МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

КОМПОНЕНТ ДЛЯ АВТОМАТИЧНОГО РЕФАКТОРИНГУ КОДУ В VISUAL STUDIO

Рівень бакалавр
Галузь знань 12 Інформаційні технології,
спеціальність 121 Інженерія програмного забезпечення
Виконав здобувач 4 курсу
Новоставський Григорій Сергійович

Керівник: к.т.н., доцент
Чикунов Павло Олександрович

Рецензент: к.т.н., доцент
Трофименко Олена Григорівна

м. Одеса – 2024 р.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ
Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційних технологій
Галузь знань: 12 Інформаційні технології
Спеціальність: 121 Інженерія програмного забезпечення
Назва освітньої програми: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Н.І. Логінова
_____ «____»_____ 20__ року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Новоставському Григорію Сергійовичу

1. Тема роботи: Компонент для автоматичного рефакторингу коду в Visual Studio
   Керівник роботи: к.т.н., доцент Чикунов Павло Олександрович
   затверджені наказом НУ «ОЮА» № 809-06 від «02» квітня 2024 року
2. Дата видачі завдання «15» вересня 2023 року.
3. Строк подання здобувачем роботи «20» травня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз засобів автоматизації рефакторингу	10.10.2003	
2	Моделювання компонента автоматичного рефакторингу	01.12.2024	
3	Програмна реалізація компонента автоматичного рефакторингу	12.02.2024	
4	Оцінка ефективності компонента автоматичного рефакторингу	27.03.2024	
5	Оформлення звітної частини	19.05.2024	

Здобувач _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Новоставський Г. С. Компонент для автоматичного рефакторингу коду в Visual Studio. – Кваліфікаційна робота бакалавра за спеціальністю 121 «Інженерія програмного забезпечення», освітньою програмою «Інженерія програмного забезпечення».

Кваліфікаційна робота викладена на 53 сторінках основного тексту і 58 сторінках загального тексту, містить 3 розділи, 17 рисунків, 1 додаток, 37 використаних джерел.

Метою роботи є моделювання, програмна реалізація та оцінка ефективності компонента для автоматизованого рефакторингу коду на мові програмування C# у IDE Visual Studio.

Об'єктом дослідження є процес автоматичного рефакторингу коду C# у середовищі розробки Visual Studio. Цей процес вибирається для дослідження через його значущість у покращенні якості програмного забезпечення та полегшенні процесу його розробки. Об'єкт дослідження охоплює аспекти автоматизації та оптимізації рефакторингу коду в зазначеному середовищі.

Предметом дослідження є техніка розробки та використання компонентів користувача для автоматичного рефакторингу коду на мові програмування C# в середовищі Visual Studio. Предмет дослідження складається з інструментів, які використовуються для аналізу та покращення якості коду.

У кваліфікаційній роботі розроблено компонент для автоматичного рефакторингу. Проведено аналіз наявних методів та інструментів для автоматичного рефакторингу, проведено моделювання компонента, розроблено алгоритми для виявлення та усунення недоліків у коді, протестовано розроблений компонент на прикладах брудного коду.

Ключові слова: рефакторинг, автоматизація, діаграма, блок-схема, компонент, Visual Studio, Roslyn, мова C#, застосунок.

ABSTRACT

Novostavsky G. S. Component for automatic code refactoring in Visual Studio - Bachelor's thesis in specialty 121 "Software Engineering", educational program "Software Engineering".

The qualification work is set out on 53 pages of the main text and 58 pages of the general text, contains 3 sections, 17 figures, 1 appendix, 37 references.

The aim of this work is to model, implement, and evaluate the effectiveness of a component for automated code refactoring in the C# within the IDE Visual Studio.

The object of the research is the process of automatic C# code refactoring in the Visual Studio development environment. This process is chosen for study due to its significance in improving software quality and facilitating the development process. The object of the research encompasses aspects of automation and optimization of code refactoring in the specified environment.

The subject of the research is the technique of developing and using user components for automatic code refactoring in the C# programming language within the Visual Studio environment. The subject of the research consists of tools used for analyzing and improving code quality.

In the qualification work, a component for automatic refactoring has been developed. An analysis of existing methods and tools for automatic refactoring has been conducted, the component has been modeled, algorithms for detecting and eliminating code deficiencies have been developed, and the developed component has been tested on examples of messy code.

Keywords: refactoring, automation, diagram, flowchart, component, Visual Studio, Roslyn, C# language, application.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

API – Application programming interface

AST – Abstract Syntax Tree

IDE – Integrated Development Environment

LINQ – Language Integrated Query

SDK – Software Development Kit

VSIX – Visual Studio Extension

ВСТУП

Постійний розвиток програмного забезпечення та зростання вимог до його якості та функціональності ставлять перед розробниками завдання забезпечити ефективний та надійний процес розробки. Однак, збільшення розміру проєктів та складності коду призводить до збільшення кількості потенційних помилок та важкості його обслуговування.

Автоматичний рефакторинг коду стає важливим інструментом для підтримки якості та розширення функціональності програмного забезпечення. Він дозволяє автоматизувати процеси виявлення та усунення недоліків у коді, що дозволяє розробникам зосередитися на важливих завданнях та підвищує швидкість розробки.

Актуальність роботи обумовлена сучасними тенденціями у сфері розробки програмного забезпечення, які вимагають постійного підвищення якості коду та його функціональності. В умовах стрімкого зростання складності програмних систем та обсягів коду, збільшується ймовірність появи помилок, а також витрати часу та ресурсів на їхнє виявлення та виправлення.

У контексті Visual Studio, що є одним з найпопулярніших середовищ розробки, є потреба у засобах для автоматичного рефакторингу коду. Інтеграція такого компонента в середовище розробки може значно полегшити роботу розробників та покращити якість та чистоту коду.

Метою роботи є розробка компонента для автоматизованого рефакторингу коду на мові програмування C# у середовищі Visual Studio, а також оцінка його ефективності.

Основними **завданнями** роботи є:

– аналіз наявних методів та інструментів для автоматичного рефакторингу коду;

– моделювання компонента автоматичного рефакторингу;

– реалізація компонента на мові програмування C# з використанням Visual Studio SDK та Roslyn API;

– тестування розробленого компонента на брудному коді;

– аналіз результатів та формулювання висновків щодо ефективності та практичного значення розробленого компонента.

Об'єктом дослідження є процес автоматичного рефакторингу коду C# у середовищі розробки Visual Studio. Цей процес вибирається для дослідження через його значущість у покращенні якості програмного забезпечення та полегшенні процесу його розробки. Об'єкт дослідження охоплює аспекти автоматизації та оптимізації рефакторингу коду в зазначеному середовищі.

Предметом дослідження є техніка розробки та використання компонентів користувача для автоматичного рефакторингу коду на мові програмування C# в середовищі Visual Studio. Предмет дослідження складається з інструментів, які використовуються для аналізу та покращення якості коду.

Розроблений компонент для автоматичного рефакторингу коду в середовищі Visual Studio має таке практичне значення:

– підвищення продуктивності розробки: застосування розробленого компонента дозволить швидше та ефективніше виправляти та оптимізувати свій код, що призведе до підвищення продуктивності розробки програмного забезпечення;

– покращення якості коду: автоматичний рефакторинг допоможе знизити кількість помилок у коді, покращити його читабельність та підтримуваність, що сприятиме створенню високоякісного програмного забезпечення;

– економія часу та ресурсів: використання автоматичного рефакторингу коду зменшить необхідність ручних втручань та витрат часу на виправлення помилок, що дозволить сконцентруватися на більш важливих завданнях.

1 АНАЛІЗ ЗАСОБІВ АВТОМАТИЗАЦІЇ РЕФАКТОРИНГУ

Рефакторинг – це процес зміни коду з метою зробити його більш зрозумілим, чистим та зручним для розробників проєкту. Рефакторинг покращує структуру проєкту без змін у функціоналі програми, він дуже потрібен в усіх проєктах бо цей процес покращує читабельність коду для усіх учасників проєкту [1]. Якщо на проєкті з’явиться новий розробник, то він зможе швидко зрозуміти що відбувається у коді. Чим зручніше код тим краще, бо знижується час його обслуговування.

Рефакторинг може покращити розробку програмного забезпечення таким чином.

Покращити якість коду: рефакторинг допомагає зробити код більш читабельним, зрозумілим і зручним для підтримки. Це зменшує ймовірність помилок і полегшує розуміння коду.

Зменшення технічного боргу: рефакторинг може зменшити технічний борг за рахунок виправлення дефектів у коді. Це означає, що розробники не будуть накопичувати велику кількість технічного боргу, який може призвести до проблем у майбутньому.

Підвищення продуктивності розробки: чистий і організований код полегшує роботу розробників і дозволяє їм швидше розуміти, змінювати і розширювати функціональність програмного забезпечення.

Підтримка та рефакторинг на віддалених етапах розробки: рефакторинг можна виконувати на будь-якому етапі розробки програмного забезпечення, в тому числі на віддалених етапах життєвого циклу проєкту. Це гарантує актуальність коду та зменшує ризик внесення змін до великих, складних проєктів.

Автоматичний рефакторинг коду перевершує ручний у своїй ефективності та швидкості, пропонуючи широкий спектр операцій без необхідності витрачати час на механічні процеси. Він також сприяє стандартизації коду, мінімізації ризиків помилок та забезпечує можливість масштабування для великих проєктів, що забезпечує покращення стабільності програмного забезпечення.

За той час, що розробники б витратили на ручний рефакторинг, автоматичні інструменти можуть здійснити значно більше змін у коді, забезпечуючи відповідність стандартам, виправляючи стилістичні помилки, оптимізуючи структуру та забезпечуючи однорідність у всьому проєкті. Це звільняє час розробників для більш важливих завдань, таких як вдосконалення функціональності програми та вирішення складних проблем.

Заощаджує час і ресурси команди розробників, оскільки він дозволяє швидко та ефективно вносити зміни в код, що в свою чергу збільшує продуктивність і прискорює розвиток програмного забезпечення.

Існують автоматизовані інструменти рефакторингу в Visual Studio, кожен з яких має свої унікальні функції та можливості, що дозволяють ефективно працювати з кодом і підтримувати його в оновленому та оптимізованому стані.

1.1 Аналіз наявних інструментів автоматичного рефакторингу коду

Популярні інструменти, подібні до ReSharper та CodeRush, відкривають перед розробниками широкі можливості для автоматизації процесу рефакторингу коду. Ці інструменти не лише допомагають у перейменуванні змінних, видаленні дублікатів коду та оптимізації структури програми, а й забезпечують ряд інших корисних функцій. Серед них можна відзначити автоматичне виявлення та виправлення потенційних проблем в коді, підказки щодо вдосконалення стилю програмування та навіть автоматичну генерацію коду за шаблонами.

Використання цих інструментів дозволяє зосередитися на написанні нового коду та вирішенні функціональних проблем, мінімізуючи витрати часу на рутинні задачі з підтримки кодової бази.

1.1.1 ReSharper

ReSharper (R#) – це потужний плагін для Microsoft Visual Studio, який значно підвищує продуктивність роботи розробників. Він пропонує широкий спектр функцій, які допомагають автоматизувати рутинні завдання, покращують код та

підвищують загальну продуктивність роботи [2]. ReSharper пропонує розробникам такі можливості, як автоматичне виявлення та виправлення помилок, оптимізація швидкодії, а також виконання різноманітних операцій рефакторингу. Він включає в себе інтелектуальні інструменти аналізу коду, розуміння мови програмування та підтримку різних технологій, що дозволяють швидко та ефективно працювати над проєктами будь-якої складності.

Крім того, ReSharper підтримує різні мови програмування, як-от C#, VB.NET, JavaScript, HTML, CSS та інші, що робить його універсальним інструментом для розробки різноманітних типів програмного забезпечення. Завдяки своїм великим функціоналом та зручним інтерфейсом, ReSharper став незамінним помічником для багатьох розробників .NET, допомагаючи їм зосередитися на написанні високоякісного та ефективного коду.

Розглянемо переваги від використання ReSharper [3]:

– *Автоматичний рефакторинг*. ReSharper надає широкий набір інструментів для автоматичного виявлення та виправлення помилок, оптимізації швидкодії та виконання різноманітних операцій рефакторингу, що значно полегшує процес розробки.

– *Широкі можливості конфігурації*. ReSharper надає розгалужені налаштування, які дозволяють користувачам налаштувати його функціональність під свої потреби та вимоги проєкту.

– *Покращена якість коду*. Використання ReSharper сприяє покращенню якості коду завдяки вбудованим інструментам аналізу, виявленню помилок та оптимізації.

– *Підвищення продуктивності*. Інтеграція ReSharper з Visual Studio дозволяє розробникам працювати ефективніше за рахунок швидкого доступу до функцій, підказок та інструментів.

– *Навігація по коду*. ReSharper надає розробникам швидку та зручну навігацію по коду, що допомагає ефективно орієнтуватися в проєкті та швидко знаходити необхідні фрагменти коду.

– *Удосконалене налагодження*. Інструменти для налагодження в ReSharper дозволяють ефективно виявляти та усувати помилки в коді, що полегшує процес налагодження та забезпечує високу якість програмного забезпечення.

– *Регулярні оновлення та покращення*. Команда розробників JetBrains постійно працює над вдосконаленням та вдосконаленням ReSharper, надаючи користувачам регулярні оновлення з новими функціями та покращеннями, що дозволяє розробникам бути в курсі нових технологій та трендів у розробці програмного забезпечення.

Недоліки використання ReSharper:

– *Велике споживання ресурсів*. ReSharper може значно збільшити споживання ресурсів (пам'яті та процесорного часу) в середовищі Visual Studio, що може призвести до зниження продуктивності роботи.

– *Вартість та ліцензування*. ReSharper є платним програмним забезпеченням, і його вартість може бути значною, особливо для комерційних проєктів. Крім того, потрібна окрема ліцензія на кожного користувача.

