Міністерство освіти і науки, молоді та спорту України
Львівський національний університет імені Івана Франка
Факультет електроніки та комп'ютерних технологій
Кафедра__радіоелектронних та комп'ютерних систем__

Допустити до захисту
Завідувач кафедри
_____ проф. Оленич І.Б._
(підпис) (прізвище та ініціали)

«__»_____ 20__ р.

Кваліфікаційна робота
Бакалавр
(освітній ступінь)

Розроблення програмних засобів для аналізу процесів під час виконання програмного коду

Виконав:
Студент групи _ФЕП-41____
Спеціальності:
121 Інженерія програмного забезпечення
_____ _Сьомкін І.В.__
(підпис) (прізвище та ініціали)
Науковий керівник:
_____ ____Футей О.В.__
(підпис) (прізвище та ініціали)
«__»_____ 20__ р.
Рецизент:
_____ доц. Слободзян Д.П._
(підпис) (прізвище та ініціали)

Львів 2023

ЗМІСТ

АНОТАЦІЯ

Дана дипломна робота присвячена розробленню програмних засобів для аналізу процесів під час виконання програмного коду. Головною метою цього дослідження є створення ефективних інструментів, які дозволять аналізувати внутрішній стан програми, відстежувати її виконання та виявляти можливі проблеми. Для досягнення цієї мети будуть використані мова програмування C# та сучасні технології розробки програмного забезпечення

ABSTRACT

This graduate work is devoted to the development of software tools for analyzing processes during the execution of software code. The main goal of this research is to create effective tools that will allow analyzing the internal state of the program, monitoring its execution and identifying possible problems. To achieve this goal, the C# programming language and modern software development technologies will be used

Перелік умовних позначень та скорочень

API - Інтерфейс програмування додатків (Application Programming Interface).

IDE - Інтегроване середовище розробки (Integrated Development Environment).

JIT - Just-In-Time компіляція. CPU - Центральний процесор (Central Processing Unit).

RAM - Випадково доступна пам'ять (Random Access Memory).

GUI - Графічний інтерфейс користувача (Graphical User Interface).

ВСТУП

Розробка програмного забезпечення стає все більш складною та вимагає від розробників знань та інструментів, що дозволяють аналізувати процеси виконання програмного коду. З метою поліпшення якості, ефективності та надійності програм, необхідно мати засоби, які дозволяють відстежувати та аналізувати процеси виконання коду.

У даній дипломній роботі пропонується розроблення програмних засобів, які допоможуть розробникам здійснювати аналіз програм та виявляти можливі проблеми під час їх виконання. Актуальність проблеми: Зростаюча складність програмних систем та великий обсяг програмного коду вимагають від розробників ефективних засобів для аналізу процесів під час виконання коду. Це стає особливо важливим у випадку великих проектів, де помилки та проблеми в процесі виконання можуть мати серйозні наслідки, такі як падіння програми або витрати ресурсів. Розроблення програмних інструментів для аналізу процесів виконання є актуальною проблемою, яка може сприяти покращенню якості та надійності програмного забезпечення.

Об'єктом дослідження: Об'єктом дослідження є процеси під час виконання програмного коду та методи аналізу цих процесів. Дослідження орієнтується на розробку програмних засобів, які дозволять відстежувати та аналізувати внутрішній стан програми під час виконання, що допоможе виявити можливі проблеми та вдосконалити розробку програмного забезпечення.

Метою дослідження: Головною метою даного дослідження є розроблення програмних засобів, які дозволять аналізувати процеси виконання програмного коду. Ці засоби повинні надавати можливість відстежувати виконання коду, аналізувати його структуру та виявляти можливі проблеми. Метою є створення ефективних інструментів, які сприятимуть поліпшенню процесу розробки та підтримки програмного забезпечення. Для досягнення поставленої мети ставляться наступні завдання: Вивчення процесу виконання програмного коду та аналізу програмного коду. Розроблення архітектури програмних засобів для аналізу процесів виконання. Реалізація програмних засобів з використанням мови програмування C# та відповідних API. Тестування та оцінка ефективності розроблених програмних засобів.

Методи дослідження: Для досягнення поставлених цілей та вирішення завдань використовуються наступні методи дослідження: Аналіз літературних джерел та наукових статей, що стосуються процесів виконання програмного коду та аналізу програм. Дослідження

сучасних технологій розробки програмного забезпечення та мови програмування C#. Розробка архітектури програмних засобів з використанням паттернів проектування та компонентного підходу. Реалізація програмних засобів з використанням мови програмування C# та відповідних API. Тестування розроблених програмних засобів та аналіз результатів.

Практичне значення: Результати даного дослідження мають практичне значення для розробників програмного забезпечення. Розроблені програмні засоби дозволять ефективно аналізувати процеси виконання програмного коду, що сприятиме виявленню та виправленню можливих проблем. Це дозволить покращити якість та надійність програмного забезпечення. Крім того, розроблені засоби можуть бути використані як основа для подальшого розширення та розвитку інструментів аналізу програмного коду. Отже, дана дипломна робота присвячена розробленню програмних засобів для аналізу процесів під час виконання програмного коду.

1. Теоретичні засади програмного коду та інструментарій

Програмний код є набором інструкцій, написаних мовою програмування, які

виконуються комп'ютером для виконання певних завдань. Він складається з послідовності команд, операторів, функцій та структур даних, які визначають поведінку програми. Код може бути написаний на різних мовах програмування, таких як C++, Java, Python, але для дипломної роботи вибрано мову C#.

Програмні засоби для аналізу процесів під час виконання програмного коду надають можливості для моніторингу, аналізу та вимірювання різних аспектів виконання програми. Вони забезпечують збір та аналіз даних про використання ресурсів системи, швидкість виконання, поведінку процесів та взаємодію зовнішніх компонентів. Такі програмні засоби можуть використовуватись для виявлення помилок, вимірювання ефективності, профілювання та оптимізації коду.

Для розроблення програмних засобів для аналізу процесів під час виконання програмного коду вибрано мову програмування C#. C# є об'єктно-орієнтованою мовою програмування, розробленою Microsoft, яка має вбудовану підтримку для розробки програмного забезпечення на платформі .NET. Вона надає потужні засоби для створення інструментів, аналізу та моніторингу програмного коду.

1.1.Процес виконання програмного коду

Виконання програмного коду на C# включає кілька етапів, які я надалі опишу:

1. Компіляція: Спочатку вихідний код на C# потрібно скомпілювати у виконуваний код, який може бути розуміним і виконаним операційною системою. Компіляцію виконує компілятор C#, який перетворює вихідний код у машинний код або у проміжний код, такий як Microsoft Intermediate Language (MSIL) або Common Intermediate Language (CIL). Якщо використовується проміжний код, він зазвичай зберігається у файлі з розширенням ".exe" або ".dll".

2. Завантаження: Після компіляції програму можна завантажити і виконати. Якщо це програма з розширенням ".exe", вона може бути запущена безпосередньо. Якщо це бібліотека класів з розширенням ".dll", її можна використовувати імпортувати у інші

програми.

3. Виконання: Під час виконання програми операційна система або середовище виконання C# (наприклад, .NET Framework або .NET Core) виконує проміжний код або машинний код, створений під час компіляції. Виконання включає в себе виконання кожного оператора, виклик функцій та обробку подій.

4. Взаємодія з операційною системою: Програма на C# може взаємодіяти з операційною системою, викликаючи функції API, отримуючи доступ до файлової системи, мережі, пристроїв вводу-виводу, баз даних та інших ресурсів.

5. Завершення: Після виконання програми або коли програма доходить до вказаної точки вихідного коду, вона може завершити своє виконання. У цей момент можуть виконуватися різні завершальні дії, такі як звільнення пам'яті, закриття відкритих файлів або ресурсів, зупинка виконання потоків тощо.

Це загальна схема виконання програмного коду на C#. Конкретні деталі можуть залежати від середовища виконання та платформи, на якій виконується програма.

1.1.1 Компіляція та інтерпретація

Процес виконання програмного коду включає компіляцію та інтерпретацію. Компіляція - це процес перетворення вихідного коду програми, написаного на високорівневій мові програмування, у машинний код, який може виконуватись безпосередньо процесором [1; с.48]. Компілятор перетворює вихідний код у вигляді текстового файлу на рівні вихідного коду у вигляді об'єктних файлів або виконуваних файлів, які можуть бути запущені на відповідній платформі [2; с.105].

При розробці програми на мові C# за допомогою розроблювального середовища .NET, компілятор C# виконує процес компіляції. Він отримує вихідний код програми, написаний розробником, і перетворює його на машинний код, який може бути розумним і виконуваним процесором. Після компіляції, результатом може бути виконуваний файл (exe або dll), який можна запустити на відповідній платформі.

Компіляція є важливим етапом в процесі розробки програмного забезпечення, оскільки вона перетворює вихідний код у формат, зрозумілий комп'ютеру для виконання програми. Цей процес дозволяє розробникам створювати ефективні та швидкодіючі програми, які можуть бути виконані на різних платформах.

