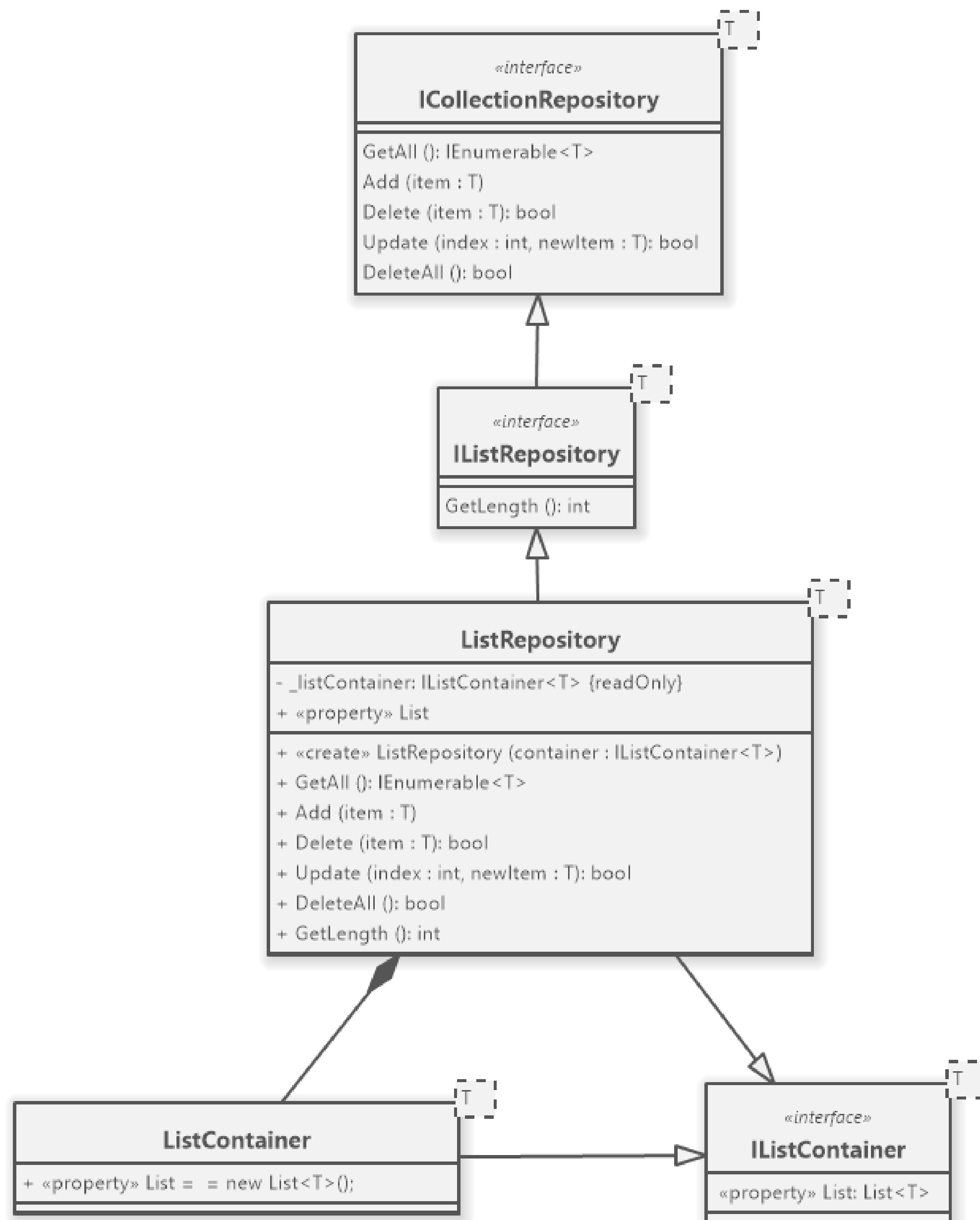

Рисунок 1.3 – Діаграма класів, яка відображає структуру класів рівня даних для колекції «список»