– *Навчання та адаптація*. Для повного використання всіх можливостей ReSharper може знадобитися час на оволодіння всіма його функціями та налаштуваннями, що може вимагати додаткового навчання та практики.

ReSharper є платним продуктом, що може бути обмеженням для тих, хто шукає безкоштовні альтернативи. Деякі користувачі також відзначають, що деякі функції ReSharper можуть бути надто інтенсивними або неінтуїтивними у використанні, що може призвести до додаткового часу на вивчення та освоєння інструменту.

1.1.2 CodeRush

CodeRush – це інноваційний інструмент для розробки програмного забезпечення, що забезпечує розробникам широкий набір функціональних можливостей для підвищення продуктивності та якості коду [4]. Вироблений

компанією DevExpress, CodeRush надає потужні інструменти для автоматизації рутинних задач.

Однією з ключових особливостей CodeRush є його широкий спектр функцій рефакторингу. Від перейменування змінних до видалення зайвого коду, цей інструмент надає розробникам зручні та ефективні інструменти для покращення структури та читабельності коду. Крім того, CodeRush допомагає автоматизувати та спрощувати багато інших аспектів розробки, забезпечуючи швидкий доступ до часто використовуваних функцій, автоматичної розробки шаблонів коду та багато іншого.

Переваги використання CodeRush [5].

– *Підвищення продуктивності*. CodeRush пропонує широкий спектр функцій, які роблять розробку програмного забезпечення простою та швидкою.

– *Покращена якість коду*. Функції рефакторингу CodeRush дозволяють розробникам покращити структуру та читабельність свого коду.

– *Багатофункціональність*. CodeRush пропонує безліч корисних інструментів, таких як автоматичне виправлення помилок, аналіз коду для оптимізації та підвищення продуктивності, а також інтеграцію зі сторонніми бібліотеками.

Недоліки використання CodeRush.

– *Вивчення інструменту*. Використання всього потенціалу CodeRush може вимагати часу на вивчення його функціоналу та навичок. Знадобитися деякий час, щоб оволодіти всіма можливостями інструменту.

– *Велике споживання ресурсів*. CodeRush може вимагати значних ресурсів системи, що може вплинути на продуктивність роботи на менш потужних комп'ютерах або великих проєктах.

– *Платна версія*. Інструмент доступний за плату, що може стати перешкодою для тих, хто шукає безкоштовні альтернативи.

Отже, CodeRush як і ReSharper має значні недоліки. Обидва інструменти потребують значні ресурси системи, та вони платні.

Також, в Visual Studio є вбудовані інструменти рефакторингу – Visual Studio Code Refactoring Tools. Ці інструменти забезпечують розширені можливості для покращення коду безпосередньо у редакторі. Вони надають можливість проводити різноманітні операції з кодом, що сприяє поліпшенню його структури, читабельності та підтримки.

Завдяки вбудованим інструментам рефакторингу в Visual Studio, можна виконувати такі операції [6]:

– вилучення коду: видалення непотрібних або зайвих фрагментів коду, що дозволяє покращити його якість та структуру;

– перейменування: зміна імен змінних, методів, класів та інших елементів коду по всьому проєкту з автоматичним оновленням посилань на них;

– вилучення методів та функцій: видалення непотрібних або зайвих методів та функцій з коду для зменшення його складності та покращення структури;

– оптимізація імпортів: автоматичне оптимізування імпортів у коді, видалення зайвих або непотрібних імпортів та додавання потрібних;

– перенесення коду: перенесення фрагментів коду між файлами та класами з автоматичним оновленням посилань на них;

– рефакторинг узагальнених виразів: покращення читабельності та структури коду шляхом рефакторингу узагальнених виразів та конструкцій.

Загалом, Visual Studio Code Refactoring Tools як вбудований інструментарій має великий набір інструментів. Але він обмежений, для розробки спеціалізованих операцій рефакторингу потрібні додаткові інструменти або розширення, які надають додаткові функції та можливості.

1.2 Засоби розробки компонента

Для розробки компонента для автоматичного рефакторингу коду в Visual Studio я скористався такими засобами:

– Мова C# для написання логіки мого компонента.

– Visual Studio Community 2022 як середовище розробки.

– Visual Studio SDK. Набір інструментів, який надає всі необхідні засоби для розробки розширень для Visual Studio.

– Roslyn API. Платформа для аналізу коду на мові C# та Visual Basic.

– Ліцензійну платформу Windows 10.

Visual Studio – це інтегроване середовище розробки (IDE) від Microsoft, призначене для розробки різноманітних програмних продуктів, від мобільних застосунків до веб-сервісів та корпоративних застосунків [7]. Завдяки своїм потужним інструментам та широкому спектру функцій, Visual Studio стала важливим інструментом для розробників у всьому світі.

Однією з ключових особливостей Visual Studio є підтримка багатьох мов програмування, включаючи C#, Visual Basic, C++, Python, JavaScript та інших. Це дає можливість працювати з будь-якою мовою і створювати різноманітні застосунки та сервіси.

Крім того, Visual Studio має широкий набір інструментів для розробки, включаючи редактор коду з підсвічуванням синтаксису, автодоповненням коду та вбудованою підтримкою Git для контролю версій. Вона також має вбудовані шаблони проєктів для різних типів програм, що дозволяє швидко створювати нові проєкти без необхідності вручну налаштовувати їх структуру.

Розширення Visual Studio – це додаткові компоненти, які розширюють функціональність стандартного інтегрованого середовища розробки (IDE) Visual Studio. Вони надають можливість налаштовувати та розширювати робоче середовище згідно потреб і завдань. Розширення можуть додавати нові функції, інтегрувати сторонні інструменти, поліпшувати робочий процес та забезпечувати підтримку для різних типів розробки, включаючи веб-розробку, мобільну розробку, розробку для хмарних платформ, ігрову розробку та інші. Вони можуть надавати спеціалізовані інструменти та функціональність для кожного типу розробки.

Мною було використано Visual Studio як основне середовище розробки та тестування компонента для автоматичного рефакторингу коду. Це середовище

надає широкий набір інструментів і можливостей, які допомагають у всіх етапах процесу розробки.

1.2.1 SDK Visual Studio

Мною було встановлено Visual Studio SDK, щоб мати доступ до всіх необхідних інструментів та ресурсів для розробки розширень та інтегрованих застосунків для Visual Studio.

Visual Studio SDK – це набір інструментів, документації та ресурсів, який надається Microsoft для розробки розширень та інтегрованих застосунків для Visual Studio. Цей SDK дозволяє розширювати функціональність та можливості Visual Studio, створюючи власні застосунки, які інтегруються безпосередньо в це робоче середовище розробки. Ось кілька ключових аспектів Visual Studio SDK [8]:

– Інструменти розробки: Visual Studio SDK надає різні інструменти для розробки розширень, включаючи шаблони проєктів, дизайнери інтерфейсу користувача та набори бібліотек для доступу до API Visual Studio. Ці інструменти допомагають швидко почати створювати розширення та застосунки.

– Документація та приклади: Visual Studio SDK містить обширну документацію та приклади коду, які допомагають зрозуміти основи розробки розширень для Visual Studio. Ця документація включає опис API, приклади використання та рекомендації щодо найкращих практик.

– Налагодження та тестування: Visual Studio SDK надає інструменти для налагодження та тестування розширень. Можна використовувати вбудовані інструменти для запуску й налагодження своїх розширень напряму з Visual Studio, що дозволяє ефективно тестувати та виправляти помилки.

– Visual Studio SDK підтримує різні мови програмування, включаючи C#, Visual Basic та C++.

Для розповсюдження та встановлення розширень для Visual Studio використовуються формат файлу VSIX.

VSIX – це архівований файл, який містить всі необхідні компоненти, ресурси та метадані для розширення. Цей файл може містити файли коду, бібліотеки, зображення, ресурси інтерфейсу користувача та інше.

Файли VSIX дозволяють збілдити свої розширення у вигляді одного компактного файлу для подальшого розповсюдження. Це спрощує процес дистрибуції розширень серед користувачів Visual Studio.

Розширення у форматі VSIX можна встановлювати у Visual Studio просто за допомогою двох кліків миші або шляхом перетягування файлу VSIX у вікно Visual Studio. Цей простий процес робить встановлення розширень легким та зручним для користувачів.

1.2.2 Roslyn

Roslyn – це відкритий вихідний код компілятора та інструментів аналізу вихідного коду для мов програмування C# та Visual Basic .NET від Microsoft. Ця технологія була вперше представлена у версії Visual Studio 2015 і з тих пір є основою для розробки програмного забезпечення на платформі .NET. Roslyn надає програмістам можливість аналізувати та модифікувати вихідний код на льоту, що відкриває широкі можливості розробки різноманітних інструментів та розширень для розробки програмного забезпечення.

Однією з переваг використання Roslyn для аналізу коду є те, що ці засоби використовують менше ресурсів пам'яті при роботі з Visual Studio, ніж інші рішення, що вимагають розробки власної бази коду для аналізу. Це дозволяє зосередитися на розробці інструментів та функціональності, не турбуючись про велике споживання ресурсів.

При дослідженні Roslyn API вивчається ряд ключових можливостей [9]:

– Аналіз Синтаксичного Дерева: Roslyn надає можливість отримати синтаксичне дерево програми, яке може бути використане для аналізу структури та складових коду, таких як класи, методи, змінні тощо.

– Аналіз Семантичної Моделі: Розуміння значення коду – типів, символів, методів та їх взаємозв'язків. Це дозволяє виконувати більш складний аналіз, такий як перевірка типів та вирішення залежностей.

– Модифікація Коду: Здатність вносити зміни в джерельний код програми, як-от додавання, видалення або зміна класів, методів та властивостей.

– Розробка Нового Коду: Можливість створювати нові фрагменти коду програми або цілі програми з нуля за допомогою Roslyn.

– Аналіз та Модифікація Застосунків: Вивчення та розробка інструментів, які використовують Roslyn для виявлення проблем в коді, автоматичного рефакторингу, виправлення помилок та аналізу використання коду.

– Інтеграція з IDE та Іншими Інструментами: Розробка розширень та інструментів, які можуть використовувати Roslyn для покращення робочого процесу програміста, забезпечення підказок, підсвічування синтаксису, автоматичного завершення коду тощо.

Аналіз Синтаксичного Дерева (AST) – це процес перетворення вихідного коду програми в структуровану ієрархічну модель, яка представляє синтаксичну структуру програми. AST є важливою складовою компіляторів та інструментів аналізу коду, оскільки він дозволяє отримувати інформацію про структуру та компоненти програми, щоб здійснювати різноманітні операції, як-от аналіз, перетворення та модифікація коду [10].

AST має вузли, як окремі конструкції мови програмування: оператори, вирази, змінні, методи, а також відносини між ними. Кожен вузол AST містить інформацію про тип, ідентифікатор, значення та інші атрибути, які визначають його роль та місце в програмі.

Процес аналізу синтаксичного дерева включає такі кроки [11]:

– Синтаксичний Розбір: Вихідний код програми зазвичай спочатку проходить через процес синтаксичного розбору, під час якого він перетворюється в деревоподібну структуру, яка відображає синтаксичну структуру програми.

– Побудова Синтаксичного Дерева: На основі результатів синтаксичного розбору будується синтаксичне дерево, що представляє програму у вигляді ієрархічної структури вузлів.

– Аналіз та Обробка Дерева: Після побудови синтаксичного дерева може проводитися аналіз та обробка його вузлів для виконання різних завдань, таких як пошук певних конструкцій, перевірка правил кодування та виявлення помилок.

Аналіз синтаксичного дерева дозволяє отримувати інсайти щодо структури та функціональності програми та використовувати цю інформацію для розробки різних інструментів, таких як рефакторингові або аналізатори коду.

На рис. 1.1 продемонстровано вигляд синтаксичного дерева:

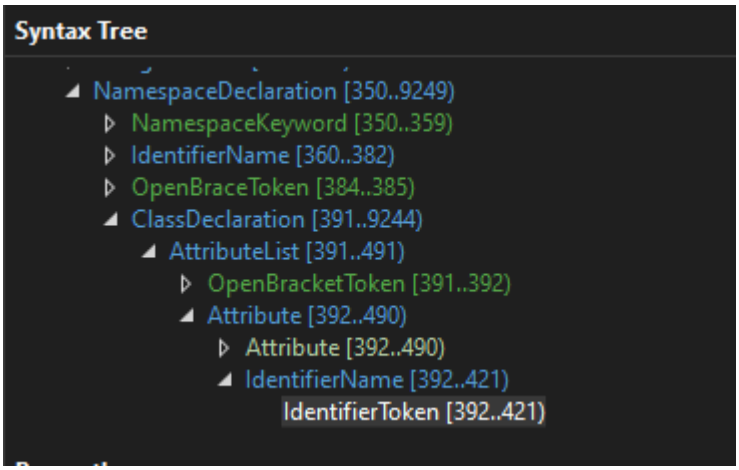

Рисунок 1.1 – Синтаксичне дерево

Аналіз семантичної моделі – це процес вивчення семантики або значення програми, що базується на її структурі та контексті використання. Цей процес зазвичай включає в себе розуміння типів даних, взаємозв'язків між об'єктами, визначення функцій та їх параметрів, а також визначення правил інтерпретації виразів та операторів [12]. Основні аспекти аналізу семантичної моделі включають [13]:

– Вирішення Імен: Це процес встановлення відповідності між іменами, що використовуються у програмі, та їх визначеннями. Він вирішує питання, якій змінній або функції посилається кожне використання певного імені.

– Виведення Типів: Виведення типів встановлює типи даних для змінних, виразів та інших конструкцій програми, які не були явно вказані. Це дозволяє визначити правильний тип для виразів та запобігає помилкам типізації.