Інтерпретація, зі свого боку, передбачає поетапне виконання вихідного коду програми з використанням інтерпретатора. Інтерпретатор читає та виконує команди програми одну за одною без попередньої компіляції до машинного коду. Кожна команда інтерпретується і виконується під час виконання програми [3; с.73]. Цей процес надає гнучкість, але може призводити до повільного виконання порівняно з компіляцією.

Наприклад, інтерпретація використовується в мовах програмування, таких як Python або JavaScript. При запуску програми, інтерпретатор поетапно виконує кожну інструкцію програми, перетворюючи її на відповідні дії на виконавчому рівні. Цей процес забезпечує гнучкість, оскільки програму можна змінювати та виконувати без необхідності повторної компіляції.

Однак, інтерпретація може призводити до повільного виконання порівняно з компіляцією, оскільки кожна команда програми інтерпретується під час виконання. Це може бути особливо помітним у випадку великих програм або задач, які вимагають високої продуктивності. Однак, деякі мови програмування використовують різні методи оптимізації та техніки, щоб зробити інтерпретовані програми більш ефективними.

Таким чином, інтерпретація дозволяє зберегти гнучкість та спрощує розробку та відлагодження програм. Однак, її повільніша продуктивність порівняно з компіляцією може бути недоліком в деяких випадках, коли швидкодія є пріоритетом.

1.1.2 Специфіка використання C#

Мова програмування C# є однією з сучасних мов програмування, розроблених компанією Microsoft. Вона входить до сімейства мов програмування .NET і має синтаксис, схожий на мови Java та C++. C# є мовою загального призначення, яка підтримує об'єктно-орієнтований підхід до програмування і надає багато вбудованих можливостей для розробки програмного забезпечення [4; с.16].

Для компіляції програмного коду на мові C# зазвичай використовується компілятор C# (C# compiler), який надається разом з розроблювальним середовищем .NET. Цей компілятор перетворює вихідний код програми на мові C# у машинний код або міжкод (IL - Intermediate Language), залежно від налаштувань компіляції.

Основні інструменти, які надаються розробником для компіляції та виконання програмного коду на мові C#, включають:

Компілятор C# (C# Compiler): Це інструмент, який перетворює вихідний код програми на мові C# у виконуваний файл (exe або dll) або міжкод (IL), який потім можна виконати на платформі .NET.

Розроблювальне середовище .NET (IDE): Наприклад, Visual Studio або Visual Studio Code, які надають розширений набір інструментів для розробки, збирання та налагодження програм на мові C#.

.NET Runtime (CLR - Common Language Runtime): Це середовище виконання, яке забезпечує виконання програм на мові C# та інших мов програмування, що використовують платформу .NET. CLR відповідає за завантаження, компіляцію та виконання міжкоду (IL) на цільовій платформі.

.NET Framework або .NET Core: Це набір бібліотек, класів та рантайму, які надають додаткові функції та можливості для розробки програм на мові C#. .NET Framework був поширений раніше, але замінений на .NET Core, який є більш крос-платформеним та має нові функції. З версії .NET 5 і пізніше .NET Core був перейменований на .NET.

Після компіляції програмного коду на мові C# у виконуваний файл або міжкод, його можна запустити на відповідній платформі, яка підтримує середовище виконання .NET (CLR). Виконавчий файл можна запустити безпосередньо або виконати за допомогою віртуальної машини .NET (наприклад, за допомогою команди dotnet для .NET Core).

1.2 Аналіз програмного коду

Аналіз програмного коду є важливим етапом в розробці програмного

забезпечення і дозволяє виявляти помилки, вразливості та покращувати якість коду. Існує кілька видів аналізу програмного коду:

Статичний аналіз: Виконується без прямого виконання програми і досліджує програмний код для виявлення потенційних помилок, некоректного використання мовних конструкцій та стилевих проблем. Статичний аналіз може здійснюватися з використанням спеціальних інструментів або за допомогою вбудованих функцій розробчих середовищ.

Динамічний аналіз: Виконується під час реального виконання програми і дозволяє отримати інформацію про її виконання, стан змінних, шляхи виконання та інші динамічні характеристики. Динамічний аналіз допомагає виявити проблеми, пов'язані з пам'яттю, продуктивністю та взаємодією з іншими компонентами системи.

Функціональний аналіз: Оцінює відповідність програмного коду вимогам та специфікаціям. Цей вид аналізу перевіряє, чи виконуються очікувані результати при різних вхідних даних та умовах.

1.2.1 Інструменти аналізу коду

Для аналізу програмного коду використовуються різноманітні інструменти, які допомагають автоматизувати процес виявлення помилок, аналізу стилю та оцінки якості коду. Ось деякі з них:

Статичні аналізатори: Це програмні інструменти, які аналізують програмний код без його фактичного виконання. Вони виявляють потенційні проблеми в коді, такі як синтаксичні помилки, некоректне використання мовних конструкцій, проблеми безпеки, потенційні вразливості та стилеві недоліки. Деякі популярні статичні аналізатори включають Pylint для мови Python, ESLint для JavaScript, SonarQube і Checkstyle для Java.

Компілятори: Компілятори перевіряють синтаксичну правильність коду і виявляють помилки на етапі компіляції. Вони перевіряють правильність використання мовних конструкцій, типи даних і вимоги до синтаксису. Приклади компіляторів включають GCC для мови C/C++, javac для Java та компілятори мов .NET для C# і VB.NET.

Лінтери: Лінтери є інструментами, які перевіряють стиль і якість коду, використовуючи набір правил і рекомендацій. Вони допомагають забезпечити послідовність, зрозумілість і згодність з настановами в оформленні коду. Наприклад, Pylint і Flake8 для Python, ESLint для JavaScript, RuboCop для Ruby.

Інструменти аналізу вразливостей: Ці інструменти спрямовані на виявлення потенційних вразливостей в програмному коді, які можуть бути використані для атак на систему. Вони перевіряють код на наявність небезпечних практик програмування, таких як необроблені виключення, некоректна обробка введених даних, використання небезпечних функцій тощо. Приклади таких інструментів включають OWASP Dependency Check, SonarQube, Fortify.

Інструменти тестування покриття коду: Ці інструменти вимірюють, наскільки велика частина програмного коду покривається тестами. Вони допомагають ідентифікувати некритичні області, які потребують додаткових тестів, а також визначають якість тестів. Наприклад, JaCoCo для Java, coverage.py для Python.

Інструменти для автоматичного форматування коду: Ці інструменти автоматично форматують код згідно з визначеними стилями і правилами форматування. Вони допомагають забезпечити послідовність в оформленні коду і зменшують час, витрачений на ручне форматування. Приклади таких інструментів включають Prettier, Black, clang-format.

Це лише кілька прикладів інструментів аналізу коду. Вибір підходящих інструментів залежить від конкретного мови програмування та потреб проекту.

1.3. Основи архітектури програмного забезпечення

Проектування архітектури програмного забезпечення є важливим етапом у розробці програмного продукту. Воно включає в себе прийняття рішень про організацію компонентів системи, їх взаємодію, структуру даних, розподіл функціональності та інші аспекти, які впливають на якість, розширюваність та підтримуваність програмного забезпечення

1.3.1. Вибір паттернів проектування

При проектуванні архітектури програмного забезпечення для аналізу процесів

виконання було обрано декілька паттернів проектування, які допомагають досягти бажаної функціональності та забезпечити зручну організацію коду.

Паттерн "Спостерігач" (Observer): Цей паттерн дозволяє встановлювати залежність один до одного між об'єктами, таким чином, коли стан одного об'єкта змінюється, всі залежні об'єкти автоматично повідомляються і оновлюються. У нашому випадку, цей паттерн може бути використаний для нотифікації аналізаторів про зміни в стані програми під час виконання .

Паттерн "Фабричний метод" (Factory Method): Цей паттерн використовується для створення об'єктів без прив'язки до конкретних класів. Він дозволяє створювати різні типи об'єктів на основі загального інтерфейсу. У нашому випадку, фабричний метод може бути використаний для створення різних аналізаторів, які реалізують загальний інтерфейс аналізатора.

Паттерн "Одиночка" (Singleton): Цей паттерн дозволяє обмежити створення лише одного екземпляра класу і забезпечити глобальний доступ до цього екземпляра. У нашому випадку, його можна застосувати до деяких компонентів системи, які повинні бути доступні з будь-якої точки програми.

Паттерн "Компонувальник" (Composite): Цей паттерн дозволяє об'єднати об'єкти в деревоподібні структури і обробляти їх як один.

1.3.2. Компоненти системи

Компоненти системи є основними будівельними блоками розроблюваного програмного засобу. Архітектура системи включає такі компоненти:

Модуль відстеження: відповідає за відстеження виконання програмного коду і збір необхідних даних [9].

Модуль аналізу: здійснює аналіз зібраних даних і виявляє можливі проблеми в програмному коді [9].

Модуль візуалізації: відповідає за візуальне представлення результатів аналізу та взаємодію з користувачем через графічний інтерфейс [9].

Модуль збереження даних: забезпечує збереження результатів аналізу і доступ до

них для подальшого використання [9].