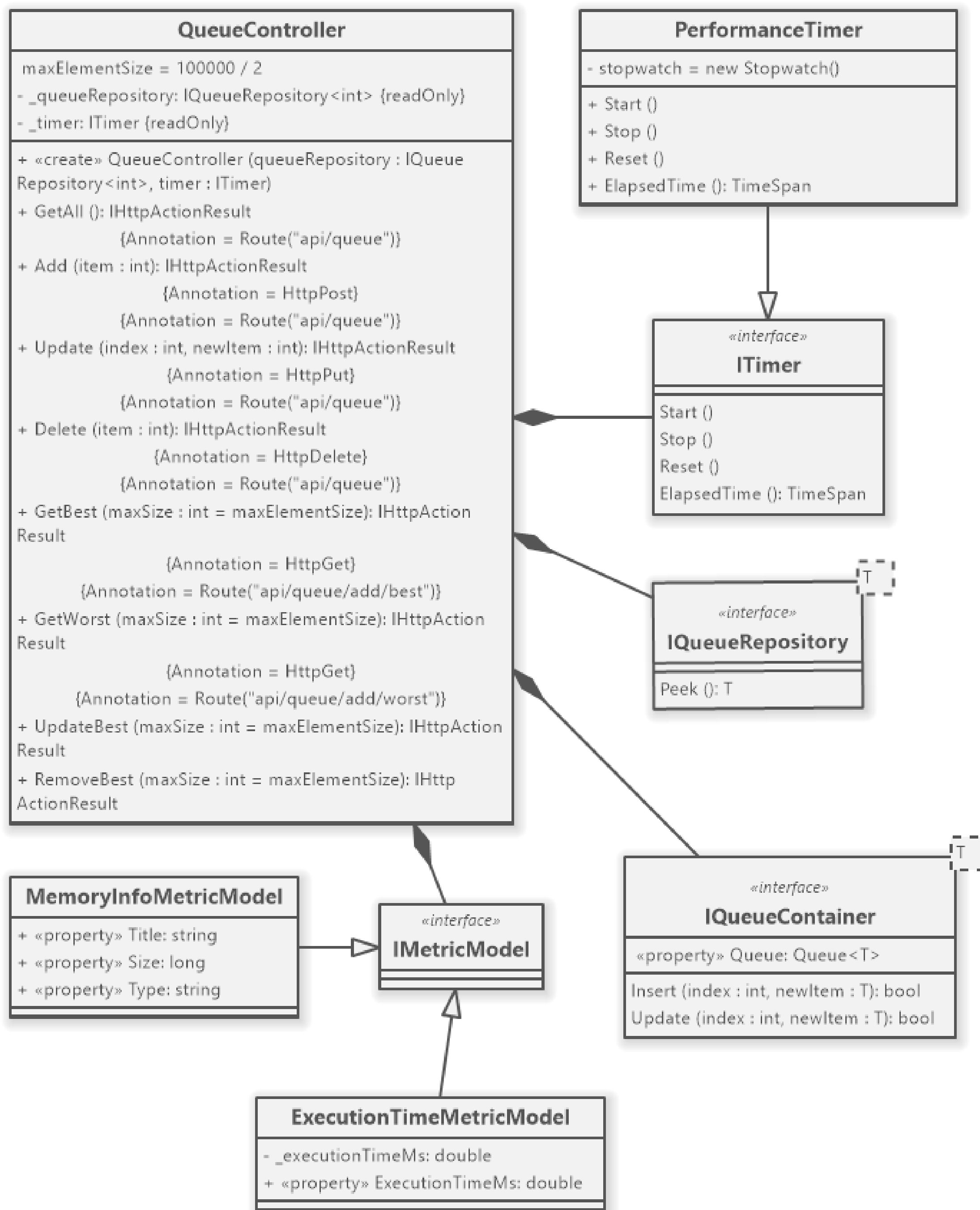

Рисунок 1.4 – Діаграма класів, взаємодія класів сервісів, моделей та даних з контролером колекції «черга»

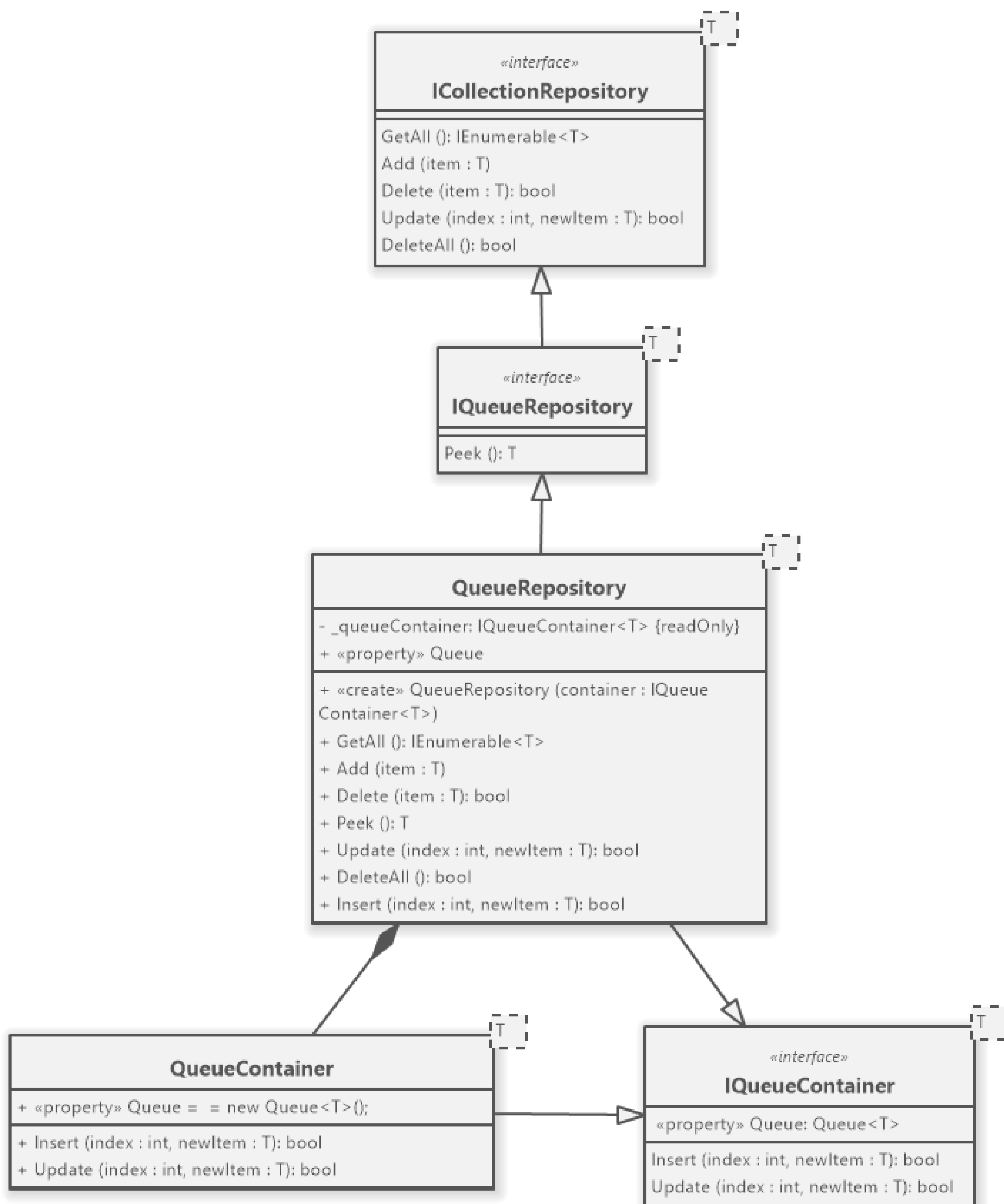

Рисунок 1.5 – Діаграма класів, яка відображає структуру класів рівня даних для колекції «черга»