– Аналіз Виразів: Цей аспект включає в себе розуміння значення виразів та їх взаємозв'язку з операторами та операндами. Він допомагає визначити, як вирази обчислюються та як вони впливають на стан програми.

– Перевірка Типів: Перевірка типів перевіряє відповідність типів у різних частинах програми та виявляє можливі помилки типізації, як-от невідповідність типів у виразах або неправильне використання операторів.

– Виявлення Помилок та Потенційних Проблем: Аналіз семантичної моделі також може допомогти виявити різні помилки та потенційні проблеми у програмі, як-от неправильне використання змінних, недостатньо чітке визначення функцій або інші аспекти, які можуть впливати на коректність та ефективність програми.

Аналіз семантичної моделі є важливим етапом у процесі розробки програмного забезпечення, оскільки він дозволяє отримати глибоке розуміння програми та виявити потенційні проблеми до їх виявлення під час виконання програми.

1.2.3 Платформа розробки

Обираючи операційну систему для використання, я вирішив скористатися Windows 10 з кількох причин. Вона пропонує широкий спектр функціональності, включаючи покращену безпеку, продуктивність та інтеграцію з різними пристроями.

Це забезпечує широку сумісність і підтримку для різних програм і інструментів, таких як Visual Studio, яка використовується для розробки компонента для автоматичного рефакторингу коду.

З приводу сумісності компонента з іншими операційними системами, розроблений компонент може запускатися на будь-якій операційній системі, де може бути встановлено Visual Studio.

Хоч Windows 10 не є найновішою операційною системою, вона залишається однією з найбільш популярних і підтримуваних версій, що забезпечує стабільність та сумісність для розробки програмного забезпечення.

1.2.4 Мова розробки

Вибираючи мову програмування для написання логіки мого компонента для автоматичного рефакторингу коду в Visual Studio, я вирішив скористатися мовою програмування C#. Це рішення базується на кількох важливих факторах, які роблять C# привабливим вибором для розробки програмного забезпечення [14]:

– Синтаксична зручність: C# має чистий і лаконічний синтаксис, який спрощує читання і розуміння коду. Це дозволяє швидше і ефективніше розробляти та підтримувати код.

– Широкий спектр функцій: C# є мовою програмування, що базується на платформі .NET, яка надає доступ до великого набору бібліотек і фреймворків для розробки різноманітних застосунків. Це дозволяє використовувати різноманітні ресурси для реалізації функціональності мого компонента.

– Інтеграція з Visual Studio: C# є основною мовою програмування для розробки розширень та компонентів для Visual Studio. Вона надає доступ до широкого спектру API і інструментів, які допомагають в розробці інтегрованих засобів для роботи з кодом.

– Підтримка розробників: C# має велику та активну спільноту розробників, що робить процес вивчення, підтримки та розвитку проєкту більш простим і доступним.

1.3 Висновки до розділу

Аналіз наявних інструментів рефакторингу дозволив визначити їх недоліки. На основі цього аналізу можна зробити висновки щодо їх ефективності.

Недоліками наявних інструментів є їх велике споживання ресурсів, що може вплинути на продуктивність та швидкість роботи. Крім того, два з трьох інструментів є комерційними продуктами, що може створювати обмеження для користувачів з обмеженим бюджетом або для команд розробки, які шукають безкоштовні або відкриті альтернативи.

Аналіз показав, що на ринку не вистачає безкоштовних та бюджетних по споживанню ресурсів альтернатив існуючим комерційним інструментам. Таким чином, існує потреба у розробці нового компонента, який б забезпечував функціональність без значного впливу на продуктивність.

Visual Studio незамінний інструмент для розробки розширень та компонентів, завдяки своїм потужним можливостям та широкому спектру інструментів. Використання Visual Studio SDK та Roslyn API розширить функціональність середовища розробки та допоможе реалізувати автоматичний рефакторинг коду.

Вибір операційної системи Windows 10 та мови програмування C# був обґрунтований їхньою широкою популярністю, підтримкою відповідних інструментів розробки та зручністю використання для цілей даного проєкту.

2 МОДЕЛЮВАННЯ КОМПОНЕНТА АВТОМАТИЧНОГО РЕФАКТОРИНГУ

Під час моделювання компонента для автоматичного рефакторингу коду необхідно враховувати різноманітні аспекти, що включають в себе не лише структуру та взаємозв'язки між компонентами, але й їх функціональні можливості та ролі в системі.

З метою забезпечення ефективного та зрозумілого процесу рефакторингу, також важливо ретельно проробити блок-схеми рефакторингових операцій. Це дозволить отримати повний огляд моделювання та функціонування компонента.

2.1 Розробка діаграм автоматичного рефакторингу

Розробка діаграм є важливою складовою процесу моделювання програмного забезпечення.

Візуалізація архітектури: Діаграми об'єктів, компонентів та класів надають візуальне відображення архітектурної структури системи. Це допомагає краще розуміти, як система організована та які компоненти взаємодіють між собою.

Аналіз та вдосконалення архітектури: Діаграми дозволяють виявити можливі проблеми або недоліки в архітектурній системи, як-от зайві залежності, недостатня модульність або низька зрозумілість. Це дає можливість вдосконалити архітектуру для забезпечення кращої продуктивності, підтримки та розширюваності системи.

Керування складністю: Діаграми допомагають управляти складністю системи, розділяючи її на менші компоненти та класи. Це сприяє полегшенню розуміння та підтримки кодової бази, а також сприяє уникненню зайвої складності.

Діаграма об'єктів (в UML – Object diagram) є важливим інструментом в архітектурному проєктуванні програмного забезпечення, відображає об'єкти та їх зв'язки в певний момент часу.

Моделювання об'єктів та їх взаємодій: Діаграма об'єктів дозволяє візуалізувати об'єкти, які складають систему, і показати, як вони взаємодіють між собою. Це допомагає зрозуміти структуру програми і виявити потенційні проблеми або недоліки в архітектурі. Діаграма об'єктів дозволяє аналізувати взаємодії між об'єктами, включаючи передачу даних та виклик методів. Це допомагає виявити можливі проблеми з інтерфейсами або залежностями [15].

Діаграма компонентів (Component diagram) відображає взаємозв'язки та залежності між компонентами системи програмного забезпечення. Діаграма компонентів дозволяє структурувати систему на окремі компоненти, які можна розглядати як самостійні частини системи з власним функціоналом і відповідальністю. Діаграма компонентів також сприяє модульності вашої системи, розділяючи її на окремі компоненти з чітко визначеними інтерфейсами. Це допомагає уникнути циклічних залежностей і забезпечити легку підтримку та розширення системи.

Діаграма компонентів дозволяє чітко визначити інтерфейси між компонентами, що сприяє розподілу відповідальності та забезпечує можливість заміни або розширення компонентів без впливу на інші частини системи [16].

На рис. 2.1 продемонстровано вигляд діаграми компонентів. Ця діаграма компонентів показує:

– компонент MagicNumberRefactoringProvider, який унаслідується від CodeRefactoringProvider та розширює його функціональність, додаючи конкретні методи для різних типів рефакторингів;

– компонент MagicNumberRefactoringProvider використовує Document для отримання синтаксичного дерева коду;

– компонент Document надає семантичну модель через SemanticModel, який потім використовується для аналізу коду;

– вся система функціонує всередині середовища розробки Visual Studio, де розширення автоматично запускає рефакторинг коду.

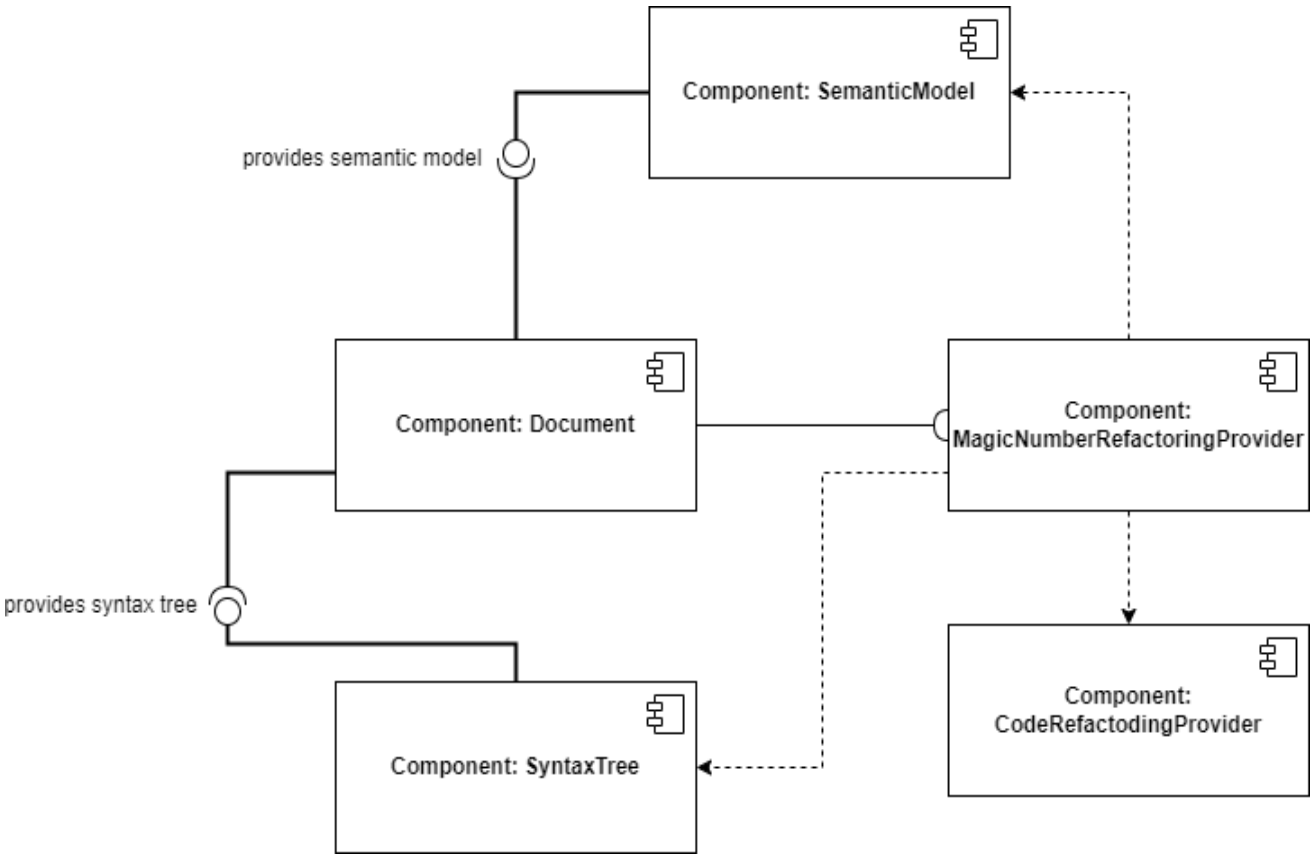

Рисунок 2.1 – Діаграма компонентів

UML-діаграма класів є важливим інструментом для візуалізації структури системи програмного забезпечення на рівні класів і їх взаємозв'язків. Діаграма класів дозволяє графічно представити всі класи системи та їх взаємозв'язки. Це полегшує розуміння структури системи та виявлення взаємодії між класами. Діаграма класів допомагає аналізувати співвідношення між класами, такі як спадкування, агрегація, композиція та асоціація. Це дозволяє краще розуміти структуру системи та виявляти можливі проблеми або покращення [17].

На рис. 2.2 продемонстровано вигляд UML-діаграми класів, що побудована з використанням мови розмітки plantUML у сервісі Draw.io:

– головний клас MagicNumberRefactoringProvider, який представляє компонент для автоматичного рефакторингу коду в середовищі Visual Studio; він має один метод ComputeRefactoringsAsync, який відповідає за обчислення рефакторингів;

– асинхронні внутрішні методи HandleSelfEncapsulateFieldRefactoringAsync, HandleConsolidateConditionalExpressionsAsync, HandleRemoveSetterAsync і HandleHideMethodRefactoringAsyn представляють окремі завдання рефакторингу коду, що можуть бути виконані в межах компонента.

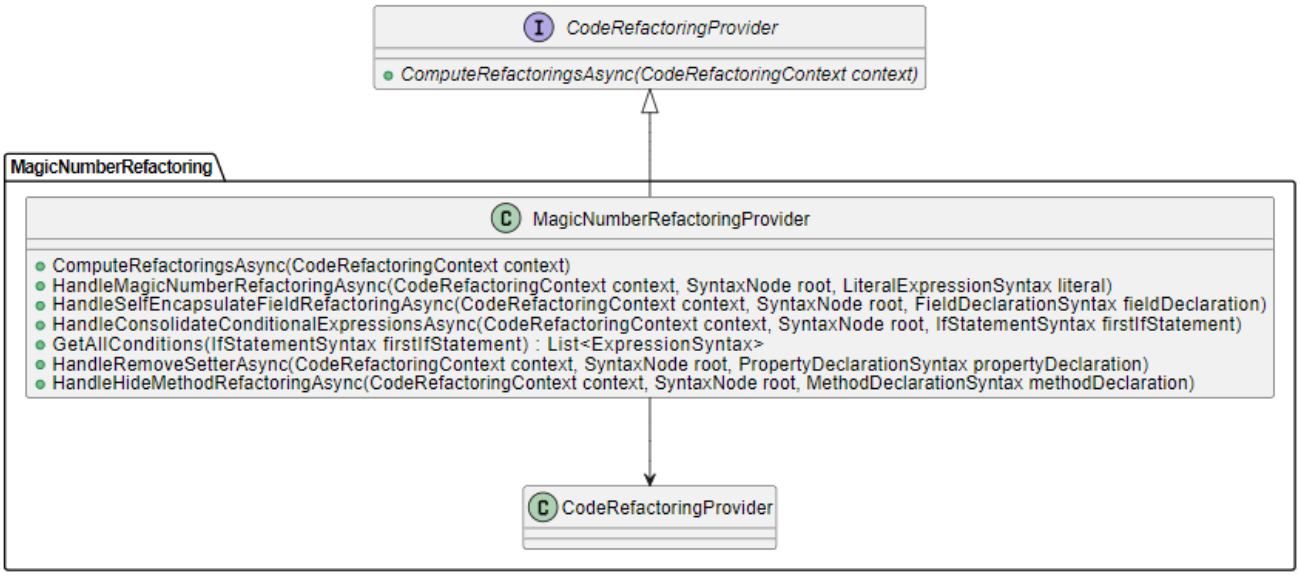

Рисунок 2.2 – UML-діаграма ієрархії класів

2.2 Аналіз технік рефакторингу та розробка блок-схем

Техніки рефакторингу – це набір шаблонів, які використовуються для покращення якості коду без зміни його зовнішньої поведінки. Деякі з типових технік рефакторингу включають вилучення дублікатів коду, перейменування змінних або методів, розділення або об'єднання класів, виділення методів або фрагментів коду у окремі функції, вилучення зайвих проміжних кроків, оптимізація структури програми та інші [18].