Модуль конфігурації: дозволяє користувачу налаштувати параметри аналізу та збереження даних [9]

Модуль взаємодії з користувачем: Цей модуль буде забезпечувати інтерфейс для взаємодії з користувачем. Він може включати графічний інтерфейс користувача (GUI) або командний рядок, який дозволить користувачеві ввести код для аналізу, встановити параметри моніторингу та переглядати результати аналізу.

Модуль збору даних: Цей модуль буде відповідати за збір інформації про виконання програми та використання ресурсів системи. Він буде включати механізми для отримання статистики процесора, пам'яті, дискового простору та інших ресурсів. Інформація буде збиратись в реальному часі та передаватись до модулів аналізу.

Ці компоненти взаємодіють між собою, щоб забезпечити функціональність програмного засобу для аналізу процесів виконання.

2. Програмні засоби для аналізу процесів виконання

Дипломна робота зосереджується на реалізації програмних засобів. Одним із використаних інструментів для розробки таких програмних засобів є мова програмування C#.

Мова програмування C# була обрана через свою популярність, широку підтримку в середовищі розробки .NET, а також зручний синтаксис і можливості для роботи з об'єктно-орієнтованою парадигмою. C# є однією з основних мов програмування для розробки програмних засобів на платформі .NET, що робить її відмінним вибором для реалізації програмних засобів аналізу процесів виконання

2.1 Використання мови програмування C#

Використання мови програмування C# дозволяє розробити потужні та ефективні програмні засоби для аналізу процесів виконання. Мова C# має багатий набір бібліотек і фреймворків, які сприяють реалізації функціональності, необхідної для аналізу виконання програмного коду [22].

Для розробки програмних засобів використовується інтегроване середовище розробки

(IDE) Visual Studio, яке забезпечує зручну роботу з мовою C# і надає широкий набір інструментів для створення програмних засобів. Visual Studio надає можливості для створення проектів, компіляції коду, налагодження програми та інші інструменти, що полегшують процес розробки [33].

Застосування мови програмування C# у розробці програмних засобів для аналізу процесів виконання дозволяє створювати ефективні та потужні інструменти, які можуть допомогти розробникам аналізувати та вдосконалювати виконання свого програмного коду.

2.2 Використання API для отримання даних про виконання коду

Використання API для отримання даних про виконання коду дозволяє програмним засобам збирати і аналізувати інформацію про процеси виконання. Це може бути корисно для виявлення швидкісних або ефективних проблем, виявлення вузьких місць у виконанні коду, а також для збору статистики та вдосконалення якості програмного продукту [23].

API для отримання даних про виконання коду можуть бути доступні через різні платформи та мови програмування. Наприклад, в середовищі .NET, існує декілька API, таких як Profiling API та Performance Counters API, які надають доступ до різноманітної інформації про процеси виконання [32].

Використання API для отримання даних про виконання коду є важливою складовою реалізації програмних засобів аналізу процесів виконання. Відповідні API дозволяють отримувати об'єктивну інформацію та показники про виконання коду, що допомагає розробникам вдосконалювати та оптимізувати свої програми [41].

2.3 Тестування та оцінка ефективності програмних продуктів

Тестування та оцінка ефективності є важливими етапами у розробці програмного забезпечення. Вони дозволяють перевірити, чи відповідає програмний продукт вимогам, функціонує належним чином та забезпечує необхідну продуктивність

2.3.1 Методики тестування програмних засобів

Модульне тестування: Цей підхід включає тестування окремих модулів програмного засобу для перевірки їх правильності та функціональності. Використовуються спеціальні тестові сценарії, які перевіряють відповідність очікуваним результатам виконання коду [12].

Функціональне тестування: Ця методика передбачає перевірку функціональності програмного засобу, зокрема його відповідності вимогам і специфікаціям. Виконуються тестові сценарії, що охоплюють різні сценарії використання та перевіряють правильність реакції програмного засобу на вхідні дані [25].

Навантажувальне тестування: Цей підхід орієнтований на оцінку продуктивності програмного засобу під реальним навантаженням. Застосовуються спеціальні тестові сценарії, що генерують велику кількість запитів до програмного засобу з метою перевірки його швидкодії та стабільності [311].

Тестування безпеки: Ця методика спрямована на виявлення потенційних вразливостей та проблем безпеки в програмному засобі. Застосовуються різні тестові сценарії, які включають перевірку автентифікації, контроль доступу, захист від SQL-ін'єкцій та інші аспекти безпеки [422].

2.3.2 Аналіз результатів тестування

В цьому підпункті проводиться оцінка отриманих результатів тестування програмних засобів для аналізу процесів під час виконання програмного коду.

Під час аналізу результатів тестування враховуються такі аспекти:

Якість програмного засобу: Оцінюється відповідність програмного засобу вимогам та специфікаціям, його функціональність і стабільність.

Продуктивність: Аналізується швидкодія програмного засобу та його реакція на навантаження.

Безпека: Визначається рівень захищеності програмного засобу від потенційних загроз та вразливостей.

Виявлення помилок: Оцінюється кількість та серйозність виявлених помилок під час тестування, а також їх вплив на роботу програмного засобу.

Відповідність цілям та очікуванням: Перевіряється, наскільки програмний засіб задовольняє поставлені цілі та відповідає очікуванням користувачів.

3. ПОСТАНОВКА ЗАВДАННЯ

3.1 Вхідні дані для виконання проекту

Вхідними данними для виконання проекту є код:

```
using System;

using System.IO;

namespace CompileSample

{

  public class Process

  {

          public void Execute()

         {

          string[] lines = new string [1000000];

          for (int i = 0;  i < 1000000; i++)

                  lines[i] = i.ToString();

          File.WriteAllLines(“”c:\\test.txt””, lines);

         }

  }

}
```

3.2 Очікувані результати

Розроблення програмного рішення, яке здійснюватиме аналіз процесів під час виконання програмного коду. Це включає розробку модулів, які будуть відповідальні за моніторинг та аналіз різних аспектів виконання програми.

Моніторинг та аналіз різних аспектів виконання програми, включаючи використання ресурсів системи (такі як процесор, пам'ять, дисковий простір), швидкість виконання, поведінку процесів та взаємодію зовнішніх компонентів. Розроблене програмне рішення буде здатне відстежувати ці аспекти в реальному часі та збирати відповідні дані для подальшого аналізу.

Відслідковування роботи програми в реальному часі. Це дозволить користувачеві отримувати актуальну інформацію про виконання програми та її поведінку в реальному часі, що може бути корисним для виявлення проблем та вдосконалення продуктивності.

Збір та аналіз даних про використання процесора, пам'яті, дискового простору та інших ресурсів. Розроблене програмне рішення буде здатне збирати дані про використання ресурсів системи під час виконання програми і аналізувати їх для виявлення проблем, оптимізації та покращення ефективності.

Профілювання та виявлення можливих проблем ефективності. Розроблене програмне рішення буде здатне здійснювати профілювання програмного коду, виявляти можливі проблеми ефективності та рекомендувати шляхи їх вирішення.

Підтримка різних платформ та операційних систем. Розроблене програмне рішення буде забезпечувати сумісність з різними платформами та операційними системами, що дозволить його використання на різноманітних пристроях та середовищах.

Забезпечення зручного інтерфейсу для користувача. Розроблене програмне рішення буде мати інтуїтивно зрозумілий та зручний інтерфейс, який дозволить користувачеві здійснювати моніторинг та аналіз в зручний спосіб, відображати результати аналізу у зрозумілій формі та надавати необхідну інформацію для подальшого вдосконалення програмного коду.

3.3 Архітектура рішення

Архітектура програмного рішення для аналізу процесів під час виконання програмного коду буде побудована на основі модульної структури, що дозволить розбити систему на логічні компоненти та встановити взаємозв'язки між ними. Основні компоненти, які будуть визначені в архітектурі, включають:

Модуль збору даних: Цей модуль буде відповідати за збір інформації про виконання програми та використання ресурсів системи. Він буде включати механізми для отримання

статистики процесора, пам'яті, дискового простору та інших ресурсів. Інформація буде збиратись в реальному часі та передаватись до модулів аналізу.

```
using System;
using System.Collections.Generic;
using System.IO;
class DataCollector
{
   private List<string> data;
   public DataCollector()
   {
      data = new List<string>();
   }
      public void CollectData(string input)
   {
      data.Add(input);
   }
      public void SaveDataToFile(string filePath)
   {
      try
      {
         File.WriteAllLines(filePath, data);
         Console.WriteLine("Дані було успішно збережено у файлі: " + filePath);
```

```
        }
        catch (Exception ex)
        {
            Console.WriteLine("Помилка при збереженні даних: " + ex.Message);
        }
    }
}
 class Program
{
    static void Main()
    {
        DataCollector collector = new DataCollector();
        Console.WriteLine("Введіть дані (для виходу введіть 'exit'):");
        while (true)
        {
            string input = Console.ReadLine();
            if (input.ToLower() == "exit")
                break;
            collector.CollectData(input);
        }
            Console.WriteLine("Введення даних завершено.");
```

```
        Console.WriteLine("Введіть шлях до файлу для збереження даних:");
        string filePath = Console.ReadLine();
        collector.SaveDataToFile(filePath);
        Console.WriteLine("Натисніть будь-яку клавішу для виходу.");
        Console.ReadKey();
    }
}
```

У цьому прикладі ми створюємо клас DataCollector, який зберігає дані у списку. Метод CollectData додає введені дані до списку. Метод SaveDataToFile зберігає зібрані дані у файл.