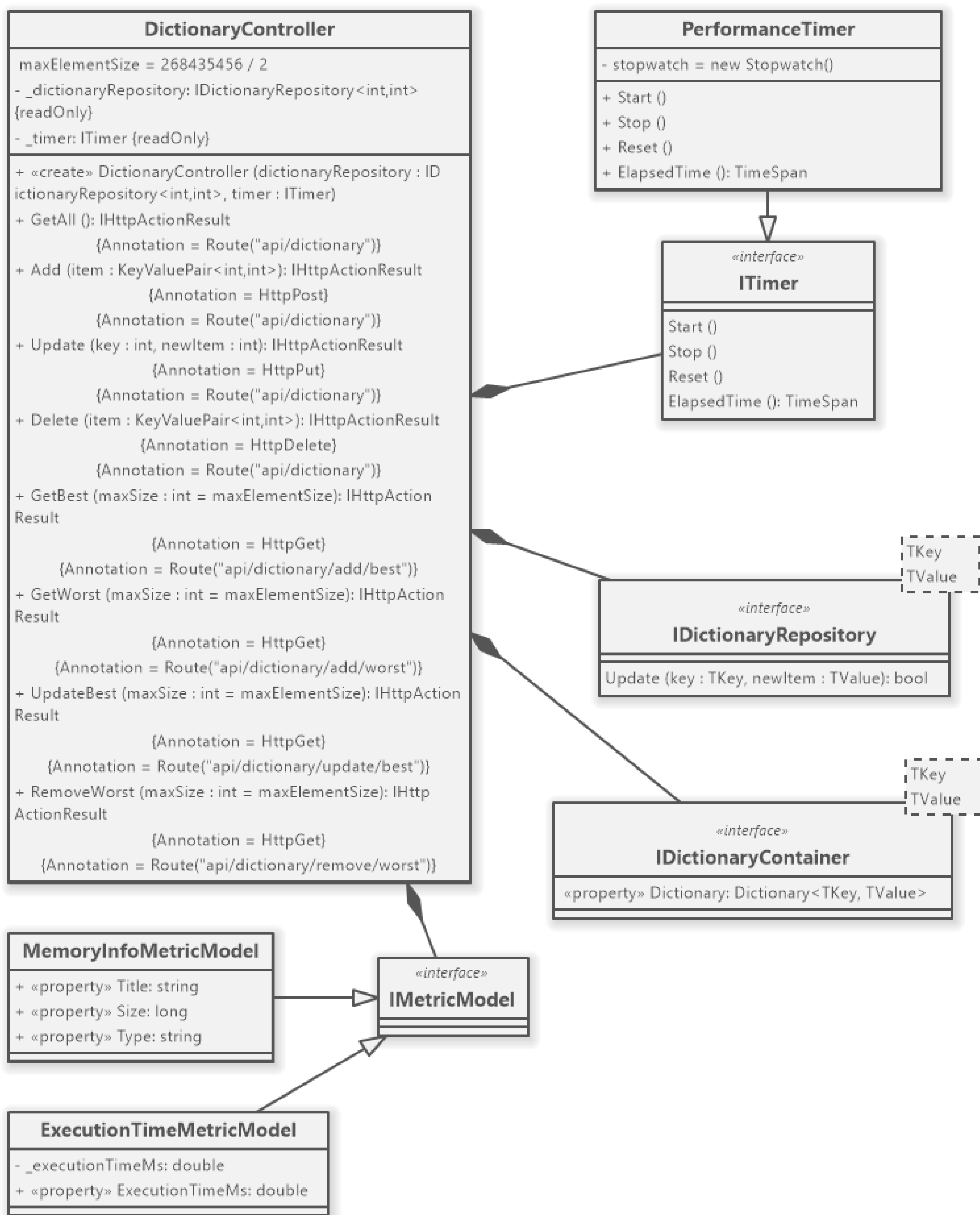

Рисунок 1.6 – Діаграма класів, взаємодія класів сервісів, моделей та даних з контролером колекції «словник»

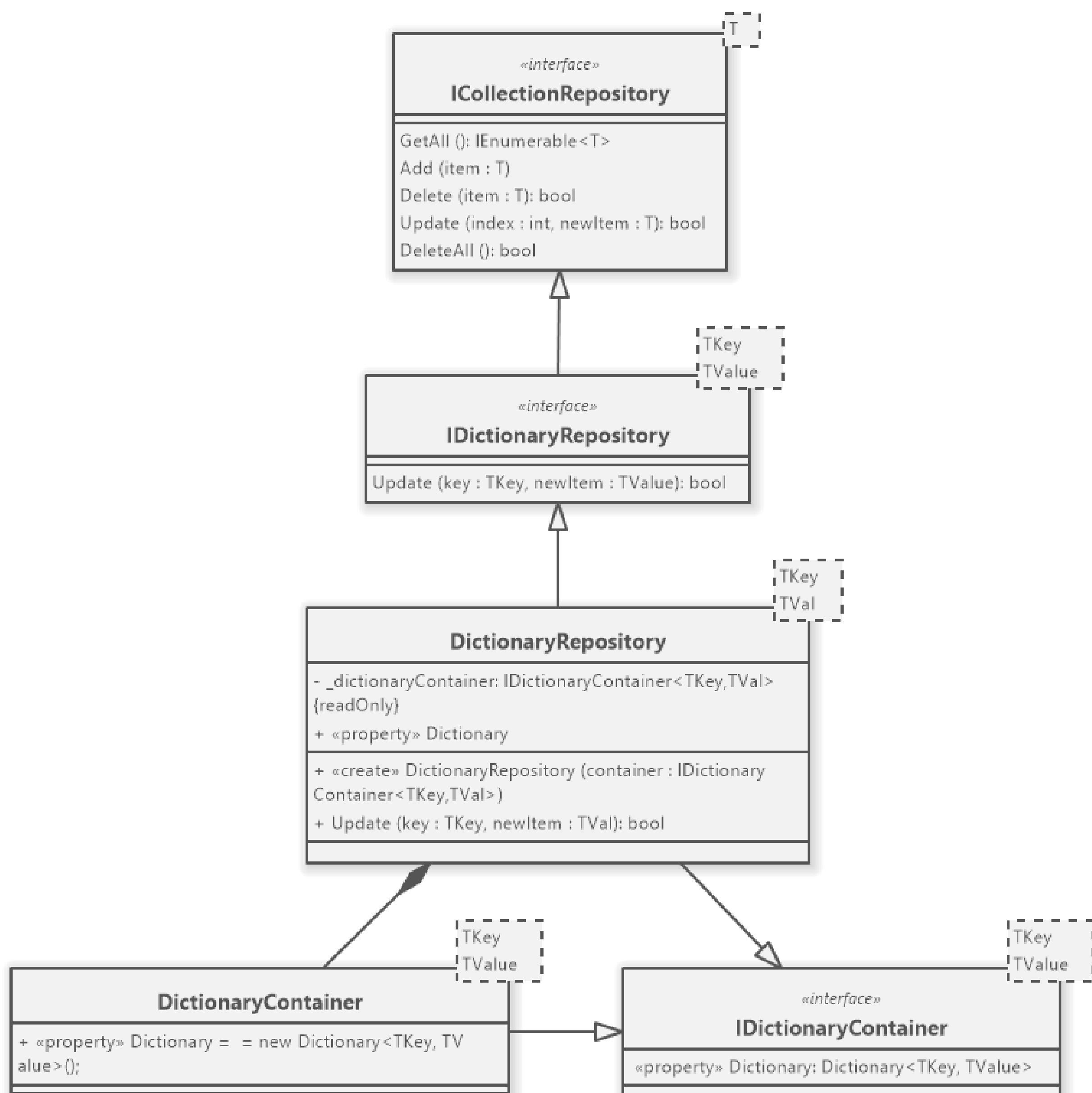

Рисунок 1.7 – Діаграма класів, яка відображає структуру класів рівня даних для колекції «словник»