Кожна техніка має свої конкретні випадки використання та переваги, і техніки можуть бути комбіновані для досягнення кращих результатів. Існує безліч технік рефакторингу, кожна з яких призначена для певних ситуацій. Потрібно вибрати ті, які найкраще підходять для конкретного випадку і дозволять покращити якість коду.

Всього в компоненті реалізовано 5 технік рефакторингу:

– Replace Magic Number with Symbolic Constant (заміна магічного числа символьною константою);

– Hide Method (приховання методу);

– Consolidate Conditional Expression (об'єднання умовних операторів);

– Remove Setting Method (видалення сетера);

– Encapsulate Field (інкапсуляція поля);

Техніка «Заміна магічного числа символьною константою» потрібна, коли у коді використовується числове значення, яке не має чіткого змісту або змінюється від використання до використання. Це може призводити до неявних помилок, складнощів у зміні коду та зменшення зрозумілості для інших розробників [19]

Розв'язанням цієї проблеми є заміна числового значення символьною константою, яка має зрозуміле найменування. Це робить код більш зрозумілим, полегшує його розуміння та зберігання, а також допомагає уникнути помилок, пов'язаних із неправильними числовими значеннями.

На рис. 2.3 наведено алгоритм техніки Replace Magic Number with Symbolic Constant.

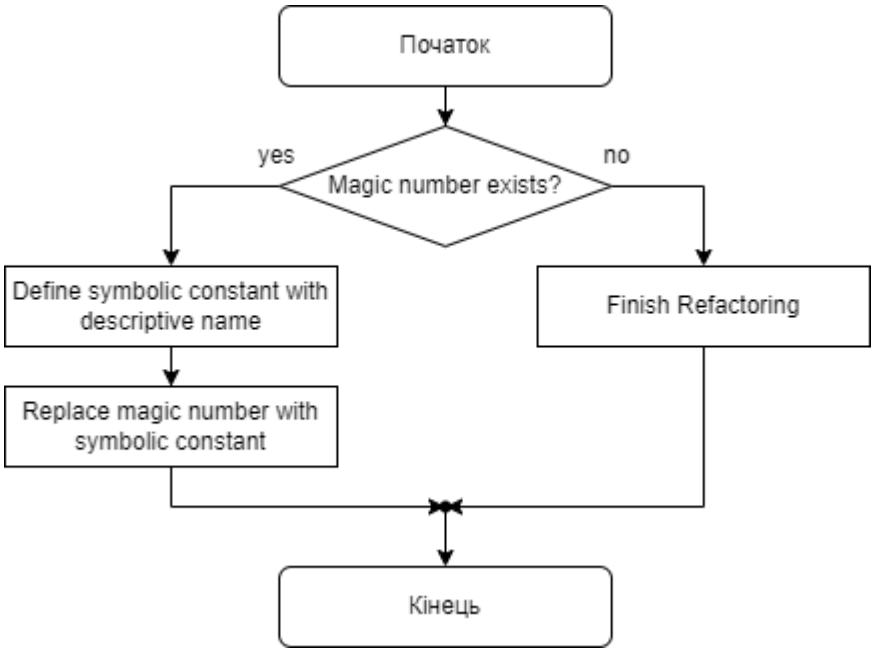

Рисунок 2.3 – Алгоритм "Replace Magic Number with Symbolic Constant"

Алгоритм Replace Magic Number with Symbolic Constant передбачає такі кроки:

– визначення магічного числа: перш ніж почати рефакторинг, потрібно знайти всі випадки використання магічного числа у коді;

– розробка символьної константи: після ідентифікації магічного числа створюється нова константа з чіткою та зрозумілою назвою, яка пояснює його значення.

Техніка рефакторингу Hide Method використовується для зниження рівня видимості методу або функції, який вважається зайвим або необов'язковим для зовнішнього користувача класу чи модуля [20].

Проблема: метод, який мало використовується або не повинен бути викликаний ззовні, все ще доступний для виклику.

Рішення: приховати метод, роблячи його приватним

На рис. 2.4 наведено алгоритм техніки Hide Method.

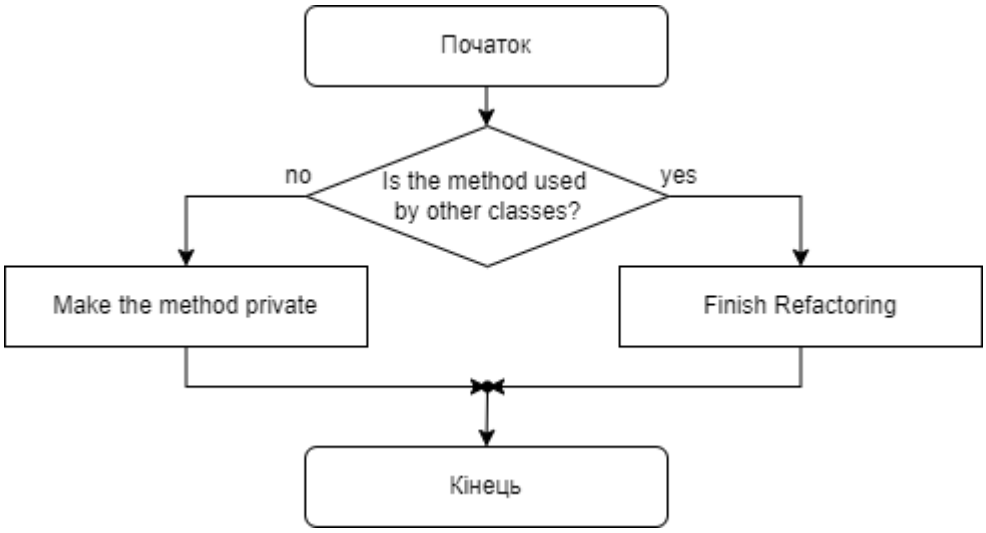

Рисунок 2.4 – Алгоритм "Hide Method"

Алгоритм Hide Method має такі кроки:

– перевірка, чи використовується метод іншими класами;

– якщо метод не використовується іншими класами (ні), переходимо до кроку 2;

– якщо метод використовується іншими класами (так), пропускаємо процес рефакторингу і завершуємо;

– якщо метод не використовується іншими класами, змінюємо його модифікатор доступу на приватний.

Техніка рефакторингу Об'єднання умовних операторів використовується для об'єднання двох або більше умовних операторів в один [21].

Проблема: Метод містить послідовність умовних операторів, що ускладнює розуміння коду та може призвести до зайвого повторення логіки.

Рішення: Заміна послідовності умовних операторів одним, який об'єднує умови за допомогою логічних операторів.

На рис. 2.5 наведено алгоритм техніки Consolidate Conditional Expression.

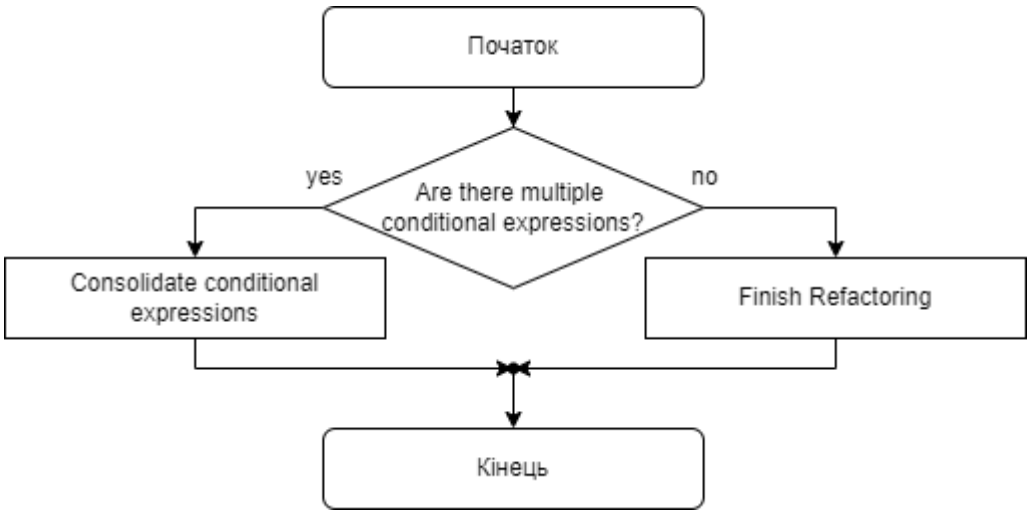

Рисунок 2.5 – Алгоритм "Consolidate Conditional Expression"

Алгоритм Consolidate Conditional Expression має кроки:

– перевірка наявності кількох умовних виразів у коді;

– якщо умовних виразів більше одного, то об'єднання їх у один умовний вираз.

Техніка рефакторингу Видалення сетера застосовується для заборони зміни стану об'єкта зовнішніми класами або модулями, які не повинні мати таку можливість [22].

Проблема: метод сетера використовується рідко або не використовується взагалі, і може призводити до неконтрольованих змін стану об'єкта.

Рішення: видалити метод сетера.

На рис. 2.6 наведено алгоритм техніки Remove Setting Method.

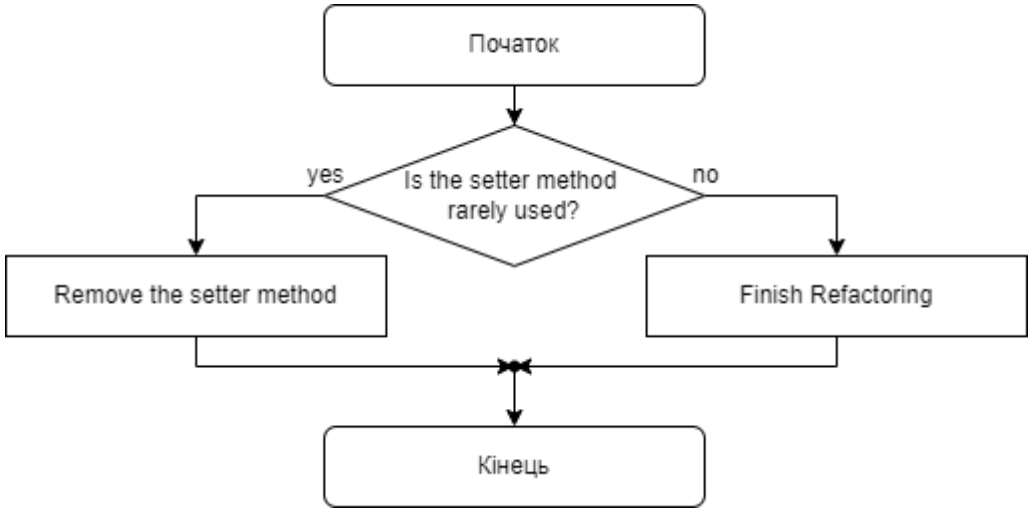

Рисунок 2.6 – Алгоритм "Remove Setting Method"

Алгоритм Remove Setting Method передбачає такі кроки:

– перевірка використання: перевірка, чи рідко використовується метод сетера;

– видалення сетера: якщо метод сетера дійсно рідко використовується, то видаляється.

Техніка рефакторингу Інкапсуляція поля застосовується для приховання прямого доступу до поля класу, забезпечуючи доступ до нього тільки через публічні методи класу [23].

Проблема: прямий доступ до поля класу може призвести до непередбачуваних змін значення поля ззовні, порушуючи принципи інкапсуляції та приховання деталей реалізації;

Рішення: приховати поле, зробивши його приватним, і забезпечити доступ до значення поля через методи доступу (гетери та сетери), які контролюють та обробляють доступ до даних.

На рис. 2.7 продемонстровано блок-схему алгоритму Encapsulate Field.

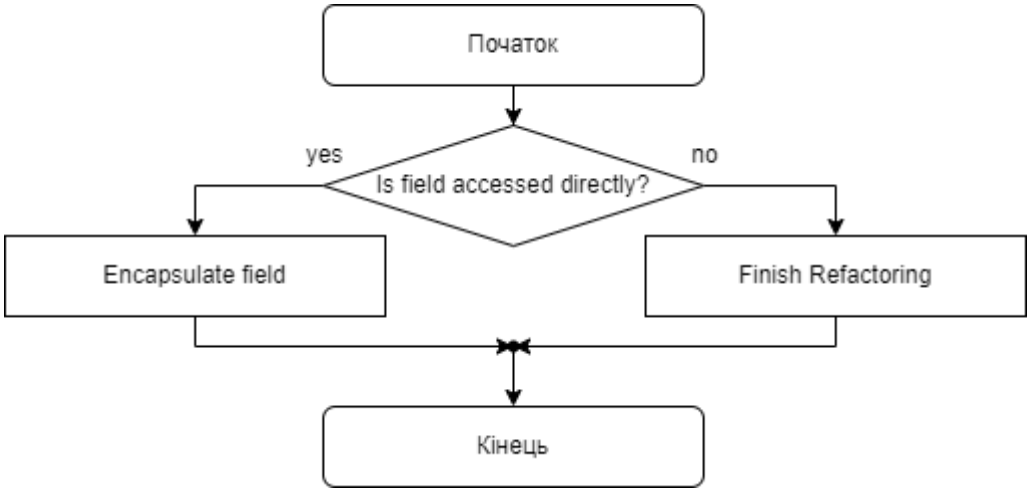

Рисунок 2.7 – Блок-схема алгоритму “Encapsulate Field”

Алгоритм Encapsulate Field передбачає такі кроки:

– перевірка прямого доступу до поля: спочатку перевіряється, чи поле безпосередньо доступне ззовні класу;

– інкапсуляція поля: якщо поле доступне ззовні, воно інкапсулюється і створює методи доступу (гетери та сетери) для звернення до поля ззовні.