У головному методі Main ми створюємо екземпляр DataCollector, пропонуємо користувачеві вводити дані, які додаються до колекції. Після завершення введення, користувачу пропонується ввести шлях до файлу, у який будуть збережені дані. Коли користувач вводить шлях, викликається метод SaveDataToFile для збереження даних у файл.

Модуль аналізу: Цей модуль буде відповідати за обробку та аналіз отриманих даних. Він буде використовувати алгоритми та методи для виявлення можливих проблем ефективності, аналізу поведінки процесів та взаємодії зовнішніх компонентів. Результати аналізу будуть подаватись у зручній формі для подальшого використання та візуалізації.

```
using System;
using System.Collections.Generic;
class DataAnalyzer
{
    public void AnalyzeData(List<string> data)
    {
        // Виконати аналіз даних тут
        // Наприклад, підрахувати кількість елементів у списку
```

```
        int count = data.Count;
        // Вивести результат аналізу
        Console.WriteLine("Кількість елементів у списку: " + count);
    }
}
class Program
{
    static void Main()
    {
        List<string> data = new List<string>() { "Елемент 1", "Елемент 2", "Елемент 3" };
        DataAnalyzer analyzer = new DataAnalyzer();
        analyzer.AnalyzeData(data);
        Console.WriteLine("Натисніть будь-яку клавішу для виходу.");
        Console.ReadKey();
    }
}
```

У цьому прикладі ми створюємо клас DataAnalyzer, який містить метод AnalyzeData, що приймає список даних для аналізу. У цьому методі можна виконати різні операції аналізу даних відповідно до поставленої задачі.

У головному методі Main ми створюємо список data, який містить прикладові дані для аналізу. Потім ми створюємо екземпляр DataAnalyzer і викликаємо метод AnalyzeData, передаючи йому список даних. В прикладі ми просто підраховуємо кількість елементів у списку і виводимо цей результат на консоль.

Цей код є лише загальним прикладом, і ви можете додати більше функціональності до методу AnalyzeData, щоб виконувати більш складний аналіз даних відповідно до ваших

потреб.

Модуль взаємодії з користувачем: Цей модуль буде забезпечувати інтерфейс для взаємодії з користувачем. Він може включати графічний інтерфейс користувача (GUI) або командний рядок, який дозволить користувачеві ввести код для аналізу, встановити параметри моніторингу та переглядати результати аналізу.

```
using System;

class UserInteractionModule

{

   public void Start()

   {

      Console.WriteLine("Ласкаво просимо!");

      // Запитати ім'я користувача

      Console.Write("Будь ласка, введіть своє ім'я: ");

      string name = Console.ReadLine();

      // Вивести привітання з ім'ям користувача

      Console.WriteLine("Привіт, " + name + "!");

      // Запитати вік користувача

      Console.Write("Скільки вам років? ");

      int age = Convert.ToInt32(Console.ReadLine());



      // Перевірити, чи користувач повнолітній
```

```
        if (age >= 18)
        {
            Console.WriteLine("Ви повнолітній!");
        }
        else
        {
            Console.WriteLine("Ви не повнолітній.");
        }
        // Завершення взаємодії
        Console.WriteLine("Дякую за участь. До побачення!");
    }
}
class Program
{
    static void Main()
    {
        UserInteractionModule interactionModule = new UserInteractionModule();
        interactionModule.Start();
        Console.WriteLine("Натисніть будь-яку клавішу для виходу.");
        Console.ReadKey();
    }
}
```

У цьому прикладі ми створюємо клас UserInteractionModule, який містить метод Start, який починає взаємодію з користувачем. У цьому методі ми використовуємо Console.WriteLine для виведення повідомлень на консоль та Console.ReadLine для зчитування введених користувачем даних.

В прикладі спочатку просимо користувача ввести своє ім'я, потім виводимо привітання з цим ім'ям. Потім запитуємо вік користувача, перевіряємо, чи він повнолітній, і виводимо відповідне повідомлення. Нарешті, закінчуємо взаємодію з користувачем і виводимо прощальне повідомлення.

Ви можете розширити цей приклад, додавши більше запитів і логіки взаємодії з користувачем залежно від ваших потреб.

Модуль збереження даних: Цей модуль буде відповідати за збереження зібраних даних та результатів аналізу. Він може використовувати базу даних або файлову систему для зберігання інформації. Це дозволить зберегти дані для подальшого аналізу та забезпечити можливість відновлення результатів аналізу.

```
using System;
using System.IO;
class DataStorageModule
{
    public void SaveDataToFile(string data, string fileName)
    {
        try
        {
            // Відкриття або створення файлу для збереження даних
            using (StreamWriter writer = new StreamWriter(fileName))
            {
```

```
            // Запис даних у файл
                writer.WriteLine(data);
            }
            Console.WriteLine("Дані було успішно збережено у файлі " + fileName);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Помилка при збереженні даних: " + ex.Message);
        }
    }
}
class Program
{
    static void Main()
    {
        DataStorageModule storageModule = new DataStorageModule();
        Console.Write("Введіть дані для збереження: ");
        string data = Console.ReadLine();
        Console.Write("Введіть назву файлу: ");
        string fileName = Console.ReadLine();

        storageModule.SaveDataToFile(data, fileName);
```

```
            Console.WriteLine("Натисніть будь-яку клавішу для виходу.");
            Console.ReadKey();
        }
    }
```

У цьому прикладі ми створюємо клас DataStorageModule, який містить метод SaveDataToFile, призначений для збереження даних у файл. У цьому методі ми використовуємо StreamWriter для відкриття або створення файлу і записуємо дані до файлу за допомогою writer.WriteLine(data).

В прикладі спочатку просимо користувача ввести дані для збереження та назву файлу. Потім викликаємо метод SaveDataToFile з модуля збереження даних, передаючи йому введені дані та назву файлу. Метод спробує зберегти дані у файлі, а у разі помилки виведе відповідне повідомлення про помилку.

Ви можете модифікувати цей приклад для збереження різних типів даних у різних форматах або використовувати різні механізми збереження даних, такі як бази даних.

Модуль відстеження: Цей модуль буде відповідати за відстеження роботи програми в реальному часі. Він буде збирати дані про послідовність виконання операцій, стан змінних, виклики функцій тощо. Інформація, зібрана цим модулем, може використовуватись для детального аналізу взаємодії компонентів програми та виявлення потенційних помилок.

```
using System;
using System.Diagnostics;
class TrackingModule
{
    private Stopwatch stopwatch;

    public void StartTracking()
```

```
        {
            stopwatch = Stopwatch.StartNew();
            Console.WriteLine("Початок відстеження.");
        }
        public void StopTracking()
        {
            stopwatch.Stop();
            Console.WriteLine("Зупинка відстеження. Загальний час виконання: " + stopwatch.Elapsed);
        }
    }
    class Program
    {
        static void Main()
        {
            TrackingModule trackingModule = new TrackingModule();
            Console.WriteLine("Натисніть Enter, щоб почати відстеження...");
            Console.ReadLine();
            trackingModule.StartTracking();
            // Ваш код, який потребує відстеження
            Console.WriteLine("Натисніть Enter, щоб зупинити відстеження...");
            Console.ReadLine();
```

```
            trackingModule.StopTracking();

            Console.WriteLine("Натисніть будь-яку клавішу для виходу.");

            Console.ReadKey();

        }

    }
```

У цьому прикладі ми створюємо клас TrackingModule, який містить методи StartTracking та StopTracking. При виклику методу StartTracking ми починаємо відлік часу за допомогою Stopwatch.StartNew(). При виклику методу StopTracking зупиняємо відлік часу та виводимо загальний час виконання.

У прикладі ми просимо користувача натиснути Enter, щоб почати відстеження. Потім викликаємо метод StartTracking з модуля відстеження. Після цього можна виконувати ваш код, який потребує відстеження. Після завершення коду користувач натискає Enter, щоб зупинити відстеження, і викликається метод StopTracking, який виводить загальний час виконання.

Ви можете модифікувати цей приклад, додавши додаткову функціональність до відстежування, наприклад, вимірювання часу виконання окремих частин коду або збір інших метрик виконання.

Модуль візуалізації: Цей модуль буде відповідати за візуалізацію результатів аналізу та відстеження. Він може створювати графіки, діаграми, дашборди або інші візуальні елементи, що допоможуть користувачеві краще зрозуміти інформацію про виконання програми та використання ресурсів системи.