2 ТЕКСТ ПРОГРАМИ

2.1 Модуль DictionaryController

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using WebAPI_ASP_Net.Repositories;
using WebAPI_ASP_Net.Repositories.Containers.Dictionary;
using WebAPI_ASP_Net.Utils;
using WebAPI_ASP_Net.Utils.MemoryUsage;
using WebAPI_ASP_Net.Utils.MetricModels;
using WebAPI_ASP_Net.Utils.Models.MetricModels;
using WebAPI_ASP_Net.Utils.Timer;

namespace WebAPI_ASP_Net.Controllers
{
    public class DictionaryController : ApiController
    {
        const int maxElementSize = 268435456 / 2;

        private readonly IDictionaryRepository<int, int>
_dictionaryRepository;

        private readonly ITimer _timer;

        public DictionaryController(IDictionaryRepository<int, int>
dictionaryRepository, ITimer timer)
        {
            _dictionaryRepository = dictionaryRepository;
            _timer = timer;
        }

        [Route("api/dictionary")]
        public IHttpActionResult GetAll()
        {
            var items = _dictionaryRepository.GetAll();
            return Ok(items);
        }

        [Route("api/dictionary")]
        [HttpPost]
        public IHttpActionResult Add(KeyValuePair<int, int> item)
        {
            _dictionaryRepository.Add(item);
            return Ok();
        }

        [Route("api/dictionary")]
        [HttpPut]
        public IHttpActionResult Update(int key, int newItem)
        {
            bool success = _dictionaryRepository.Update(key, newItem);
```

```
            if (success)
            {
                return Ok();
            }
            else
            {
                return NotFound();
            }
        }

        [Route("api/dictionary")]
        [HttpDelete]
        public IHttpActionResult Delete(KeyValuePair<int, int> item)
        {
            bool success = _dictionaryRepository.Delete(item);
            if (success)
            {
                return Ok();
            }
            else
            {
                return NotFound();
            }
        }
        [Route("api/dictionary/add/best")]
        [HttpGet]
        public IHttpActionResult GetBest(int maxSize = maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IDictionaryContainer<int, int> dictionaryContainer = new
DictionaryContainer<int, int>();

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            double executionTimeMs = 0.0;

            for (int i = 0; i < maxSize; i++)
            {
                try
                {
                    _timer.Start();
                    dictionaryContainer.Dictionary.Add(i, i);
                    _timer.Stop();

                    dictionaryContainer.Dictionary.Clear();
```

```
                    executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                    _timer.Reset();
                }
                catch
                {
                    break;
                }
            }
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Dictionary add (best)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
```

```
                    }
                }
            };

            return Ok(performanceResult);
        }

        [Route("api/dictionary/add/worst")]
        [HttpGet]
        public IHttpActionResult GetWorst(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IDictionaryContainer<int, int> dictionaryContainer = new
DictionaryContainer<int, int>();

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before Test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            _timer.Start();
            for (int i = 0; i < maxSize; i++)
            {
                try
                {
                    dictionaryContainer.Dictionary.Add(i, i);
                }
                catch
                {
                    break;
                }
            }
            _timer.Stop();
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs =
_timer.ElapsedTime().TotalMilliseconds
            };
```

```
            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Dictionary add (worst)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }

        [Route("api/dictionary/update/best")]
        [HttpGet]
        public IHttpActionResult UpdateBest(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IDictionaryContainer<int, int> dictionaryContainer = new
DictionaryContainer<int, int>();

            for (int i = 0; i < maxSize; i++)
            {
                dictionaryContainer.Dictionary.Add(i, i);
            }
```

```
            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            double executionTimeMs = 0.0;

            for (int i = 0; i < maxSize; i++)
            {
                try
                {
                    _timer.Start();
                    dictionaryContainer.Dictionary[i] = i + 1;
                    _timer.Stop();

                    executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                    _timer.Reset();
                }
                catch
                {
                    break;
                }
            }
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Dictionary update (best)",
                Metrics = new Dictionary<string,
```

```
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }

        [Route("api/dictionary/remove/worst")]
        [HttpGet]
        public IHttpActionResult RemoveWorst(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IDictionaryContainer<int, int> dictionaryContainer = new
DictionaryContainer<int, int>();

            for (int i = 0; i < maxSize; i++)
            {
                dictionaryContainer.Dictionary.Add(i, i);
            }

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            double executionTimeMs = 0.0;

            for (int i = maxSize - 1; i >= 0; i--)
            {
                try
```

```
                {
                    _timer.Start();
                    dictionaryContainer.Dictionary.Remove(i);
                    _timer.Stop();

                    executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                    _timer.Reset();
                }
                catch
                {
                    break;
                }
            }
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Dictionary remove (worst)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
```