2.3 Висновки до розділу

У ході моделювання компонента для автоматичного рефакторингу коду виконана розробка діаграми компонентів та діаграми ієрархії класів. Дослідження структури, функціональності та взаємозв'язків між компонентами дозволило визначити ключові аспекти, які впливають на ефективність та продуктивність рефакторингу. Ретельне проєктування діаграм та блок-схем рефакторингових операцій забезпечує чітке розуміння процесу оптимізації коду. Виконано аналіз п’яти технік рефакторингу та їх візуалізація у вигляді блок-схем.

3 РОЗРОБКА КОМПОНЕНТА

3.1 Аналіз використаних бібліотек

Використані бібліотеки головним чином включаються в склад Microsoft.CodeAnalysis, яка є частиною Roslyn – відкритої платформи компілятора C# та VB.NET від Microsoft. Вона надає інструменти для аналізу та модифікації вихідного коду. Основні компоненти бібліотеки включають [24]:

– Синтаксичний аналізатор: Дозволяє створювати абстрактне синтаксичне дерево (AST) з вихідного коду, що дозволяє аналізувати структуру програми.

– Семантичний аналізатор: Надає доступ до інформації про типи, змінні, методи та інші семантичні аспекти коду, що дозволяє проводити більш глибокий аналіз.

– Система символів: Включає в себе інформацію про символи програми, як-от класи, методи, змінні тощо, і надає доступ до них для подальшого аналізу та модифікації.

– Аналізатор компіляції: Забезпечує можливість компіляції вихідного коду на льоту, що дає змогу проводити аналіз інтерактивно під час редагування коду в середовищі розробки.

– Сервіси для рефакторингу: Надає інструменти для автоматичного виявлення та застосування рефакторингу, які полегшують покращення структури та якості коду.

Простір імен Microsoft.CodeAnalysis.CodeActions включає класи та інтерфейси, які дозволяють представляти та виконувати дії зміни вихідного коду в редакторі. Основні компоненти цього простору імен включають [25]:

– CodeAction: Це представлення одного конкретного дії зміни в коді, яке можна виконати. Включає назву дії та делегат, який визначає, як саме ця дія буде виконана.

– CodeActionContext: Контекст, який надає інформацію про місце, де можливо виконати дію зміни, а також доступ до додаткових параметрів та налаштувань.

– CodeActionProvider: Абстрактний клас або інтерфейс, який представляє провайдера дій зміни коду. Цей провайдер визначає, які саме дії доступні для конкретного контексту в коді.

– CodeActionOperation: Представлення окремої операції, яка включається до дії зміни коду. Наприклад, додавання нового рядка коду, видалення частини коду або заміна одного фрагмента коду іншим.

– CodeRefactoring: Представлення рефакторингової дії, яка може бути виконана для покращення якості коду без зміни його поведінки.

Простір імен Microsoft.CodeAnalysis.CodeRefactorings включає класи та інтерфейси, які дозволяють створювати та виконувати рефакторингові операції над вихідним кодом в середовищі Roslyn. Основні компоненти цього простору імен включають [26]:

– CodeRefactoringProvider: Абстрактний клас або інтерфейс, який представляє провайдера рефакторингів. Цей провайдер визначає, які рефакторинги доступні для конкретного контексту в коді.

– CodeRefactoringContext: Контекст, який надає інформацію про місце, де можливо застосувати рефакторинг, а також доступ до додаткових параметрів та налаштувань.

Microsoft.CodeAnalysis.CSharp – це простір імен, який містить класи та інші засоби для роботи з мовою програмування C# в середовищі Roslyn. Основні компоненти цього простору імен включають [27]:

– CSharpSyntaxNode: Базовий клас для всіх вузлів синтаксичного дерева C#.

– SyntaxFactory: Клас, який містить фабричні методи для розробки різних синтаксичних конструкцій мови C#.

– SyntaxKind: Перерахування, яке містить всі можливі типи синтаксичних вузлів в мові C#.

– SyntaxTree: Клас, який представляє синтаксичне дерево програми на мові C#. Він може бути використаний для аналізу та маніпулювання кодом на рівні синтаксичних конструкцій.

– CSharpCompilation: Клас, який представляє компіляцію програми на мові C#. Він надає методи для компіляції та аналізу програмного коду.

Простір імен Microsoft.CodeAnalysis.CSharp.Syntax містить класи, які представляють синтаксичні конструкції мови програмування C# в середовищі Roslyn. Цей простір імен є важливим для аналізу, маніпулювання та генерації синтаксичних вузлів коду на C#. Основні класи та компоненти цього простору імен включають [28]:

– SyntaxNode: Базовий клас для всіх синтаксичних вузлів в C#. Він представляє синтаксичну конструкцію та може містити дочірні вузли та токени.

– SyntaxToken: Клас, який представляє токен (лексему) в коді на C#. Він містить інформацію про сам токен та його місце в тексті програми.

– SyntaxTrivia: Клас, який представляє додаткову інформацію, яка відноситься до синтаксичних токенів, таку як коментарі, прогалини та підказки форматування.

– Різні класи синтаксичних вузлів: Наприклад, ClassDeclarationSyntax, MethodDeclarationSyntax, ExpressionSyntax і так далі. Кожен з цих класів представляє конкретну синтаксичну конструкцію в коді на C#.

Простір імен System.Linq включає в себе типи і методи, пов'язані з мовою LINQ (Language Integrated Query) в середовищі .NET Framework. LINQ надає можливість писати структуровані запити для отримання даних з різних джерел, таких як масиви, колекції, бази даних тощо. Основні складові простору імен System.Linq включають [29]:

– Розширені методи LINQ: Набір методів розширення, які надають зручний синтаксис для виконання операцій запиту, таких як фільтрація, сортування, групування, вибірка тощо. Наприклад: Where, OrderBy, Select, GroupBy.

– Інтерфейси запиту LINQ: Інтерфейси, які дозволяють виразно виражати структуровані запити. Наприклад: IQueryable, IEnumerable.

– Типи результатів запиту: Типи, які представляють результати виконання запиту. Наприклад: IGrouping, IOrderedQueryable.

– Інші утилітарні класи і типи: Наприклад, Enumerable – клас, який містить статичні методи для роботи з колекціями, та інші утилітарні типи, які допомагають у роботі з LINQ.

Простір імен System.Threading.Tasks включає класи та інтерфейси, які дозволяють робити асинхронні операції у програмах, що працюють в середовищі .NET Framework. Основні компоненти цього простору імен включають [30]:

– Клас Task: Представляє асинхронну операцію, яку можна виконувати паралельно з іншими операціями. Task може повертати значення або не мати результату.

– Клас Task<TResult>: Підклас Task, який представляє асинхронну операцію з результатом певного типу.

– Клас TaskFactory: Надає зручний спосіб розробки та запуску асинхронних завдань.

– Інтерфейс IAsyncResult: Інтерфейс, який використовується для реалізації асинхронних операцій у старіших версіях .NET Framework.

– Деякі інші утилітарні класи та інтерфейси: Наприклад, Parallel, який надає можливість паралельного виконання операцій, а також деякі інші класи для роботи з асинхронними операціями.

Простір імен Microsoft.CodeAnalysis.Formatting надає можливості для форматування вихідного коду програм за допомогою інструментів, які надаються платформою .NET Compiler Platform (Roslyn). Цей простір імен містить класи та інтерфейси, які дозволяють налаштовувати форматування коду відповідно до встановлених стандартів чи власних правил. Деякі ключові компоненти простору імен включають [31]:

– Клас SyntaxNodeFormatter: Надає можливості для форматування синтаксичних вузлів (SyntaxNode), які представляють вихідний код програм.

– Інтерфейс ISyntaxFormattingService: Визначає методи для форматування синтаксичних вузлів та тексту за допомогою різних правил форматування.

– Класи та інтерфейси для налаштування форматування: Включають можливості налаштування різних аспектів форматування, таких як вирівнювання, відступи, використання пробілів та табуляцій, стиль розміщення фігурних дужок тощо.

System – це один з основних просторів імен в мові програмування C#. Він містить базові класи та типи даних, які використовуються у багатьох програмах. Деякі ключові компоненти простору імен System включають [32]:

– System.Console: Клас для взаємодії з консоллю, такий як виведення тексту, зчитування введення користувача тощо.

– System.IO: Набір класів для роботи з файлами, каталогами та іншими ресурсами вводу-виводу.

– System.Collections: Включає базові класи та інтерфейси для роботи з колекціями об'єктів, такими як списки, масиви, стеки та черги.

– System.Threading: Містить класи та інтерфейси для роботи з потоками виконання, мультипотоковість та асинхронні операції.

– System.Diagnostics: Надає можливості для взаємодії з системними процесами, управління подіями та журналами подій.

– System.Text: Містить класи для операцій з рядками та текстовими даними, включаючи кодування та розкодування тексту.

System.Collections.Generic – це ще один простір імен в мові програмування C#, який надає загальні колекції, як-от списки, масиви, словники і черги, які можуть працювати з будь-яким типом даних. Основні компоненти System.Collections.Generic включають [33]:

– List<T>: Динамічний список, який може змінювати розмір під час виконання і містить елементи одного типу.

– Dictionary<TKey, TValue>: Асоціативний масив, який містить пари ключ-значення та дозволяє швидкий доступ до значення за допомогою ключа.

– Queue<T>: Колекція, що працює за принципом першим прийшов – першим вийшов, де елементи додаються в кінець черги і видаляються з початку.

– Stack<T>: Колекція, що працює за принципом останнім прийшов – першим вийшов, де елементи додаються та видаляються з кінця стеку.

3.2 Аналіз програмної архітектури

Головним компонентом програми є CodeRefactoringProvider, який виступає як основний двигун для виконання автоматичних рефакторингів коду в середовищі Visual Studio. Цей клас є ключовим елементом, що відповідає за аналіз, інтерпретацію та застосування різних видів рефакторингу до виявлених фрагментів коду. Він надає можливість автоматизувати та спростити процес покращення якості та читабельності коду для розробників, що працюють у середовищі [34].

Крім того, CodeRefactoringProvider взаємодіє з іншими складовими середовища, такими як редактором коду та панеллю рефакторингу, щоб забезпечити зручний та ефективний процес рефакторингу. Він відповідає за виконання операцій зміни структури коду.

Він використовує асинхронний метод ComputeRefactoringsAsync, щоб аналізувати код та визначати, який вид рефакторингу слід застосовувати до кожного знайденого фрагмента коду. Метод є асинхронним, що означає, що він виконується в асинхронному контексті, не блокуючи основний потік виконання програми. Це дозволяє програмі продовжувати виконання інших операцій або реагувати на події, поки метод виконує свою роботу.

Такий підхід є особливо корисним у випадках, коли операції, які виконуються в методі, можуть займати тривалий час, такий як аналіз великих обсягів даних або взаємодія з мережевими ресурсами.

В цьому методі відбувається аналіз коду з метою виявлення можливих місць для застосування рефакторингу. Такий асинхронний підхід дозволяє ефективно

використовувати ресурси та запобігає блокуванню користувальницького інтерфейсу в середовищі Visual Studio під час виконання операцій рефакторингу.

Ключовою перевагою асинхронного програмування є покращення продуктивності та реагування програми, а також зниження впливу на загальну продуктивність системи під час виконання важких операцій.

Метод працює програмно таким чином:

– Отримання синтаксичного дерева: Спочатку метод отримує синтаксичне дерево (див. рис. 1.1) коду, який аналізується. Це дерево представляє структуру програмного коду, розбиту на вузли, які представляють різні конструкції мови програмування.

– Пошук відповідних фрагментів коду: За допомогою синтаксичного дерева метод визначає контекстні області, в яких можливий рефакторинг коду.

– Виконання операцій рефакторингу: Після виявлення відповідних фрагментів коду метод виконує необхідні операції рефакторингу. Це може включати заміну коду, видалення непотрібних конструкцій, введення нових елементів або будь-які інші зміни, які покращують якість або читабельність коду.

– Розробка дії рефакторингу: Після виконання рефакторингу метод створює об'єкт дії рефакторингу (CodeAction), який включає в себе інформацію про зміни, що були внесені в код. Цей об'єкт може містити опис операції, яку можна виконати, а також новий код або посилання на файл з внесеними змінами.

– Реєстрація дії рефакторингу: Нарешті, створена дія рефакторингу реєструється у середовищі Visual Studio. Це дозволяє користувачу бачити доступні операції рефакторингу у панелі інструментів або контекстному меню та виконувати їх якщо треба.

Об'єкт CodeAction є ключовим компонентом у механізмі рефакторингу в середовищі Visual Studio. Він представляє собою дію або операцію, яку можна виконати над фрагментом коду для його покращення, реорганізації або оптимізації. Основні характеристики CodeAction включають [35]:

– Опис дії: CodeAction містить текстовий опис операції рефакторингу, яку він представляє. Цей опис зазвичай короткий та зрозумілий, щоб користувач міг швидко зрозуміти, яку зміну він збирається виконати.