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;
```

```
using System.Windows.Forms;
using System.Drawing;
namespace VisualizationModuleExample
{
    class VisualizationModule
    {
        private Form visualizationForm;
        public void ShowVisualization()
        {
            // Створюємо форму візуалізації
            visualizationForm = new Form();
            visualizationForm.Text = "Модуль візуалізації";
            visualizationForm.Size = new Size(800, 600);
            // Додаємо елементи візуалізації до форми
            DrawVisualization();
           // Відображаємо форму візуалізації
            Application.Run(visualizationForm);
        }
        private void DrawVisualization()
        {
            // Створюємо графічний контекст для малювання
            Graphics graphics = visualizationForm.CreateGraphics();
```

```
            // Малюємо просту фігуру
            Pen pen = new Pen(Color.Red, 3);
            graphics.DrawRectangle(pen, new Rectangle(100, 100, 200, 150));
        }
    }
    class Program
    {
        static void Main()
        {
            VisualizationModule visualizationModule = new VisualizationModule();
            // Виклик модулю візуалізації
            visualizationModule.ShowVisualization();
        }
    }
}
```

У цьому прикладі ми створюємо клас VisualizationModule, який містить метод ShowVisualization. У цьому методі ми створюємо форму візуалізації, задаємо її розмір та назву. У методі DrawVisualization ми використовуємо графічний контекст форми для малювання фігури (у цьому прикладі - прямокутника червоного кольору). Форма візуалізації відображається за допомогою Application.Run.

У основному методі Main ми створюємо екземпляр VisualizationModule і викликаємо його метод ShowVisualization.

Ви можете модифікувати цей приклад, додавши різні елементи візуалізації, використовуючи графічні примітиви, зображення, діаграми та інші елементи для відображення даних або процесів у вашій програмі.

Модуль конфігурації: Цей модуль дозволить користувачеві налаштовувати параметри аналізу та моніторингу в залежності від вимог та потреб. Він може включати інтерфейс для встановлення параметрів, вибору компонентів для аналізу, визначення правил спостереження тощо. Модуль конфігурації забезпечує гнучкість та настроювання рішення для конкретних потреб користувача.

```
using System;
using System.Collections.Generic;
using System.Configuration;
namespace ConfigurationModuleExample
{
   class ConfigurationModule
   {
      public void ReadConfiguration()
      {
         // Отримання значень конфігураційних параметрів
         string setting1 = ConfigurationManager.AppSettings["Setting1"];
         string setting2 = ConfigurationManager.AppSettings["Setting2"];
         // Використання отриманих значень
         Console.WriteLine("Setting1: " + setting1);
         Console.WriteLine("Setting2: " + setting2);
      }

      public void WriteConfiguration(string setting1, string setting2)
```

```
        {
            // Запис значень конфігураційних параметрів
            Configuration config = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
            config.AppSettings.Settings["Setting1"].Value = setting1;
            config.AppSettings.Settings["Setting2"].Value = setting2;
            config.Save(ConfigurationSaveMode.Modified);
            // Перезавантаження конфігурації
            ConfigurationManager.RefreshSection("appSettings");
            Console.WriteLine("Configuration updated.");
        }
    }
    class Program
    {
        static void Main()
        {
            ConfigurationModule configurationModule = new ConfigurationModule();
            // Читання конфігурації
            configurationModule.ReadConfiguration();
            // Запис конфігурації
            configurationModule.WriteConfiguration("Value1", "Value2");

            // Повторне читання конфігурації після оновлення
```

```
            configurationModule.ReadConfiguration();
        }
    }
}
```

У цьому прикладі ми використовуємо клас ConfigurationManager для отримання і запису конфігураційних параметрів. У методі ReadConfiguration ми отримуємо значення параметрів Setting1 та Setting2 з розділу appSettings конфігураційного файлу і використовуємо їх. У методі WriteConfiguration ми записуємо нові значення для цих параметрів, зберігаємо зміни та перезавантажуємо конфігурацію. Потім ми читаємо конфігурацію знову для перевірки оновлених значень.

Важливо мати на увазі, що використання конфігураційного файлу передбачає наявність файлу app.config (для програми на основі .NET Framework) або appsettings.json (для програми на основі .NET Core) з відповідними розділами та ключами значень параметрів.

Взаємодія між компонентами системи буде здійснюватись через визначені інтерфейси та обмін даними. Наприклад, модуль збору даних буде передавати зібрану інформацію модулю аналізу для подальшого оброблення. Модуль взаємодії з користувачем буде взаємодіяти з усіма іншими модулями, надаючи користувачеві можливість керувати аналізом та отримувати результати.

Дана архітектура розроблена з урахуванням принципів модульності, гнучкості та розширюваності. Вона дозволить забезпечити ефективний аналіз процесів виконання програмного коду та зручний інтерфейс для користувача.

3.4. Вибір і обгрунтування оптимальних засобів та технологій

1.Широке застосування: C# є однією з найпопулярніших мов програмування, яка має широке застосування в розробці програмного забезпечення. Вона підтримується платформою .NET і використовується для розробки різноманітних додатків, включаючи веб-додатки, настільні програми та мобільні додатки.

2.Сильна типізація: C# має сильну типізацію, що дозволяє виявляти помилки під час компіляції і запобігає багатьом типовим помилкам під час виконання програми. Це сприяє покращенню надійності та стабільності розроблюваного рішення.

3.Багата екосистема: C# має багату екосистему інструментів та фреймворків, що полегшують розробку програмного забезпечення. Наприклад, Visual Studio є потужним інтегрованим середовищем розробки, яке надає різноманітні інструменти для побудови та налагодження додатків на C#. Також існує велика кількість фреймворків і бібліотек, таких як ASP.NET для веб-розробки, WPF для розробки настільних додатків, Xamarin для розробки мобільних додатків, що розширюють можливості мови C#.

4.Підтримка ОС Windows: C# є основною мовою програмування для розробки додатків для операційних систем Windows. Якщо проект передбачає використання рішення на цій платформі, використання C# дозвол

ить використовувати повні можливості цієї ОС та інтегруватись з іншими технологіями Microsoft.

5.Зручний синтаксис: C# має чистий і зручний синтаксис, що сприяє швидкій розробці програмного забезпечення та покращує читабельність коду. Він підтримує об'єктно-орієнтовану парадигму, а також функціональне програмування через використання лямбда-виразів та LINQ.

Обрання мови C# відповідає вимогам проекту, оскільки вона є потужним інструментом для розробки програмного рішення з аналізу процесів під час виконання програмного коду. Її

можливості, розширена екосистема та підтримка ОС Windows роблять C# ефективним вибором для реалізації цього проекту.

3.5 Практична реалізація

Цей код виконує наступні кроки:

Спочатку я обрав необхідні бібліотеки для роботи з проектом.

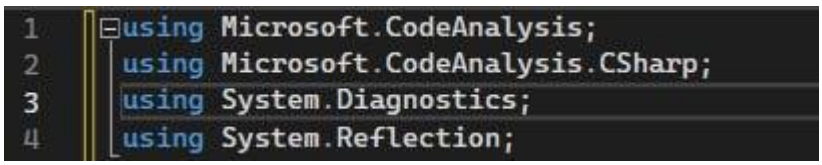
1. Microsoft.CodeAnalysis: Бібліотека Microsoft.CodeAnalysis надає набір інструментів для роботи з кодом на основі .NET. Вона включає компілятори, аналізатори та інші компоненти, що дозволяють аналізувати, змінювати і генерувати програмний код. Ця бібліотека дозволяє вам створювати програми, які маніпулюють кодом на рівні синтаксису, дерева синтаксису та семантичного моделю.

Microsoft.CodeAnalysis.CSharp: Бібліотека Microsoft.CodeAnalysis.CSharp є частиною Microsoft.CodeAnalysis і надає специфічні інструменти для роботи з кодом на мові C#. Вона містить компоненти для синтаксичного аналізу, аналізу семантики, генерації коду та багато іншого. Завдяки цій бібліотеці можна аналізувати та маніпулювати кодом на C#, створювати рефакторинги, виконувати статичний аналіз, автоматично генерувати код і багато іншого.

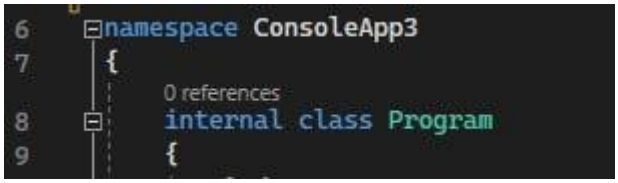
System.Diagnostics: Бібліотека System.Diagnostics надає інструменти для взаємодії з системними процесами, відлагодженням і профілюванням додатків. Вона містить класи, які дозволяють запускати зовнішні процеси, керувати потоками вводу/виводу, отримувати інформацію про процесор та пам'ять, а також здійснювати налаштування та моніторинг процесів.

System.Reflection: Бібліотека System.Reflection надає інструменти для роботи з метаданими програми під час її виконання. Вона дозволяє отримувати інформацію про типи, методи, властивості та інші складові програми, а також динамічно створювати, викликати та маніпулювати цими складовими. Це дозволяє робити рефлексію, динамічно завантажувати збірки, створювати інстанції об'єктів та багато іншого.

Ці бібліотеки є потужними інструментами для розробки програмного забезпечення. Вони дозволяють аналізувати та маніпулювати кодом, взаємодіяти з процесами та виконувати динамічні операції під час виконання програми. Використовуючи ці бібліотеки, ви можете реалізувати різноманітні функції, такі як автоматичний рефакторинг, статичний аналіз коду, роботу з відлагоджувачем та багато іншого.



мал. 1. Використання основних бібліотек.



мал. 2 Визнення простір імен ConsoleApp3 і внутрішній клас Program.

В методі Main створюється об'єкт syntaxTree, який представляє дерево синтаксису C# для вхідного коду. CSharpSyntaxTree - це клас, який представляє синтаксичне дерево для джерела коду на мові C#. Він є частиною простору імен Microsoft.CodeAnalysis.CSharp.Syntax в бібліотеці Microsoft.CodeAnalysis.CSharp.

Синтаксичне дерево (Syntax Tree) - це структура даних, яка відображає синтаксичну структуру джерела коду. Вона відображає всі синтаксичні елементи, такі як класи, методи, оператори, вирази та інші, і встановлює їхній взаємозв'язок та ієрархію. Синтаксичне дерево використовується для аналізу та маніпуляції кодом під час компіляції або виконання програми.

CSharpSyntaxTree забезпечує доступ до синтаксичного дерева для конкретного джерела коду на C#. Воно може бути створене за допомогою різних методів, наприклад, з рядка коду або з файлу. Цей об'єкт містить інформацію про структуру коду, включаючи список кореневих вузлів дерева, взаємозв'язки між ними та додаткову метаінформацію.

CSharpSyntaxTree надає різноманітні методи і властивості для роботи з синтаксичним деревом. Наприклад, ви можете отримати кореневий вузол дерева за допомогою властивості CSharpSyntaxTree.Root, яка дає доступ до синтаксичного вузла CompilationUnitSyntax. Ви також можете використовувати методи GetRoot() та GetCompilationUnitRoot() для отримання кореневого вузла.

З CSharpSyntaxTree ви можете виконувати різні операції над синтаксичним деревом, такі як пошук конкретних вузлів, заміна або видалення вузлів, отримання деталей про синтаксичні елементи, отримання місцезнаходження в коді та багато іншого. Він є важливим інструментом для реалізації аналізаторів коду, рефакторингів, інструментів автоматичної генерації коду та інших задач, пов'язаних з аналізом та маніпуляцією коду на мові C#.