```
                        GCMemorySize,
                        processMemorySizeBeforeTest,
                        processMemorySizeAfterTest
                    }
                }
            }
        };

        return Ok(performanceResult);
    }
  }
}
```

2.2 Модуль ListController

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using WebAPI_ASP_Net.Repositories.Containers.List;
using WebAPI_ASP_Net.Repositories.List;
using WebAPI_ASP_Net.Utils;
using WebAPI_ASP_Net.Utils.MemoryUsage;
using WebAPI_ASP_Net.Utils.MetricModels;
using WebAPI_ASP_Net.Utils.Models.MetricModels;
using WebAPI_ASP_Net.Utils.Timer;

namespace WebAPI_ASP_Net.Controllers
{
    public class ListController : ApiController
    {
        const int maxElementSize = 268435456 / 2;
        private readonly IListRepository<int> _listRepository;
        private readonly ITimer _timer;

        public ListController(IListRepository<int> listRepository,
ITimer timer)
        {
            _listRepository = listRepository;
            _timer = timer;
        }

        [Route("api/list")]
        public IHttpActionResult GetAll()
        {
            var items = _listRepository.GetAll();
            return Ok(items);
        }

        [Route("api/list")]
        [HttpPost]
        public IHttpActionResult Add(int item)
        {
```

```
            _listRepository.Add(item);
            return Ok(item);
        }

        [Route("api/list")]
        [HttpPut]
        public IHttpActionResult Update(int index, int newItem)
        {
            bool success = _listRepository.Update(index, newItem);
            if (success)
            {
                return Ok();
            }
            else
            {
                return NotFound();
            }
        }

        [Route("api/list")]
        [HttpDelete]
        public IHttpActionResult Delete(int item)
        {
            bool success = _listRepository.Delete(item);
            if (success)
            {
                return Ok();
            }
            else
            {
                return NotFound();
            }
        }
        [Route("api/list/add/best")]
        [HttpGet]
        public IHttpActionResult GetBest(int maxSize = maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IListContainer<int> listContainer = new
ListContainer<int>();

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before Test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            _timer.Start();
            for (int i = 0; i < maxSize; i++)
```

```
            {
                try
                {
                    listContainer.List.Add(i);
                }
                catch
                {
                    break;
                }
            }
            _timer.Stop();
            GC.Collect();


            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs =
_timer.ElapsedTime().TotalMilliseconds
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "List add (best)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel> {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
```

```
                        processMemorySizeAfterTest
                    }
                }
            }
        };

        return Ok(performanceResult);
    }

    [Route("api/list/update/best")]
    [HttpGet]
    public IHttpActionResult UpdateBest(int maxSize =
maxElementSize)
    {
        GC.Collect();
        GC.WaitForPendingFinalizers();

        IListContainer<int> listContainer = new
ListContainer<int>();

        for (int i = 0; i < maxSize; i++)
        {
            listContainer.List.Add(i);
        }

        var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
        {
            Title = "Process before test",
            Size = MemoryInfoProvider.GetProcessMemorySize(),
            Type = EMemorySizeType.Byte.ToString(),
        };

        double executionTimeMs = 0.0;

        for (int i = 0; i < maxSize; i++)
        {
            try
            {
                _timer.Start();
                listContainer.List[0] = i + 1; // Оновлення
елемента
                _timer.Stop();

                executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                _timer.Reset();
            }
            catch
            {
                break;
            }
        }
```

```
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "List update (best)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }

        [Route("api/list/remove/worst")]
        [HttpGet]
```

```
        public IHttpActionResult RemoveWorst(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IListContainer<int> listContainer = new
ListContainer<int>();

            for (int i = 0; i < maxSize; i++)
            {
                listContainer.List.Add(i);
            }

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            double executionTimeMs = 0.0;

            for (int i = maxSize - 2; i >= 0; i--)
            {
                try
                {
                    _timer.Start();
                    listContainer.List.RemoveAt(i); // Видалення
елемента
                    _timer.Stop();

                    executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                    _timer.Reset();
                }
                catch
                {
                    break;
                }
            }
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
```

```
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "List remove (worst)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }

    }
}
```

2.3 Модуль QueueController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;
using WebAPI_ASP_Net.Repositories.Containers.Queue;
using WebAPI_ASP_Net.Repositories.Queue;
using WebAPI_ASP_Net.Utils;
using WebAPI_ASP_Net.Utils.MemoryUsage;
```

```
using WebAPI_ASP_Net.Utils.MetricModels;
using WebAPI_ASP_Net.Utils.Models.MetricModels;
using WebAPI_ASP_Net.Utils.Timer;

namespace WebAPI_ASP_Net.Controllers
{
    public class QueueController : ApiController
    {
        const int maxElementSize = 100000 / 2;
        private readonly IQueueRepository<int> _queueRepository;
        private readonly ITimer _timer;

        public QueueController(IQueueRepository<int> queueRepository,
ITimer timer)
        {
            _queueRepository = queueRepository;
            _timer = timer;
        }

        [Route("api/queue")]
        public IHttpActionResult GetAll()
        {
            var items = _queueRepository.GetAll();
            return Ok(items);
        }

        [Route("api/queue")]
        [HttpPost]
        public IHttpActionResult Add(int item)
        {
            _queueRepository.Add(item);
            return Ok();
        }

        [Route("api/queue")]
        [HttpPut]
        public IHttpActionResult Update(int index, int newItem)
        {
            bool success = _queueRepository.Update(index, newItem);
            if (success)
            {
                return Ok();
            }
            else
            {
                return NotFound();
            }
        }

        [Route("api/queue")]
        [HttpDelete]
        public IHttpActionResult Delete(int item)
        {
```

```
            bool success = _queueRepository.Delete(item);
            if (success)
            {
                return Ok();
            }
            else
            {
                return NotFound();
            }
        }

        [Route("api/queue/add/best")]
        [HttpGet]
        public IHttpActionResult GetBest(int maxSize = maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IQueueContainer<int> dataContainer = new
QueueContainer<int>();

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before Test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            _timer.Start();
            for (int i = 0; i < maxSize; i++)
            {
                try
                {
                    dataContainer.Queue.Enqueue(i);
                }
                catch
                {
                    break;
                }
            }
            _timer.Stop();
            GC.Collect();


            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
```

```
            {
                ExecutionTimeMs =
_timer.ElapsedTime().TotalMilliseconds
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Queue add (best)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel> {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }
        [Route("api/queue/add/worst")]
        [HttpGet]
        public IHttpActionResult GetWorst(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IQueueContainer<int> dataContainer = new
QueueContainer<int>();

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
```

```
                Title = "Process before Test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            for (int i = 0; i < 2; i++)
            {
                dataContainer.Queue.Enqueue(i);
            }

            _timer.Start();
            for (int i = 2; i < maxSize; i++)
            {
                int queueSize = dataContainer.Queue.Count();
                dataContainer.Insert(1, i);
            }

            _timer.Stop();
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs =
_timer.ElapsedTime().TotalMilliseconds
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Queue add (worst)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel> {
                            executionTime
                        }
                    },
```

```
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }

        [Route("api/queue/update/best")]
        [HttpGet]
        public IHttpActionResult UpdateBest(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IQueueContainer<int> dataContainer = new
QueueContainer<int>();

            for (int i = 0; i < maxSize; i++)
            {
                dataContainer.Queue.Enqueue(i);
            }