– Застосування змін: Об'єкт CodeAction також містить логіку та інструкції щодо того, які зміни потрібно вносити в код для виконання відповідної операції рефакторингу.

– Контекстні умови: Деякі CodeAction можуть бути доступними лише в певних контекстах або для певних типів коду. Це допомагає уникнути випадкового або непридатного застосування рефакторингів та забезпечує коректну роботу механізму рефакторингу.

Загалом, об'єкт CodeAction виступає як абстракція для виконання конкретних операцій рефакторингу над фрагментами коду в середовищі Visual Studio, що робить його ключовим елементом у впровадженні автоматичних покращень у програмному коді.

3.3 Реалізація алгоритмів рефакторингу

Кожен з алгоритмів рефакторингу виконується асинхронно за допомогою ключового слова await, що дозволяє програмі виконувати інші завдання під час виконання операцій рефакторингу. Це дозволяє забезпечити плавну та продуктивну роботу програми навіть у випадку великої кількості запитів на рефакторинг або складних операцій над кодом [36].

В першому алгоритмі – Заміна магічного числа символьною константою (див. рис. 2.4, метод HandleMagicNumberRefactoringAsync), завданням є заміна магічних чисел у вихідному коді на символьні константи. Програмно алгоритм реалізується через такі кроки:

– Отримання кореневого вузла синтаксичного дерева: Спочатку необхідно отримати кореневий вузол синтаксичного дерева, який представляє весь вихідний код програми.

– Знаходження магічного числа в коді: Пошук вказаного магічного числа у синтаксичному дереві. Це виконується за допомогою методу FindNode, який отримує місце входження числа в коді.

– Аналіз значення магічного числа: Використовуючи модель семантики, аналізується код, щоб отримати значення магічного числа. Це значення потрібно для розробки символьної константи.

– Розробка символьної константи: На основі отриманого значення магічного числа створюється нова символьна константа з унікальним іменем.

– Заміна магічного числа на константу: Магічне число вихідного коду замінюється на використання новоствореної символьної константи.

– Реєстрація дії рефакторингу: Після заміни магічного числа на символьну константу, дія рефакторингу реєструється для подальшого підтвердження користувачем.

На рис. 3.1 продемонстровано застосування алгоритму техніки Replace Magic Number with Symbolic Constant:

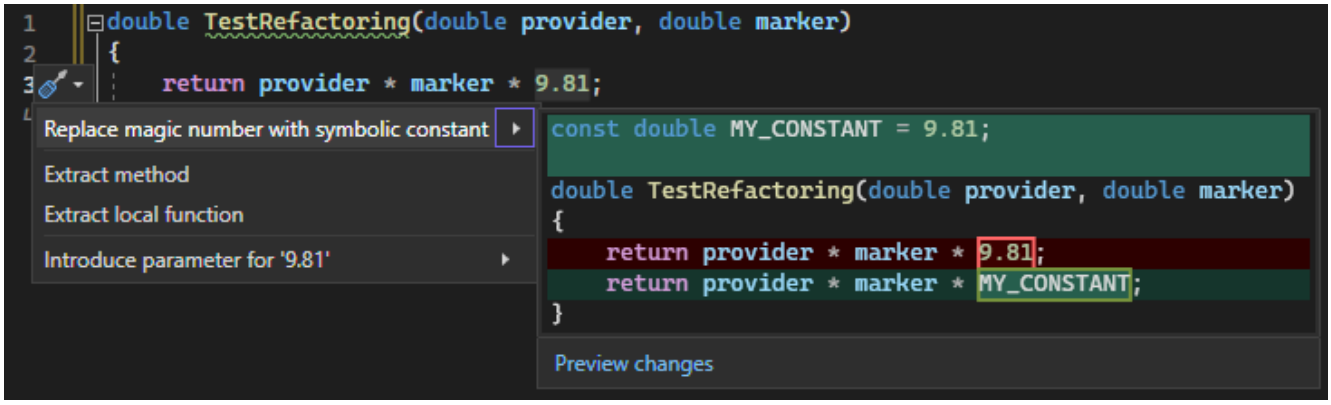

Рисунок 3.1 – Застосування алгоритму "Replace Magic Number with Symbolic Constant"

У другому алгоритмі – Інкапсуляція поля (рис. 2.8, метод HandleSelfEncapsulateFieldRefactoringAsync), завданням є автоматизація процесу перетворення публічних полів класу у властивості з відповідними гетерами і сетерами. Програмно алгоритм реалізується через такі кроки:

– Аналіз коду: Спочатку програма аналізує вихідний код для визначення наявності публічних полів, які потрібно інкапсулювати.

– Розробка приватних полів: Для кожного публічного поля створюється відповідне приватне поле, яке буде приховувати доступ до даних.

– Розробка методів доступу (гетера і сетера): Для кожного публічного поля створюються методи доступу (гетер і сетер), які забезпечують контрольований доступ до даних через приватне поле.

– Заміна публічних полів на властивості: Публічні поля замінюються на властивості, які використовують новостворені приватні поля та методи доступу.

– Реєстрація дії рефакторингу: Після здійснення змін в коді, програма реєструє дію рефакторингу для підтвердження користувачем та збереження змін у вихідному коді.

На рис. 3.2 продемонстровано застосування алгоритму техніки Encapsulate Field:

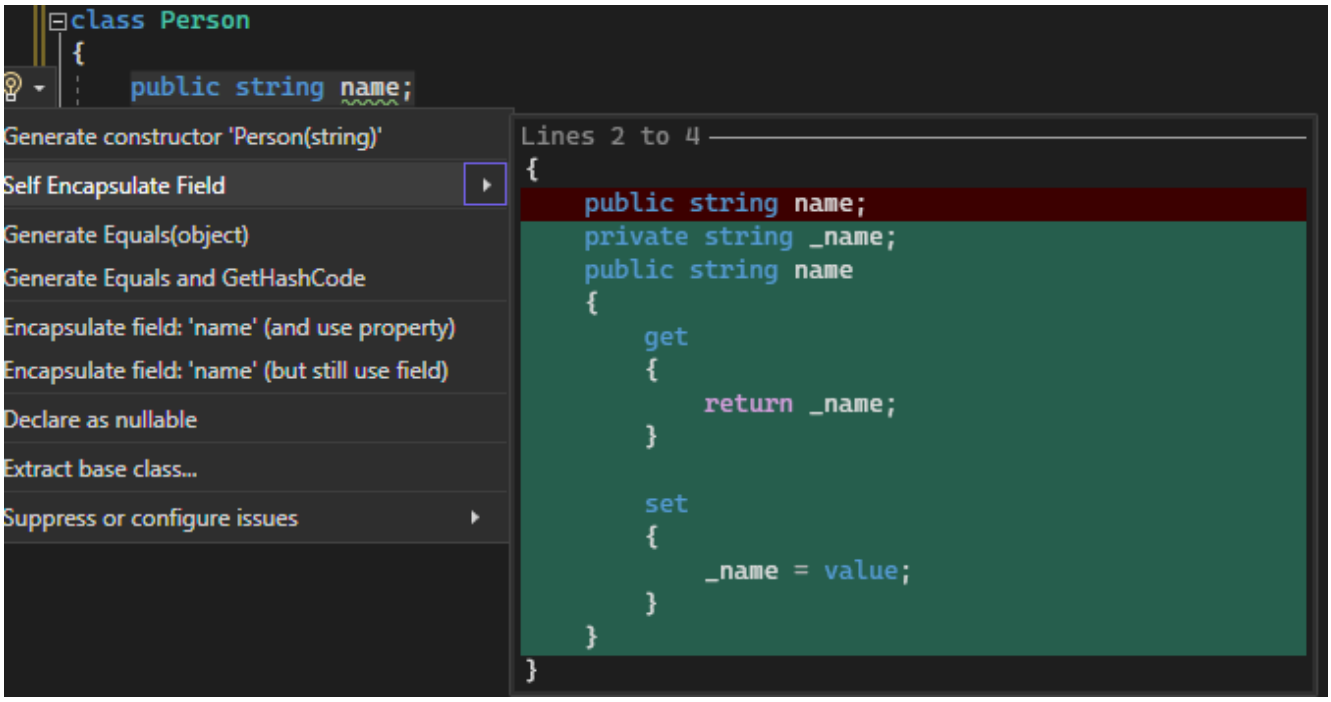

Рисунок 3.2 – Застосування алгоритму "Encapsulate Field"

У третьому алгоритмі – Об'єднання умовних операторів (рис. 2.6)(HandleConsolidateConditionalExpressionsAsync), завданням є автоматичне

об'єднання послідовності умовних виразів в один. Програмно алгоритм реалізується через такі кроки:

– Аналіз умовних виразів: Перш ніж почати об'єднання, програма аналізує послідовність умовних виразів для визначення можливості їх об'єднання.

– Об'єднання умовних виразів: Якщо знайдено послідовність умовних виразів, програма об'єднує їх в один вираз за допомогою логічного оператора.

– Оновлення коду: Після об'єднання умовних виразів програма оновлює вихідний код, замінюючи початкові умовні вирази новим об'єднаним виразом.

На рис. 3.3 продемонстровано застосування алгоритму техніки Consolidate Conditional Expression:

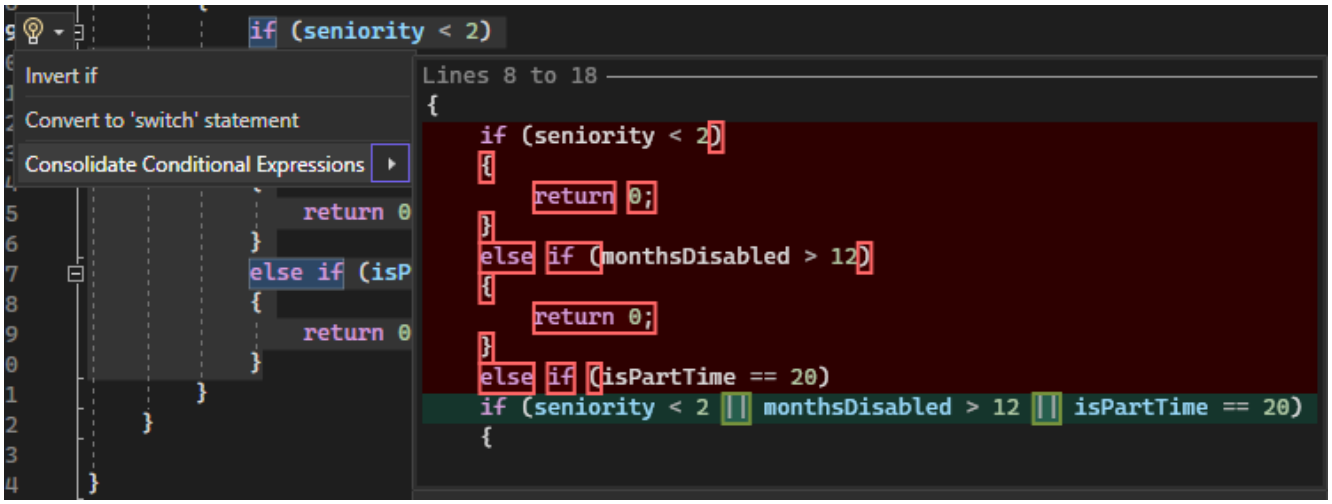

Рисунок 3.3 – Застосування алгоритму "Consolidate Conditional Expression"

У четвертому алгоритмі – Видалення сетера (рис. 2.7, метод HandleRemoveSetterAsync), завданням є автоматичне вилучення методу сетера для властивості класу. Програмно алгоритм реалізується через такі кроки:

– Знаходження властивості з сетером: Першим кроком програма аналізує код для визначення властивостей, які мають сетер.

– Вилучення сетера: Після виявлення властивості з сетером програма вилучає цей сетер з відповідного коду класу.

– Оновлення викликів сетера: Якщо виклики сетера існують в інших частинах коду, програма оновлює їх, щоб вони більше не викликали вилучений сетер.

На рис. 3.4 продемонстровано застосування алгоритму техніки Remove Setting Method:

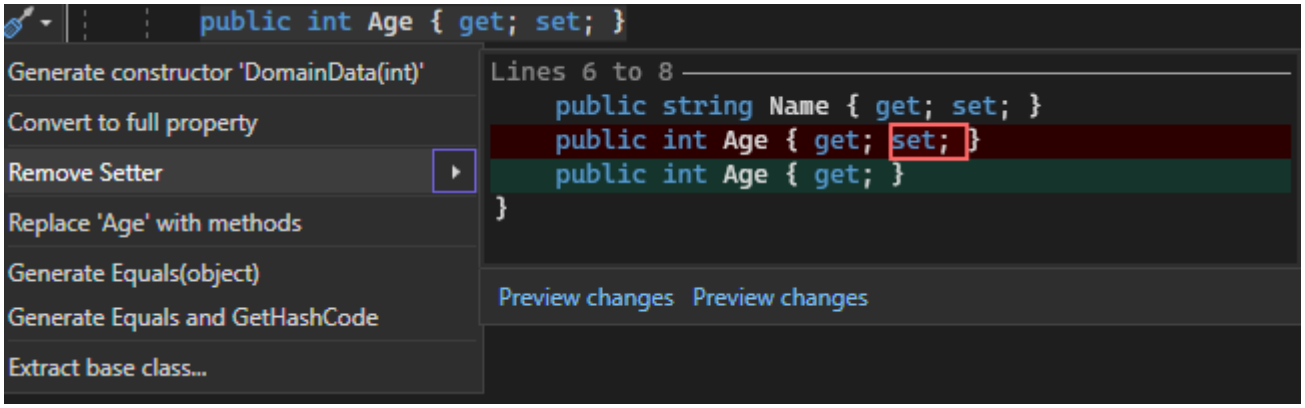

Рисунок 3.4 – Застосування алгоритму "Remove Setting Method"

У п`ятому алгоритмі – Приховання методу (рис. 2.5, метод HandleHideMethodRefactoringAsync), завданням є автоматичне перетворення публічного методу класу на приватний. Програмно алгоритм реалізується через такі кроки:

– Знаходження методу для приховування: Першим кроком програма аналізує код для визначення публічних методів, які можна перетворити на приватні.