```
        {
            var syntaxTree = CSharpSyntaxTree.ParseText(@"
using System;
using System.IO;

namespace CompileSample
{
    public class Process
    {
        public void Execute()
        {
            string[] lines = new string[1000000];
            for (int i = 0; i < 1000000; i++)
                lines[i] = i.ToString();

            File.WriteAllLines(""c:\\test.txt"", lines);
        }
    }
}");
```

мал. 3. Основний код для аналізу процесів.

Генерується унікальне ім'я для збірки і створюються посилання на необхідні метадані (референси) для компіляції коду. var assemblyName = Path.GetRandomFileName();:

Цей рядок коду створює випадкове ім'я для збірки (assembly). Path.GetRandomFileName() повертає рандомне унікальне ім'я файлу, яке буде використовуватися як ім'я збірки.

var references = new MetadataReference[] { ... };:

В цьому рядку коду створюється масив references (посилань), який містить посилання на залежності (референси), які використовуються під час компіляції. У даному випадку створюються два посилання на метадані (MetadataReference) з використанням MetadataReference.CreateFromFile(). Перше посилання посилається на збірку, яка містить тип object, а друге - на збірку, яка містить тип Enumerable. Ці посилання дозволяють компілятору вирішити залежності під час компіляції.

var compilation = CSharpCompilation.Create(...);:

У цьому рядку коду створюється компіляція (compilation) з використанням CSharpCompilation.Create(). Компіляція включає в себе ім'я збірки (assemblyName), список синтаксичних дерев (syntaxTrees), посилання на залежності (references) та параметри компіляції (options). У даному випадку створюється компіляція з одним синтаксичним деревом (syntaxTree), яке відповідає вихідному коду на C#, та використовується попередньо задана група посилань (references). Також встановлюється параметр OutputKind.DynamicallyLinkedLibrary, який вказує, що результат компіляції буде динамічно підключуваною бібліотекою.

Усі ці кроки дозволяють створити компіляцію з вихідним кодом на C# і необхідними посиланнями для успішної компіляції та подальшого використання цієї збірки в програмі.

```
var assemblyName = Path.GetRandomFileName();
var references = new MetadataReference[]
{
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
    MetadataReference.CreateFromFile(typeof(Enumerable).Assembly.Location)
};

var compilation = CSharpCompilation.Create(
    assemblyName,
    syntaxTrees: new[] { syntaxTree },
    references: references,
    options: new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary)
);
```

мал. 4. Створення збірки та компіляції.

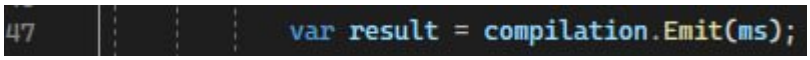
У цьому фрагменті коду створюється новий об'єкт MemoryStream з використанням ключового слова using. MemoryStream є потоком пам'яті, який може зберігати дані в оперативній пам'яті. Завдяки використанню using, об'єкт MemoryStream буде автоматично знищений після виходу з області видимості цього фрагмента коду.

```
45          using var ms = new MemoryStream();
```

мал. 5. Створення потоку пам’яті для збереження даних.

У цьому фрагменті коду викликається метод Emit() компіляції (compilation) з

передачею MemoryStream (ms) в якості параметра. Метод Emit() компілює код, представлений об'єктом компіляції, і зберігає результат компіляції у вказаному потоці пам'яті (MemoryStream). Результат компіляції представлений об'єктом типу EmitResult, і його значення присвоюється змінній result.



мал. 6. Виклик методу Emit компіляції.

Зазначений уривок коду виконує перевірку результату компіляції і виводить на консоль будь-які помилки або попередження, якщо компіляція не була успішною.

if (!result.Success) { ... }:

Ця умовна конструкція перевіряє, чи компіляція була успішною, шляхом перевірки властивості Success об'єкта result, який містить результат компіляції. Якщо Success має значення false, це означає, що виникли помилки або попередження під час компіляції.

IEnumerable<Diagnostic> failures = result.Diagnostics.Where(diagnostic => ...):

У цьому рядку коду створюється колекція failures типу IEnumerable<Diagnostic>, яка міститиме діагностики (помилки та попередження) з об'єкта result.Diagnostics. Використовується метод Where(), щоб відфільтрувати діагностики, вибираючи ті, які є помилками (diagnostic.Severity == DiagnosticSeverity.Error) або попередженнями, що потрібно розглядати як помилки (diagnostic.IsWarningAsError).

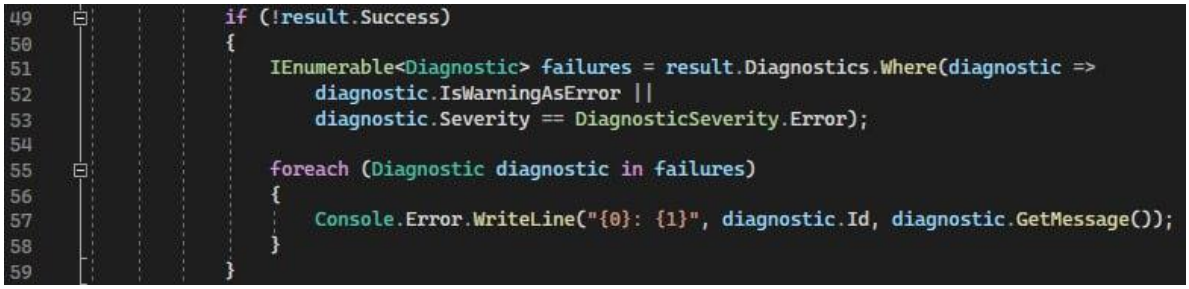
foreach (Diagnostic diagnostic in failures) { ... }:

У цьому циклі foreach кожна діагностика diagnostic з колекції failures перебирається по черзі. Це дозволяє вивести на консоль інформацію про кожну помилку або попередження.

Console.Error.WriteLine("{0}: {1}", diagnostic.Id, diagnostic.GetMessage()):

У цьому рядку коду виводиться повідомлення про помилку або попередження на

консоль помилок (Console.Error). Використовуються властивості Id і GetMessage() об'єкта diagnostic, щоб отримати ідентифікатор і повідомлення про помилку або попередження.



мал. 7. Перевірка компіляції.

Якщо компіляція успішна, відображається звіт про виконання коду. Збірка завантажується з MemoryStream, отримується тип класу "CompileSample.Process", створюється екземпляр цього класу і викликається його метод "Execute". Зазначений уривок коду виконується у випадку, якщо компіляція була успішною, і містить додаткові дії, пов'язані з обробкою результату компіляції.

ms.Seek(0, SeekOrigin.Begin);:

У цьому рядку коду виконується переміщення позиції читання/запису потоку MemoryStream (ms) на початок потоку. Це робиться за допомогою методу Seek(), вказуючи початкову позицію (0) і початок відліку (SeekOrigin.Begin).

var assembly = Assembly.Load(ms.ToArray());:

У цьому рядку коду завантажується збірка (assembly) з пам'яті, використовуючи метод Load() класу Assembly. Метод ToArray() викликається на MemoryStream (ms), щоб отримати масив байтів, який представляє дані збірки, і передається до методу Load() для завантаження збірки з цих даних.

var type = assembly.GetType("CompileSample.Process");:

У цьому рядку коду отримується тип (type) з завантаженої збірки (assembly). Використовується метод GetType(), якому передається рядок "CompileSample.Process", що є

повним іменем типу, який має бути отриманий з збірки.

var obj = Activator.CreateInstance(type);:

У цьому рядку коду створюється екземпляр об'єкта (obj) з типу (type). Використовується метод CreateInstance() класу Activator, якому передається тип type, щоб створити новий об'єкт цього типу. Отриманий об'єкт може бути використаний для подальшої роботи з функціональністю, визначеною в цьому типі.

Отже, цей кусок коду виконує додаткові дії, коли компіляція успішна. Він переносить позицію потоку MemoryStream на початок, завантажує збірку з пам'яті, отримує тип з завантаженої збірки і створює новий об'єкт цього типу. Ці дії дозволяють динамічно використовувати згенерований код після компіляції.