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            double executionTimeMs = 0.0;

            for (int i = 0; i < maxSize; i++)
            {
                try
                {
                    _timer.Start();
                    dataContainer.Update(i, i + 1); // Оновлення
елемента
                    _timer.Stop();

                    executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                    _timer.Reset();
```

```
                }
                catch
                {
                    break;
                }
            }
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Queue update (best)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };
```

```
            return Ok(performanceResult);
        }

        [Route("api/queue/remove/best")]
        [HttpGet]
        public IHttpActionResult RemoveBest(int maxSize =
maxElementSize)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();

            IQueueContainer<int> dataContainer = new
QueueContainer<int>();

            var processMemorySizeBeforeTest = new
MemoryInfoMetricModel
            {
                Title = "Process before test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
                Type = EMemorySizeType.Byte.ToString(),
            };

            for (int i = 0; i < maxSize; i++)
            {
                dataContainer.Queue.Enqueue(i);
            }

            double executionTimeMs = 0.0;

            for (int i = 0; i < maxSize; i++)
            {
                try
                {
                    _timer.Start();
                    dataContainer.Queue.Dequeue(); // Видалення
елемента
                    _timer.Stop();

                    executionTimeMs +=
_timer.ElapsedTime().TotalMilliseconds;
                    _timer.Reset();
                }
                catch
                {
                    break;
                }
            }
            GC.Collect();

            var processMemorySizeAfterTest = new MemoryInfoMetricModel
            {
                Title = "Process after test",
                Size = MemoryInfoProvider.GetProcessMemorySize(),
```

```
                Type = EMemorySizeType.Byte.ToString(),
            };

            var executionTime = new ExecutionTimeMetricModel
            {
                ExecutionTimeMs = executionTimeMs
            };

            var GCMemorySize = new MemoryInfoMetricModel
            {
                Title = "GC",
                Size = MemoryInfoProvider.GetGCHeapSize(true),
                Type = EMemorySizeType.Byte.ToString(),
            };

            PerformanceTestModel performanceResult = new
PerformanceTestModel
            {
                TestName = "Queue remove (Best)",
                Metrics = new Dictionary<string,
IEnumerable<IMetricModel>>
                {
                    {
                        "Test execution time",
                        new List<IMetricModel>
                        {
                            executionTime
                        }
                    },
                    {
                        "Memory",
                        new List<IMetricModel>
                        {
                            GCMemorySize,
                            processMemorySizeBeforeTest,
                            processMemorySizeAfterTest
                        }
                    }
                }
            };

            return Ok(performanceResult);
        }

    }
}
```

2.4 Модуль IDictionaryContainer

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories.Containers.Dictionary
```

```
{
    public interface IDictionaryContainer<TKey, TValue>
    {
        Dictionary<TKey, TValue> Dictionary { get; }
    }
}
```

2.5 Модуль DictionaryContainer

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories.Containers.Dictionary
{
    public class DictionaryContainer<TKey, TValue> :
IDictionaryContainer<TKey, TValue>
    {
        public Dictionary<TKey, TValue> Dictionary { get; } = new
Dictionary<TKey, TValue>();
    }
}
```

2.6 Модуль ListContainer

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories.Containers.List
{
    public class ListContainer<T> : IListContainer<T>
    {
        public List<T> List { get; } = new List<T>();
    }
}
```

2.7 Модуль IListContainer

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories.Containers.List
{
    public interface IListContainer<T>
    {
        List<T> List { get; }
    }
}
```

2.8 Модуль IQueueContainer

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories.Containers.Queue
{
    public interface IQueueContainer<T>
    {
        Queue<T> Queue { get; }

        bool Insert(int index, T newItem);
        bool Update(int index, T newItem);
    }
}
```

2.9 Модуль QueueContainer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace WebAPI_ASP_Net.Repositories.Containers.Queue
{
    public class QueueContainer<T> : IQueueContainer<T>
    {
        public Queue<T> Queue { get; } = new Queue<T>();

        public bool Insert(int index, T newItem)
        {
            var tempQueue = new Queue<T>();
            var count = Queue.Count;

            if (index >= 0 && index <= count)
            {
                // Додайте елементи з початку черги до тимчасової
черги
                for (int i = 0; i < index; i++)
                {
                    tempQueue.Enqueue(Queue.Dequeue());
                }

                // Додайте новий елемент до тимчасової черги
                tempQueue.Enqueue(newItem);

                // Додайте інші елементи з початкової черги назад до
тимчасової черги
                while (Queue.Count > 0)
                {
                    tempQueue.Enqueue(Queue.Dequeue());
                }
```

```
                // Очистіть початкову чергу і скопіюйте елементи назад
з тимчасової черги
                Queue.Clear();
                foreach (var item in tempQueue)
                {
                    Queue.Enqueue(item);
                }

                return true;
            }

            return false;
        }

        public bool Update(int index, T newItem)
        {
            var tempList = Queue.ToList();
            if (index > -1 && index < tempList.Count)
            {
                tempList[index] = newItem;
                Queue.Clear();
                foreach (var i in tempList)
                {
                    Queue.Enqueue(i);
                }
                return true;
            }
            return false;
        }

    }
}
```

2.10 Модуль IDictionaryRepository

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories
{
    public interface IDictionaryRepository<TKey, TValue> :
ICollectionRepository<KeyValuePair<TKey, TValue>>
    {
        bool Update(TKey key, TValue newItem);
    }
}
```