– Перетворення методу на приватний: Після виявлення методу для приховування програма змінює його модифікатор доступу.

– Оновлення викликів методу: Якщо метод викликається в інших частинах коду, програма оновлює ці виклики, змінюючи їх доступ до методу на приватний.

На рис. 3.5 продемонстровано застосування алгоритму техніки Hide Method:

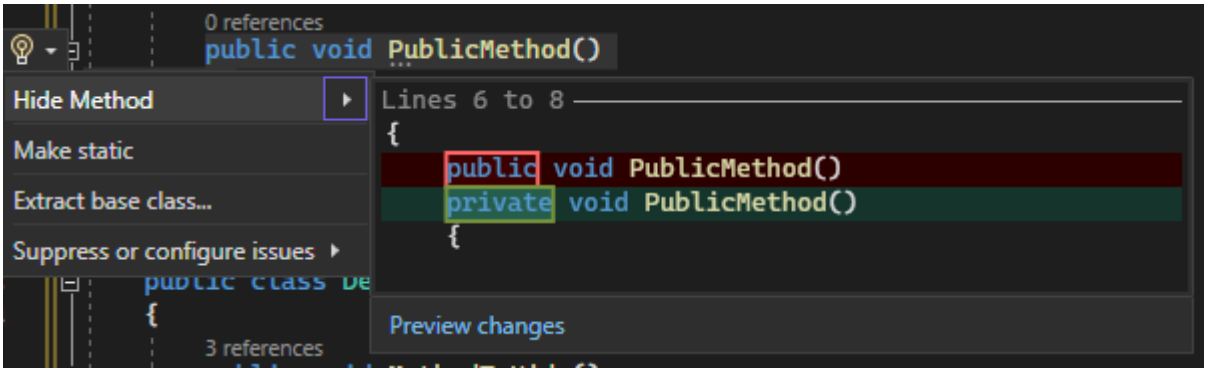

Рисунок 3.5 – Застосування алгоритму "Hide Method"

3.4 Аналіз результатів автоматичного рефакторингу

Для оцінки ефективності розробленого компоненту автоматичного рефакторингу коду в середовищі Visual Studio було проведено тестування на кількох наборах "брудного" коду.

Заміна магічного числа символьною константою (Replace Magic Number with Symbolic Constant):

– час виконання: алгоритм спрацював швидко, виконуючи заміну магічних чисел на константи без значних затримок;

– точність: у всіх випадках магічні числа були коректно ідентифіковані та замінені на константи;

– вплив на код: читабельність коду значно покращилася, оскільки символьні константи надавали більше сенсу значенням, використаним у програмі;

– недоліки: деякі складні випадки, де магічні числа були частиною виразів або використовувалися в різних контекстах, вимагали ручної перевірки після рефакторингу.

Інкапсуляція поля (Encapsulate Field):

– час виконання: процес інкапсуляції полів був швидким та ефективним;

– точність: публічні поля були коректно замінені на приватні поля з відповідними методами доступу;

– вплив на код: код став більш структурованим, з чітко визначеними методами доступу до даних;

– недоліки: у випадках, коли поле мало спеціальні умови або валідації, доводилося вручну коригувати згенеровані методи доступу.

Об'єднання умовних операторів (Consolidate Conditional Expressions):

– час виконання: алгоритм об'єднання умовних виразів працював ефективно навіть для великих наборів умов;

– точність: умовні вирази були коректно об'єднані, що спростило логіку коду;

– вплив на код: код став менш розгалуженим та більш читабельним, що зменшило ймовірність помилок;

– недоліки: у деяких складних логічних конструкціях доводилося додатково перевіряти правильність об'єднання.

Видалення сетера (Remove Setter):

– час виконання: алгоритм швидко знаходив та видаляв сетери;

– точність: усі сетери, що підлягали видаленню, були коректно видалені без залишкових слідів у коді;

– вплив на код: видалення сетерів допомогло підвищити безпеку даних та зменшити кількість можливих побічних ефектів при зміні властивостей;

– недоліки: у випадках, де сетери мали додаткову логіку, вимагалося ручне втручання для забезпечення коректної роботи коду після рефакторингу.

Приховання методу (Hide Method):

– час виконання: алгоритм приховання методів працював ефективно та швидко;

– точність: усі знайдені методи були коректно перетворені на приватні;

– вплив на код: код став більш інкапсульованим, що покращило його безпеку та зменшило ймовірність ненавмисного використання методів;

– недоліки: у випадках, коли методи використовувалися в інших класах або проєктах, доводилося коригувати виклики методів.

Розроблений компонент для автоматичного рефакторингу коду в середовищі Visual Studio показав високу ефективність у спрощенні та покращенні якості коду. Алгоритми виконувалися швидко і точно, з мінімальною потребою в

ручних корекціях. Основні переваги включають підвищення читабельності коду, зменшення кількості помилок та оптимізацію процесу розробки.

Однак, у деяких випадках складність коду вимагала додаткової перевірки та корекції після виконання рефакторингу. Загалом, компонент демонструє високу корисність для автоматизації процесів рефакторингу в реальних умовах розробки програмного забезпечення.

3.5 Робота користувача з компонентом

Для використання компонента потрібно мати останню версію Visual Studio 2022 Community Edition (рис 3.6) [37].

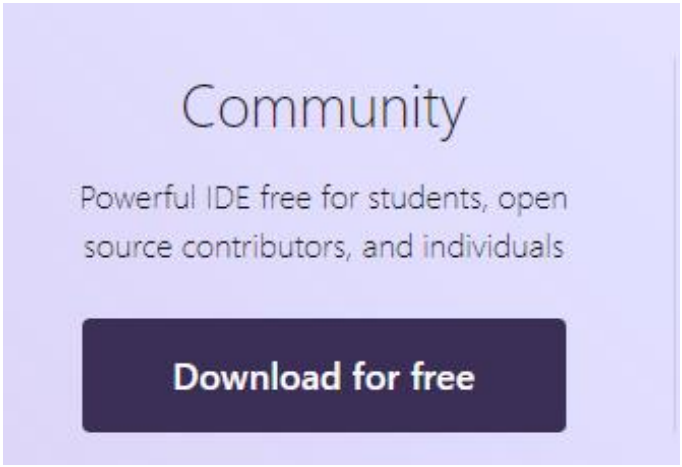

Рисунок 3.6 – Visual Studio 2022 Community

У Visual Studio потрібно встановити пакет Visual Studio extension development (рис. 3.7).

Далі для встановлення розширення потрібно відкрити вікно Extensions, а потім Manage Extensions (рис. 3.8). У полі пошуку знайти CodeRefactoring3 і натиснути Download.

Або можна встановити компонент через інсталятор (рис. 3.9). Потрібно запустити файл з розширенням .vsix і натиснути на кнопку Install.

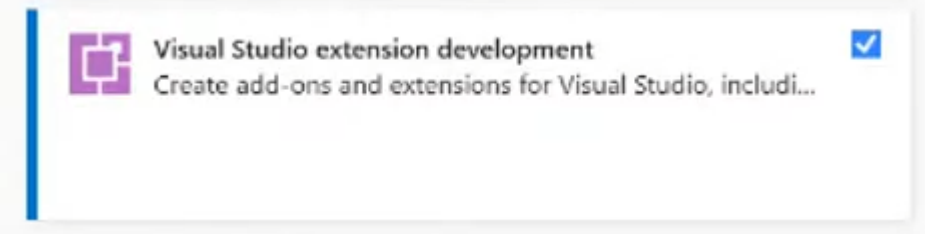

Рисунок 3.7 – Visual Studio extension development

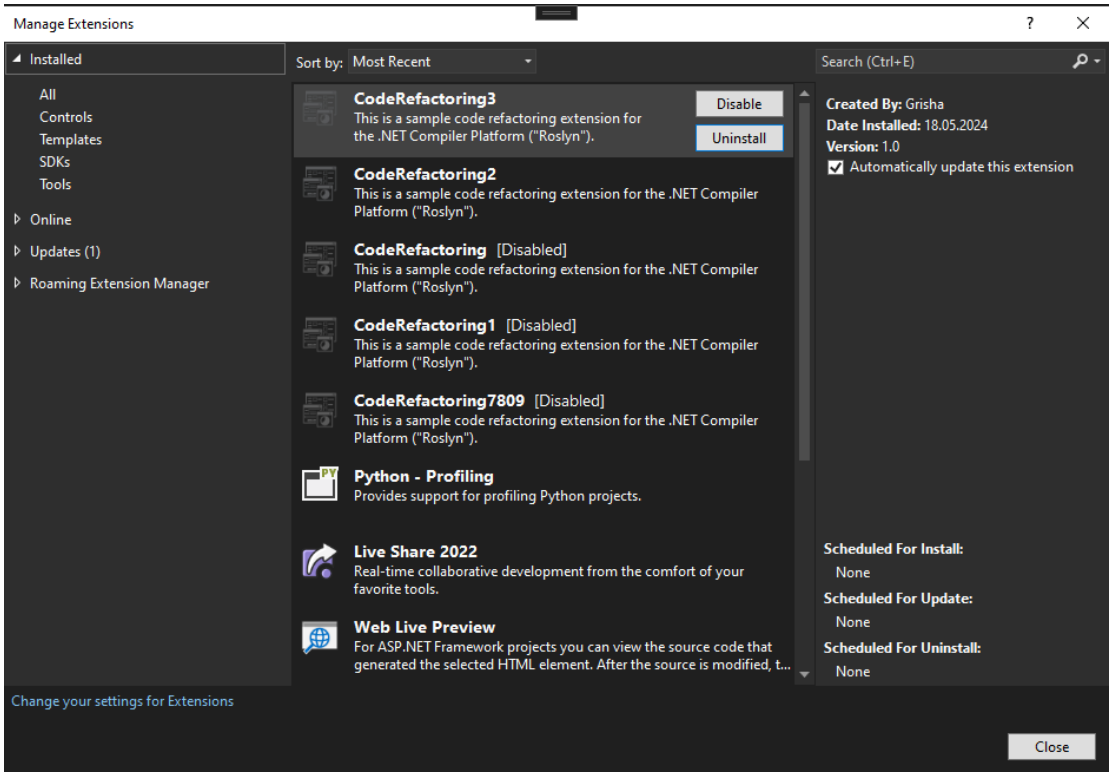

Рисунок 3.8 – Налаштування середовища

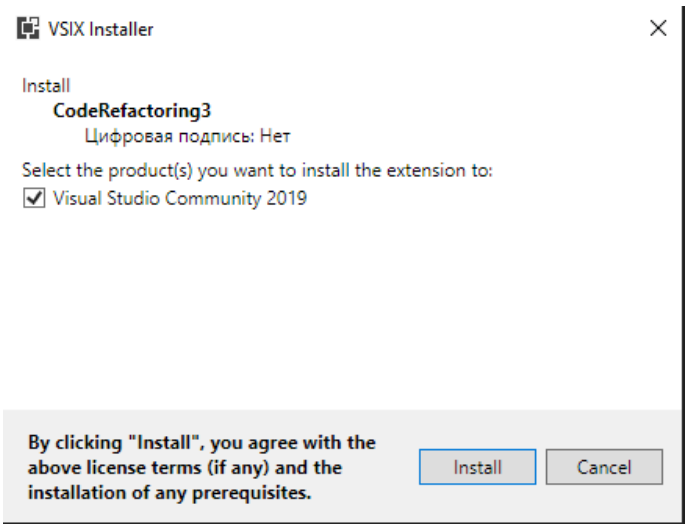

Рисунок 3.9 – Інсталятор Vsix

Для роботи з компонентом потрібно виділити фрагмент коду, який потрібно підвергнути рефакторингу та натиснути на іконку Quick Actions, або натиснути поєднання клавіш Alt + Enter.

3.6 Висновки до розділу

Під час реалізації компонента стало зрозуміло, що чітке структурування та планування всіх етапів є необхідним для досягнення ефективності та продуктивності рефакторингу. Зокрема, важливість отримання та аналізу синтаксичного дерева коду, пошуку відповідних фрагментів та виконання точних операцій рефакторингу дуже важливі при розробці алгоритмів.

Процес інтеграції компонента у середовище Visual Studio 2022 Community підтвердив, що забезпечення зручності користування та простоти встановлення через пакет розширень або інсталятор Vsix є важливим для його широкого прийняття розробниками. Це дозволило визначити, що доступність та інтуїтивність використання компонента безпосередньо впливають на його ефективність.

ВИСНОВКИ

У роботі вирішено завдання моделювання, програмної реалізації та оцінки ефективності компонента для автоматизованого рефакторингу коду на мові програмування C# у середовищі Visual Studio.

На основі виконаної роботи можна зробити кілька ключових висновків, що відображають як теоретичні, так і практичні аспекти розробки та впровадження програмних компонентів.

Проведений аналіз сучасних підходів до рефакторингу коду підтвердив важливість цього процесу для покращення якості, продуктивності та підтримки програмних систем.

У ході моделювання компонента для автоматичного рефакторингу коду виконана розробка UML-діаграм компонентів та ієрархії класів. Дослідження структури, функціональності та взаємозв'язків між компонентами дозволило визначити ключові аспекти, які впливають на ефективність та продуктивність рефакторингу. Ретельне проєктування діаграм та блок-схем рефакторингових операцій забезпечує чітке розуміння процесу оптимізації коду.

Виконано аналіз п’яти технік рефакторингу та їх візуалізація у вигляді блок-схем. Використання асинхронних методів для виконання операцій рефакторингу забезпечує плавну та безперебійну роботу середовища розробки навіть при великому навантаженні.