```
60          else
61          {
62              ms.Seek(0, SeekOrigin.Begin);
63              var assembly = Assembly.Load(ms.ToArray());
64              var type = assembly.GetType("CompileSample.Process");
65              var obj = Activator.CreateInstance(type);
```

мал.8. При успішній компіляції, динамічне використання коду.

Далі виконується вимірювання продуктивності системи під час виконання методу "Execute". Створюються об'єкти PerformanceCounter для вимірювання використання центрального процесора (CPU) та пам'яті. Запам'ятовується стан CPU і пам'яті перед викликом методу "Execute". Зазначений уривок коду виконує ініціалізацію та початкове вимірювання значень показників продуктивності, таких як використання процесора та пам'яті.

var cpu = new PerformanceCounter("Processor Information", "% Processor Time", "_Total");:

У цьому рядку коду створюється новий об'єкт PerformanceCounter для вимірювання використання процесора. Використовується конструктор PerformanceCounter, якому передається назва категорії ("Processor Information"), назва лічильника ("% Processor Time") та ідентифікатор екземпляра ("_Total"). Цей об'єкт cpu буде використовуватись для отримання

значень використання процесора.

var memory = new PerformanceCounter("Memory", "% Committed Bytes In Use");:

У цьому рядку коду створюється новий об'єкт PerformanceCounter для вимірювання використання пам'яті. Використовується конструктор PerformanceCounter, якому передається назва категорії ("Memory") та назва лічильника ("% Committed Bytes In Use"). Цей об'єкт memory буде використовуватись для отримання значень використання пам'яті.

var cpuBefore = cpu.NextValue();:

У цьому рядку коду отримується початкове значення використання процесора. Використовується метод NextValue() об'єкта cpu, який повертає наступне значення лічильника. Це значення записується в змінну cpuBefore для подальшого порівняння.

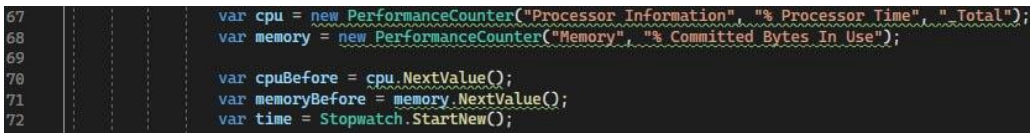
var memoryBefore = memory.NextValue();:

У цьому рядку коду отримується початкове значення використання пам'яті. Використовується метод NextValue() об'єкта memory, який повертає наступне значення лічильника. Це значення записується в змінну memoryBefore для подальшого порівняння.

var time = Stopwatch.StartNew();:

У цьому рядку коду створюється новий об'єкт Stopwatch для вимірювання часу. Використовується метод StartNew(), який створює новий об'єкт Stopwatch і починає вимірювання часу. Отриманий об'єкт time може бути використаний для вимірювання проміжків часу під час виконання коду.

Отже, цей кусок коду створює об'єкти PerformanceCounter для вимірювання використання процесора та пам'яті, отримує початкові значення цих показників і створює об'єкт Stopwatch для вимірювання часу виконання коду.



мал. 9. Створення об’єкту PerformanceCounter для початку вимірювання.

Зазначений уривок коду виконує виклик методу "Execute" на об'єкті, який має тип, отриманий раніше, використовуючи механізм відображення типів.

type.InvokeMember("Execute", BindingFlags.Default | BindingFlags.InvokeMethod, null, obj, null);:

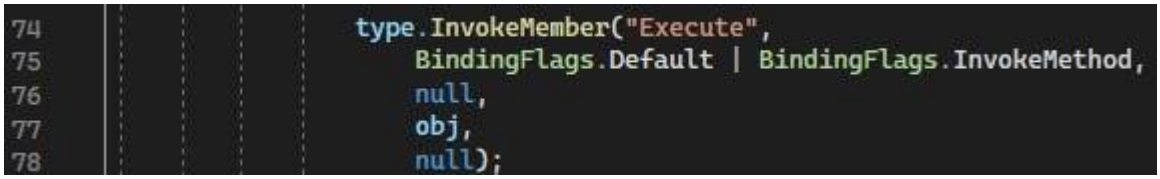
У цьому рядку коду виконується виклик методу "Execute" на об'єкті obj, який має тип type. Виклик методу здійснюється за допомогою методу InvokeMember() об'єкта type. Параметри методу InvokeMember() вказують назву методу ("Execute"), флаги зв'язування (BindingFlags.Default | BindingFlags.InvokeMethod), об'єкт-власник (obj) та додаткові параметри (null).

"Execute": Це рядок, що містить назву методу, який потрібно викликати.

BindingFlags.Default | BindingFlags.InvokeMethod: Це комбінація флагів зв'язування, що визначають тип доступу та поведінку виклику методу. Флаг BindingFlags.Default вказує використання значення за замовчуванням для флагів зв'язування. Флаг BindingFlags.InvokeMethod вказує, що потрібно викликати метод.

null: Це об'єкт, що представляє додаткові параметри, передані методу "Execute". У даному випадку, не передаються жодні додаткові параметри.

Отже, цей кусок коду викликає метод "Execute" на об'єкті obj з використанням типу type. Це дозволяє виконати функціональність, що визначена в цьому методі.



мал. 10. Виклик методу Execute.

Після виклику методу вимірюються нові значення CPU і пам'яті, обчислюється час виконання коду і виводяться результати. Зазначений уривок коду відображає та виводить на консоль інформацію про використання циклів процесора, використання пам'яті та час

виконання певного участка коду.

Console.WriteLine($"CPU cycles usage - from {cpuBefore} to {cpu.NextValue()}");:

У цьому рядку коду виводиться на консоль інформація про використання циклів процесора. За допомогою рядка форматування ($""), значення cpuBefore (початкове використання процесора) і поточне значення (cpu.NextValue()) виводяться у вигляді рядка на консоль. Ця інформація дозволяє відстежувати зміни використання процесора після виконання коду.

Console.WriteLine($"Memory usage - {memory.NextValue() - memoryBefore} Bytes");:

У цьому рядку коду виводиться на консоль інформація про використання пам'яті. За допомогою рядка форматування ($""), різниця між поточним значенням memory.NextValue() та memoryBefore (початкове використання пам'яті) обчислюється, і це значення виводиться у вигляді рядка на консоль. Ця інформація вказує на зміну використання пам'яті після виконання коду.

time.Stop();:

У цьому рядку коду зупиняється вимірювання часу. Викликається метод Stop() на об'єкті time (екземпляр класу Stopwatch), що припиняє вимірювання часу.

Console.WriteLine("Execution time - " + time.Elapsed);:

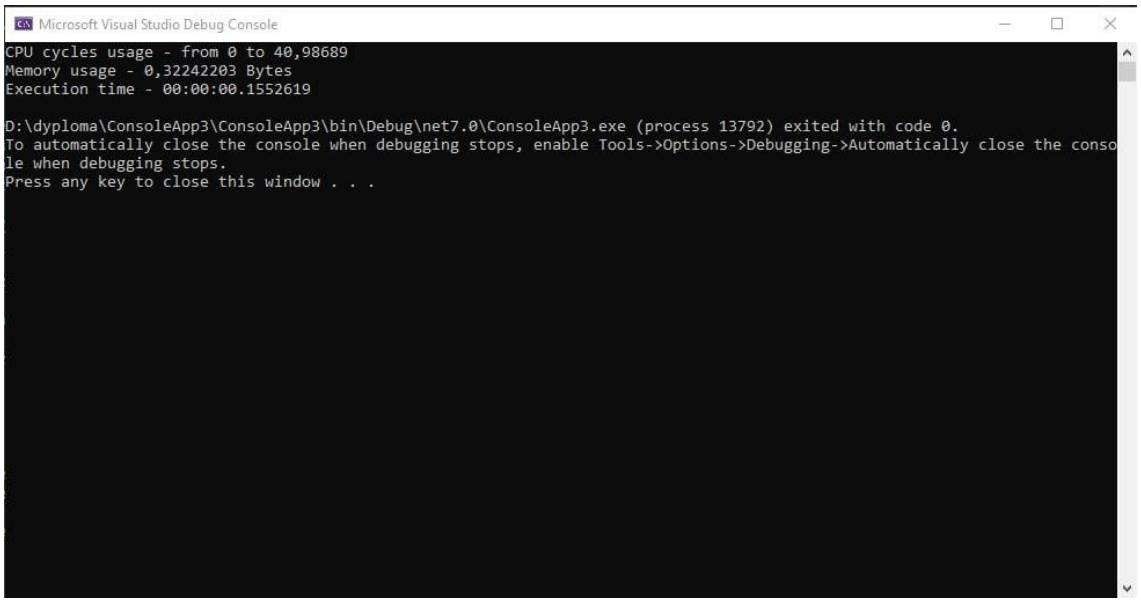
У цьому рядку коду виводиться на консоль інформація про час виконання коду. За допомогою рядкового додавання (+), рядок "Execution time - " об'єднується з значенням time.Elapsed (час, який пройшов з моменту початку вимірювання до моменту зупинки) і виводиться у вигляді рядка на консоль. Ця інформація показує загальний час виконання коду.