2.11 Модуль DictionaryRepository

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

```
using WebAPI_ASP_Net.Repositories.Containers.Dictionary;

namespace WebAPI_ASP_Net.Repositories
{
    public class DictionaryRepository<TKey, TVal> :
IDictionaryRepository<TKey, TVal>, IDictionaryContainer<TKey, TVal>
    {
        private readonly IDictionaryContainer<TKey, TVal>
_dictionaryContainer;

        public Dictionary<TKey, TVal> Dictionary =>
_dictionaryContainer.Dictionary;

        Dictionary<TKey, TVal> IDictionaryContainer<TKey,
TVal>.Dictionary => throw new System.NotImplementedException();

        public DictionaryRepository(IDictionaryContainer<TKey, TVal>
container)
        {
            _dictionaryContainer = container;
        }

        public bool Update(TKey key, TVal newItem)
        {
            if (_dictionaryContainer.Dictionary.TryGetValue(key, out
TVal value))
            {
                _dictionaryContainer.Dictionary[key] = newItem;
                return true;
            }
            return false;
        }

        IEnumerable<KeyValuePair<TKey, TVal>>
ICollectionRepository<KeyValuePair<TKey, TVal>>.GetAll()
        {
            return _dictionaryContainer.Dictionary;
        }

        public void Add(KeyValuePair<TKey, TVal> item)
        {
            _dictionaryContainer.Dictionary.Add(item.Key, item.Value);
        }

        public bool Delete(KeyValuePair<TKey, TVal> item)
        {
            return _dictionaryContainer.Dictionary.Remove(item.Key);
        }

        public bool Update(int index, KeyValuePair<TKey, TVal>
newItem)
        {
```

```
            if (index >= 0 && index <
_dictionaryContainer.Dictionary.Count)
            {
                KeyValuePair<TKey, TVal> item =
_dictionaryContainer.Dictionary.ElementAt(index);
                _dictionaryContainer.Dictionary[item.Key] =
newItem.Value;
                return true;
            }
            return false;
        }
        public bool DeleteAll()
        {
            try
            {
                _dictionaryContainer.Dictionary.Clear();
            }
            catch
            {
                return false;
            }
            return true;
        }
    }
}
```

2.12 Модуль IListRepository

```
namespace WebAPI_ASP_Net.Repositories.List
{
    public interface IListRepository<T> : ICollectionRepository<T>
    {
        int GetLength();
    }
}
```

2.13 Модуль ListRepository

```
using System.Collections.Generic;
using WebAPI_ASP_Net.Repositories.Containers.List;
using WebAPI_ASP_Net.Repositories.List;

namespace WebAPI_ASP_Net.Repositories
{
    public class ListRepository<T> : IListRepository<T>,
IListContainer<T>
    {
        private readonly IListContainer<T> _listContainer;

        public List<T> List => _listContainer.List;
```

```
        public ListRepository(IListContainer<T> container)
        {
            _listContainer = container;
        }

        public IEnumerable<T> GetAll()
        {
            return _listContainer.List;
        }

        public void Add(T item)
        {
            _listContainer.List.Add(item);
        }

        public bool Delete(T item)
        {
            return _listContainer.List.Remove(item);
        }

        public bool Update(int index, T newItem)
        {
            if (index > -1 && index < _listContainer.List.Count)
            {
                _listContainer.List[index] = newItem;
                return true;
            }
            return false;
        }

        public bool DeleteAll()
        {
            try {
                _listContainer.List.Clear();
            } catch {
                return false;
            }
            return true;
        }

        public int GetLength()
        {
            return _listContainer.List.Count;
        }
    }
}
```

2.14 Модуль IQueueRepository

```
using System;
```

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

namespace WebAPI_ASP_Net.Repositories.Queue
{
    public interface IQueueRepository<T> : ICollectionRepository<T>
    {
        T Peek();
    }
}
```

2.15 Модуль QueueRepository

```
using System.Collections.Generic;
using System.Linq;
using WebAPI_ASP_Net.Repositories.Containers.Queue;
using WebAPI_ASP_Net.Repositories.Queue;

namespace WebAPI_ASP_Net.Repositories
{
    public class QueueRepository<T>: IQueueRepository<T>,
IQueueContainer<T>
    {
        private readonly IQueueContainer<T> _queueContainer;
        public Queue<T> Queue => _queueContainer.Queue;

        public QueueRepository(IQueueContainer<T> container)
        {
            _queueContainer = container;
        }
        public IEnumerable<T> GetAll()
        {
            return _queueContainer.Queue;
        }

        public void Add(T item)
        {
            _queueContainer.Queue.Enqueue(item);
        }

        public bool Delete(T item)
        {
            var tempList = _queueContainer.Queue.ToList();
            bool success = tempList.Remove(item);
            if (success)
            {
                _queueContainer.Queue.Clear();
                foreach (var i in tempList)
                {
                    _queueContainer.Queue.Enqueue(i);
```

```
                }
            }
            return success;
        }

        public T Peek()
        {
            return _queueContainer.Queue.Peek();
        }

        public bool Update(int index, T newItem)
        {
            var tempList = _queueContainer.Queue.ToList();
            if (index > -1 && index < tempList.Count)
            {
                tempList[index] = newItem;
                _queueContainer.Queue.Clear();
                foreach (var i in tempList)
                {
                    _queueContainer.Queue.Enqueue(i);
                }
                return true;
            }
            return false;
        }

        public bool DeleteAll()
        {
            try
            {
                _queueContainer.Queue.Clear();
            }
            catch
            {
                return false;
            }
            return true;
        }

        public bool Insert(int index, T newItem)
        {
            return _queueContainer.Insert(index, newItem);
        }
    }
}
```

2.16 Модуль ICollectionRepository

```
using System.Collections.Generic;

namespace WebAPI_ASP_Net.Repositories
```

```
{
    public interface ICollectionRepository<T>
    {
        IEnumerable<T> GetAll();
        void Add(T item);
        bool Delete(T item);
        bool Update(int index, T newItem);
        bool DeleteAll();
    }
}
```

2.17 Модуль ExecutionTimeMetricModel

```
using System;

namespace WebAPI_ASP_Net.Utils.MetricModels
{
    public class ExecutionTimeMetricModel : IMetricModel
    {
        private double _executionTimeMs;

        public double ExecutionTimeMs
        {
            get => _executionTimeMs;
            set => _executionTimeMs = Math.Round(value, 6);
        }
    }
}
```

2.18 Модуль GlobalEnums

```
namespace WebAPI_ASP_Net.Utils.Models.MetricModels
{
    enum EMemorySizeType
    {
        Bit,
        Byte,
        Kilobyte,
        Megabyte,
        Gigabyte,
    }

    enum ECountElementType
    {
        Item,
        Items,
    }
}
```

2.19 Модуль IMetricModel

```
namespace WebAPI_ASP_Net.Utils.MetricModels
{
    public interface IMetricModel
    {
    }
}
```

2.20 Модуль MemoryInfoMetricModel

```
using WebAPI_ASP_Net.Utils.MetricModels;

namespace WebAPI_ASP_Net.Utils.MemoryUsage
{
    public class MemoryInfoMetricModel : IMetricModel
    {
        public string Title { get; set; }
        public long Size { get; set; }
        public string Type { get; set; }
    }
}
```

2.21 Модуль PerformanceTestModel