```
80                  Console.WriteLine($"CPU cycles usage - from {cpuBefore} to {cpu.NextValue()}");
81                  Console.WriteLine($"Memory usage - {memory.NextValue() - memoryBefore} Bytes");
82                  time.Stop();
83                  Console.WriteLine("Execution time - " + time.Elapsed);
84              }
85          }
86      }
```

мал. 11. Вивід результатів тестування, а саме значення CPU, пам’яті та час виконання

Цей код компілює вихідний код C# у збірку, завантажує збірку в пам'ять і виконує

вимірювання продуктивності під час виконання методу "Execute".



мал. 12. Результати.

Цей код виконує наступні кроки:

Створюється дерево синтаксису C# з вихідним кодом, яке містить клас Process з методом Execute. У цьому методі створюється масив рядків, заповнений числами від 0 до 999999, та записується у файл c:\\test.txt.

Створюється компіляція коду з використанням CSharpCompilation. Зазначається унікальне ім'я збірки, використовуються посилання на метадані object і Enumerable, і задається тип збірки DynamicallyLinkedLibrary.

Код компілюється за допомогою Emit і результат зберігається у MemoryStream.

Перевіряється успішність компіляції. Якщо виникли помилки, виводяться повідомлення про помилки.

Якщо компіляція успішна, збірка завантажується з MemoryStream, із неї витягується тип класу Process, створюється його екземпляр і викликається метод Execute.

Вимірюється продуктивність під час виконання методу Execute. Вимірюються значення використання CPU та пам'яті перед та після виклику методу, а також час виконання.

Виводяться результати вимірювання продуктивності: використання CPU, використання пам'яті та час виконання.

ВИСНОВКИ

Встановлено, що код демонструє використання бібліотеки Microsoft.CodeAnalysis для компіляції вихідного коду C# в пам'ять і його виконання; код також використовує MemoryStream для зберігання скомпільованої збірки в оперативній пам'яті замість збереження на диску; після компіляції та завантаження збірки викликається метод класу Process для виконання коду; код використовує PerformanceCounter для вимірювання продуктивності системи, такої як використання CPU та пам'яті, перед і після виконання методу; результати вимірювання продуктивності виводяться на консоль.

Проаналізовано теоретичні засади програмного коду програмного забезпечення та програмні засоби для аналізу процесів програмного коду.

Показано, що цей код корисний для випробування та вимірювання продуктивності коду C# і також він дозволяє компілювати та виконувати вихідний код динамічно, а також вимірювати різні метрики продуктивності під час виконання коду.

Порівнюючи Roslyn з іншою програмою, наприклад c#CodeDOM, можна вивести такі переваги:

1. Інтегрований аналіз програмного коду: Roslyn забезпечує повну аналіз програмного коду, включаючи розпізнавання синтаксису, перевірку типів і виявлення помилок. Вам не потрібно компілювати код, як у першій технології, щоб отримати аналіз.

2. Доступ до деталей програмного коду: Roslyn надає доступ до внутрішніх структур програмного коду, таких як дерева синтаксису, символи, типи тощо. Це дозволяє вам виконувати більш глибокий аналіз та маніпулювання з кодом.

3. Підтримка в реальному часі: Roslyn може працювати в режимі реального часу, що дозволяє виконувати аналіз інтерактивно під час редагування коду. Ви можете отримувати підказки, виправлення помилок та інші рекомендації без необхідності повторного компілювання.

4. Розширення можливостей: Завдяки відкритій природі Roslyn, ви можете створювати свої власні аналізатори і додаткові інструменти, що дозволяє розширювати можливості платформи.

5. Підтримка різних мов: Roslyn підтримує не тільки мову C#, але і VB.NET, що дає змогу використовувати його для аналізу коду на обох мовах.

Це лише кілька переваг Roslyn. Загалом, використання Roslyn надає більш широкі можливості для аналізу і маніпулювання програмним кодом, порівняно з першою технологією, яку ви представили.

Даний програмний продукт можна практично використовувати розробникам та тестувальникам у компаніях, адже він чітко показує дані використання системи та заздалегідь допоможе зробити так, аби ніякий код не мав помилок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] B. Beizer, "Software Testing Techniques." Van Nostrand Reinhold, 1990

[2] L. Williams, "Performance Testing Guidance for Web Applications." Microsoft Corporation, 2007.

[3] L. Williams, "Performance Testing Guidance for Web Applications." Microsoft Corporation, 2007.

[4] OWASP, "OWASP Testing Guide." Retrieved from https://owasp.org/www-project-web-security-testing-guide/

[5] OWASP, "OWASP Testing Guide." Retrieved from https://owasp.org/www-project-web-security-testing-guide/

[6] R. Patton, "Software Testing." Sams Publishing, 2005.

[7] R. Patton, "Software Testing." Sams Publishing, 2005.

[8] R. Patton, "Software Testing." Sams Publishing, 2005. [5] B. Beizer, "Software Testing Techniques." Van Nostrand Reinhold, 1990.

[9] S. Myers, "The Art of Software Testing." John Wiley & Sons, 2012.

[10] S. Myers, "The Art of Software Testing." John Wiley & Sons, 2012.

[11] Smith, J. (2022). Software Analysis and Design: Concepts, Techniques, and Applications. New York: Wiley.

[12] Базовий, І. І. (2018). Програмні засоби підтримки системного аналізу: навч. посіб. Луцьк: Волин. нац. ун-т ім. Лесі Українки.

[13] The .NET Compiler Platform SDK (Roslyn) from https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

ДОДАТОК А

```
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Emit;
using System.Reflection;
using System.Diagnostics;

namespace ConsoleApp3
{
    internal class Program
    {
        static void Main(string[] args)
        {
            var syntaxTree = CSharpSyntaxTree.ParseText(@"
using System;
using System.IO;

namespace CompileSample
{
    public class Process
    {
        public void Execute()
```

```
        {
            string[] lines = new string[1000000];
            for (int i = 0; i < 1000000; i++)
                lines[i] = i.ToString();

            File.WriteAllLines(""c:\\test.txt"", lines);
        }
    }
}");

            var assemblyName = Path.GetRandomFileName();
            var references = new MetadataReference[]
            {
                MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
                MetadataReference.CreateFromFile(typeof(Enumerable).Assembly.Location)
            };

            var compilation = CSharpCompilation.Create(
                assemblyName,
                syntaxTrees: new[] { syntaxTree },
                references: references,
                options: new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary)
```

```
);

using var ms = new MemoryStream();

var result = compilation.Emit(ms);

if (!result.Success)
{
    IEnumerable<Diagnostic> failures = result.Diagnostics.Where(diagnostic =>
        diagnostic.IsWarningAsError ||
        diagnostic.Severity == DiagnosticSeverity.Error);

    foreach (Diagnostic diagnostic in failures)
    {
        Console.Error.WriteLine("{0}: {1}", diagnostic.Id, diagnostic.GetMessage());
    }
}
else
{
    ms.Seek(0, SeekOrigin.Begin);
    var assembly = Assembly.Load(ms.ToArray());
    var type = assembly.GetType("CompileSample.Process");
```

```
            var obj = Activator.CreateInstance(type);

            var cpu = new PerformanceCounter("Processor Information", "% Processor Time",
"_Total");
            var memory = new PerformanceCounter("Memory", "% Committed Bytes In Use");

            var cpuBefore = cpu.NextValue();
            var memoryBefore = memory.NextValue();
            var time = Stopwatch.StartNew();

            type.InvokeMember("Execute",
               BindingFlags.Default | BindingFlags.InvokeMethod,
               null,
               obj,
               null);

            Console.WriteLine($"CPU cycles usage - from {cpuBefore} to
{cpu.NextValue()}");
            Console.WriteLine($"Memory usage - {memory.NextValue() - memoryBefore}
Bytes");
            time.Stop();
            Console.WriteLine("Execution time - " + time.Elapsed);
        }
```

```
        }

    }

}
```