```
using System.Collections.Generic;
using WebAPI_ASP_Net.Utils.MetricModels;

namespace WebAPI_ASP_Net.Utils
{
    public class PerformanceTestModel
    {
        public string TestName { get; set; }
        public IDictionary<string, IEnumerable<IMetricModel>> Metrics
{ get; set; }
    }
}
```

2.22 Модуль MemoryInfoProvider

```
using System;
using System.Diagnostics;

namespace WebAPI_ASP_Net.Utils
{
    public static class MemoryInfoProvider
    {
        public static long GetProcessMemorySize()
        {
            Process process = Process.GetCurrentProcess();
            return process.WorkingSet64;
        }
```

```
        public static long GetGCHeapSize(bool forceFullCollection)
        {
            return GC.GetTotalMemory(forceFullCollection);
        }
        public static long GetMemorySizeWithPerfomanceCounter()
        {
            PerformanceCounter memoryCounter = new
PerformanceCounter("Process", "Working Set - Private",
Process.GetCurrentProcess().ProcessName);
            return memoryCounter.RawValue;
        }
    }
}
```

2.23 Модуль ITimer

```
using System;

namespace WebAPI_ASP_Net.Utils.Timer
{
    public interface ITimer
    {
        void Start();

        void Stop();

        void Reset();

        TimeSpan ElapsedTime();
    }
}
```

2.24 Модуль PerformanceTimer

```
using System;
using System.Diagnostics;

namespace WebAPI_ASP_Net.Utils.Timer
{
    public class PerformanceTimer : ITimer
    {
        private Stopwatch stopwatch = new Stopwatch();

        public void Start()
        {
            stopwatch.Start();
        }

        public void Stop()
```

```
        {
            stopwatch.Stop();
        }

        public void Reset()
        {
            stopwatch.Reset();
        }

        public TimeSpan ElapsedTime()
        {
            return stopwatch.Elapsed;
        }
    }

}
```

2.25 Модуль TestController

```
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using System.Diagnostics;
using System.Text;

[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase
{
    [HttpGet("api-tester")]
    public async Task<ActionResult<IEnumerable<TestData>>>
GetTests([FromQuery] string request, [FromQuery] int numberOfRequests)
    {
        // Створення списку для результатів тестів
        List<Task<TestData>> tasks = new List<Task<TestData>>();

        // Виклик API та отримання даних для кількох запитів
        for (int i = 0; i < numberOfRequests; i++)
        {
            string apiUrl = request;
            tasks.Add(CallApiAndExtractTestData(apiUrl));
        }

        // Очікування завершення всіх асинхронних запитів
        TestData[] testResults = await Task.WhenAll(tasks);

        // Повернення списку тестів
        return testResults.ToList();
    }

    [HttpGet("api-tester-timing")]
```

```
    public async Task<ActionResult<TestExecutionTime>>
GetTestsTiming([FromQuery] string request, [FromQuery] int
numberOfRequests)
    {

        var stopwatch = Stopwatch.StartNew();

        using (var httpClient = new HttpClient())
        {
            for (int i = 0; i < numberOfRequests; i++)
            {
                HttpResponseMessage response = await
httpClient.GetAsync(request);
            }
        }

        stopwatch.Stop();

        // Повернення загального часу виконання всіх запитів
        var responseModel = new TestExecutionTime
        {
            ExecutionTimeMs = stopwatch.Elapsed.TotalMilliseconds
        };

        return responseModel;
    }

    [HttpGet("api-tester-timing-async")]
    public async Task<ActionResult<TestExecutionTime>>
GetTestsTimingASync([FromQuery] string request, [FromQuery] int
numberOfRequests)
    {
        Stopwatch stopwatch = new Stopwatch();

        using (HttpClient client = new HttpClient())
        {
            stopwatch.Start();

            // Створюємо масив завдань (tasks)
            Task<HttpResponseMessage>[] tasks = new
Task<HttpResponseMessage>[numberOfRequests];

            // Запускаємо асинхронні запити
            for (int i = 0; i < numberOfRequests; i++)
            {
                tasks[i] = client.GetAsync(request);
            }

            // Очікуємо завершення всіх асинхронних запитів
            await Task.WhenAll(tasks);

            stopwatch.Stop();
        }
```

```
        // Повернення загального часу виконання всіх запитів
        var responseModel = new TestExecutionTime
        {
            ExecutionTimeMs = stopwatch.Elapsed.TotalMilliseconds
        };

        return responseModel;
    }

    // Метод для виклику API та отримання даних з тестами
    private async Task<TestData> CallApiAndExtractTestData(string
apiUrl)
    {
        string apiResponse = await CallApi(apiUrl);

        // Обробка отриманих даних та витягнення тестів
        return ExtractTestDataFromApiResponse(apiResponse);
    }

    // Метод для виклику API
    private async Task<string> CallApi(string apiUrl)
        {
        using (HttpClient client = new HttpClient())
        {
            HttpResponseMessage response = await
client.GetAsync(apiUrl);

            if (response.IsSuccessStatusCode)
            {
                return await response.Content.ReadAsStringAsync();
            }

            return null;
        }
    }


    // Метод для обробки отриманих даних та витягнення тестів
    private TestData ExtractTestDataFromApiResponse(string
apiResponse)
    {
        if (!string.IsNullOrEmpty(apiResponse))
        {
            // Використовуйте Newtonsoft.Json для десеріалізації JSON
            return
JsonConvert.DeserializeObject<TestData>(apiResponse);
        }

        return null;
    }
}
```

2.26 Модуль TestData

```
public class TestData
{
    public string TestName { get; set; }
    public Metrics Metrics { get; set; }
}
```

2.27 Модуль Metrics

```
public class Metrics
{
    [JsonProperty("Test execution time")]
    public List<TestExecutionTime> TestExecutionTime { get; set; }
    public List<Memory> Memory { get; set; }
}
```

2.28 Модуль TestExecutionTime

```
public class TestExecutionTime
{
    private double _executionTimeMs;

    public double ExecutionTimeMs
    {
        get => _executionTimeMs;
        set => _executionTimeMs = Math.Round(value, 6);
    }
}
```

2.29 Модуль Memory

```
public class Memory
{
    public string Title { get; set; }
    public int Size { get; set; }
    public string Type { get; set; }
}
```