Розробка конкретних алгоритмів для виконання автоматичного рефакторингу продемонструвала можливість автоматизації рутинних завдань, що значно зменшує час, витрачений розробниками на виконання цих операцій вручну.

Інтеграція компонента у середовище Visual Studio 2022 Community Edition підтвердила важливість забезпечення зручності встановлення та використання програмного продукту. Простота інтеграції через розширення або інсталятор робить компонент доступним для широкого кола розробників, що сприяє його поширенню та використанню.

На основі цього дослідження можна зробити висновок, що ретельне моделювання та реалізація компонентів для автоматичного рефакторингу коду є ключовим етапом у забезпеченні високої якості програмних продуктів. Впровадження таких компонентів дозволяє значно підвищити ефективність процесу розробки, зменшити кількість помилок у коді та забезпечити легкість підтримки програмних систем. Розроблений компонент для автоматичного рефакторингу коду в Visual Studio має потенціал стати цінним інструментом в арсеналі сучасного розробника.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке рефакторинг коду і чому він важливий для тестувальників. URL: https://training.qatestlab.com/blog/technical-articles/what-is-code-refactoring-and-why-is-it-important-for-testers/

2. ReSharper. URL: https://reporter.zp.ua/resharper-l-uk.html

3. ReSharper. The Visual Studio Extension for .NET Developers URL: https://www.jetbrains.com/resharper/

4. About CodeRush. URL: https://www.componentsource.com/product/coderush/about

5. CodeRush for VS 2022. URL: https://marketplace.visualstudio.com/items?itemName=DevExpress.CodeRushforVS2022

6. Using the Visual Studio Code Refactoring Tools. URL: https://www.codeguru.com/tools/using-the-visual-studio-code-refactoring-tools/

7. What is Visual Studio? URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022

8. Visual Studio SDK. URL: https://learn.microsoft.com/en-us/visualstudio/extensibility/visual-studio-sdk?view=vs-2022

9. roslyn–analyzers. URL: https://github.com/dotnet/roslyn-analyzers

10. Abstract Syntax Tree. URL: https://www.sciencedirect.com/topics/computer-science/abstract-syntax-tree

11. Abstract Syntax Tree (AST) – Explained in Plain English. URL: https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38

12. What Is a Semantic Model? URL: https://docs.oracle.com/en/cloud/paas/analytics-cloud/acmdg/what-is-semantic-model.html

13. Semantic models in simple terms. URL: https://blog.tabulareditor.com/2023/11/17/semantic-models-in-simple-terms/

14. Про середовище програмування C#. URL: https://foxminded.ua/seredovyshche-prohramuvannia-si-sharp/

15. Що таке діаграма об’єктів в UML? Навчайтеся на прикладі. URL: https://www.guru99.com/uk/uml-object-diagram.html

16. Повне розуміння діаграми компонентів UML за допомогою легкого методу. URL: https://www.mindonmap.com/uk/blog/uml-component-diagram/

17. Що таке діаграма класів UML і найкращий творець діаграм класів UML. URL: https://www.mindonmap.com/uk/blog/what-is-uml-class-diagram/

18. Прийоми рефакторингу. URL: https://refactoring.guru/uk/refactoring/techniques

19. Заміна магічного числа символьною константою. URL: https://refactoring.guru/uk/replace-magic-number-with-symbolic-constant

20. Приховання методу. URL: https://refactoring.guru/uk/hide-method

21. Об'єднання умовних операторів. URL: https://refactoring.guru/uk/consolidate-conditional-expression

22. Видалення сетера. URL: https://refactoring.guru/uk/remove-setting-method

23. Інкапсуляція поля. URL: https://refactoring.guru/uk/encapsulate-field

24. Microsoft.CodeAnalysis Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis?view=roslyn-dotnet-4.7.0

25. Microsoft.CodeAnalysis.CodeActions Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.codeactions?view=roslyn-dotnet-4.6.0

26. Microsoft.CodeAnalysis.CodeRefactorings Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.coderefactorings?view=roslyn-dotnet-4.7.0

27. Microsoft.CodeAnalysis.CSharp Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp?view=roslyn-dotnet-4.7.0

28. Microsoft.CodeAnalysis.CSharp.Syntax Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.syntax?view=roslyn-dotnet-4.7.0

29. System.Linq Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/system.linq?view=net-8.0

30. System.Threading.Tasks Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks?view=net-8.0

31. Microsoft.CodeAnalysis.Formatting Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.formatting?view=roslyn-dotnet-4.7.0

32. System Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/system?view=net-8.0

33. System.Collections.Generic Namespace. URL: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-8.0

34. CodeRefactoringProvider Class. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.coderefactorings.coderefactoringprovider?view=roslyn-dotnet-4.1.0

35. CodeAction Class. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.codeactions.codeaction?view=roslyn-dotnet-4.6.0

36. Asynchronous programming scenarios. URL: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios

37. Visual Studio 2022. URL: https://visualstudio.microsoft.com/downloads/

ДОДАТКИ

Додаток А. Програмний код

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CodeRefactorings;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis.Formatting;
using System;
using System.Collections.Generic;
using Microsoft.CodeAnalysis.FindSymbols;

namespace MagicNumberRefactoring
{
[ExportCodeRefactoringProvider(LanguageNames.CSharp,
    Name = nameof(MagicNumberRefactoringProvider))]
1 reference
public class MagicNumberRefactoringProvider : CodeRefactoringProvider
{
    0 references
    public override async Task ComputeRefactoringsAsync(CodeRefactoringContext context)
    {
        var root = await context.Document.GetSyntaxRootAsync
            (context.CancellationToken).ConfigureAwait(false);
        var node = root.FindNode(context.Span);

        if (node is LiteralExpressionSyntax literal && literal.IsKind
            (SyntaxKind.NumericLiteralExpression))
        {
            await HandleMagicNumberRefactoringAsync(context, root, literal);
        }
        else if (node is FieldDeclarationSyntax fieldDeclaration && fieldDeclaration.Modifiers.Any
            (SyntaxKind.PublicKeyword))
        {
            await HandleSelfEncapsulateFieldRefactoringAsync(context, root, fieldDeclaration);
        }
    else if (node is IfStatementSyntax firstIfStatement)
    {
        await HandleConsolidateConditionalExpressionsAsync(context, root, firstIfStatement);
    }
    else if (node is PropertyDeclarationSyntax propertyDeclaration)
    {
        await HandleRemoveSetterAsync(context, root, propertyDeclaration);
    }
    else if (node is MethodDeclarationSyntax methodDeclaration)
    {
        await HandleHideMethodRefactoringAsync(context, root, methodDeclaration);
    }
}
```

```
private async Task HandleMagicNumberRefactoringAsync(CodeRefactoringContext context,
    SyntaxNode root, LiteralExpressionSyntax literal)
{

    var semanticModel = await context.Document.GetSemanticModelAsync
        (context.CancellationToken).ConfigureAwait(false);


    var constantValue = semanticModel.GetConstantValue(literal,
        context.CancellationToken);
    if (!constantValue.HasValue)
        return;
var constantName = "MY_CONSTANT";
var constant = SyntaxFactory.FieldDeclaration(
    SyntaxFactory.VariableDeclaration(
        SyntaxFactory.ParseTypeName("double"),
        SyntaxFactory.SingletonSeparatedList(
            SyntaxFactory.VariableDeclarator(
                SyntaxFactory.Identifier(constantName))
                .WithInitializer(
                    SyntaxFactory.EqualsValueClause(
                        SyntaxFactory.LiteralExpression(
                            SyntaxKind.NumericLiteralExpression,
                            SyntaxFactory.Literal(Convert.ToDouble(constantValue.Value)))))))
    .WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.ConstKeyword)));


var newRoot = root.ReplaceNode(literal, SyntaxFactory.IdentifierName(constantName));

    newRoot = newRoot.InsertNodesBefore(newRoot.ChildNodes().First(), new[] { constant });


    var action = CodeAction.Create("Replace magic number with symbolic constant",
        cancellationToken =>
        Task.FromResult(context.Document.WithSyntaxRoot(newRoot)));


    context.RegisterRefactoring(action);
}
private async Task HandleSelfEncapsulateFieldRefactoringAsync(CodeRefactoringContext context,
    SyntaxNode root, FieldDeclarationSyntax fieldDeclaration)
{

    var semanticModel = await context.Document.GetSemanticModelAsync
        (context.CancellationToken).ConfigureAwait(false);


    var variableDeclarator = fieldDeclaration.Declaration.Variables.First();
    var variableName = variableDeclarator.Identifier.Text;


    var privateFieldName = "_" + char.ToLower(variableName[0]) + variableName.Substring(1);
```

```
var privateField = fieldDeclaration
    .WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PrivateKeyword)))
    .WithDeclaration(SyntaxFactory.VariableDeclaration(fieldDeclaration.Declaration.Type,
        SyntaxFactory.SingletonSeparatedList(SyntaxFactory.VariableDeclarator(privateFieldName))))
    .WithTriviaFrom(fieldDeclaration)
    .WithAdditionalAnnotations(Formatter.Annotation);


var getter = SyntaxFactory.AccessorDeclaration(SyntaxKind.GetAccessorDeclaration)
   .WithBody(SyntaxFactory.Block(SyntaxFactory.SingletonList<StatementSyntax>(
       SyntaxFactory.ReturnStatement(SyntaxFactory.IdentifierName(privateFieldName)))));


var setter = SyntaxFactory.AccessorDeclaration(SyntaxKind.SetAccessorDeclaration)
    .WithBody(SyntaxFactory.Block(SyntaxFactory.SingletonList<StatementSyntax>(
        SyntaxFactory.ExpressionStatement(
            SyntaxFactory.AssignmentExpression(
                SyntaxKind.SimpleAssignmentExpression,
                SyntaxFactory.IdentifierName(privateFieldName),
                SyntaxFactory.IdentifierName("value"))))));

 var property = SyntaxFactory.PropertyDeclaration(
     fieldDeclaration.Declaration.Type,
     variableName)
     .WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword)))
     .WithAccessorList(SyntaxFactory.AccessorList(SyntaxFactory.List(new[] { getter, setter })))
     .WithTriviaFrom(fieldDeclaration)
     .WithAdditionalAnnotations(Formatter.Annotation);


 var newRoot = root.ReplaceNode(fieldDeclaration, privateField);


 var privateFieldNode = newRoot.DescendantNodes().
     OfType<FieldDeclarationSyntax>()
     .FirstOrDefault(f => f.Declaration.Variables.Any
     (v => v.Identifier.Text == privateFieldName));
 if (privateFieldNode != null)
 {
     newRoot = newRoot.InsertNodesAfter(privateFieldNode, new[] { property });
 }
     var action = CodeAction.Create("Self Encapsulate Field", cancellationToken =>
         Task.FromResult(context.Document.WithSyntaxRoot(newRoot)));


     context.RegisterRefactoring(action);
}

1 reference
private async Task HandleConsolidateConditionalExpressionsAsync
    (CodeRefactoringContext context, SyntaxNode root, IfStatementSyntax firstIfStatement)
{
    var allConditions = GetAllConditions(firstIfStatement);

    if (allConditions.Count < 2)
        return;

    var consolidatedCondition = allConditions.Aggregate
        ((previous, current) =>
        SyntaxFactory.BinaryExpression
        (SyntaxKind.LogicalOrExpression, previous, current));
```

```
    var newIfStatement = firstIfStatement
        .WithCondition(consolidatedCondition)
        .WithElse(null);

    var newRoot = root.ReplaceNode(firstIfStatement, newIfStatement);

    var action = CodeAction.Create("Consolidate Conditional Expressions",
        cancellationToken =>
        Task.FromResult(context.Document.WithSyntaxRoot(newRoot)));
    context.RegisterRefactoring(action);
}
 private List<ExpressionSyntax> GetAllConditions(IfStatementSyntax firstIfStatement)
 {
     var allConditions = new List<ExpressionSyntax> { firstIfStatement.Condition };

     var currentStatement = firstIfStatement.Else?.Statement;
     while (currentStatement is IfStatementSyntax ifStatement)
     {
         allConditions.Add(ifStatement.Condition);
         currentStatement = ifStatement.Else?.Statement;
     }

     return allConditions;
 }
private async Task HandleRemoveSetterAsync(CodeRefactoringContext context,
    SyntaxNode root, PropertyDeclarationSyntax propertyDeclaration)
{

    var setter = propertyDeclaration.AccessorList?.Accessors.FirstOrDefault
        (accessor => accessor.Kind() == SyntaxKind.SetAccessorDeclaration);

    if (setter == null)
        return;


    var newAccessorList = propertyDeclaration.AccessorList.RemoveNode
        (setter, SyntaxRemoveOptions.KeepNoTrivia);


    var newPropertyDeclaration = propertyDeclaration.WithAccessorList(newAccessorList);


     var newRoot = root.ReplaceNode(propertyDeclaration, newPropertyDeclaration);


     var action = CodeAction.Create("Remove Setter", cancellationToken =>
         Task.FromResult(context.Document.WithSyntaxRoot(newRoot)));


     context.RegisterRefactoring(action);
 }
```

```
private async Task HandleHideMethodRefactoringAsync(CodeRefactoringContext context,
    SyntaxNode root, MethodDeclarationSyntax methodDeclaration)
{

    if (methodDeclaration.Modifiers.Any(SyntaxKind.StaticKeyword) || methodDeclaration.
        Modifiers.Any(SyntaxKind.PrivateKeyword))
    {
        return;
    }

    var semanticModel = await context.Document.GetSemanticModelAsync
        (context.CancellationToken).ConfigureAwait(false);
    var methodSymbol = semanticModel.GetDeclaredSymbol(methodDeclaration);
  if (methodSymbol == null)
      return;

  var references = await SymbolFinder.FindReferencesAsync(methodSymbol,
      context.Document.Project.Solution, context.CancellationToken).ConfigureAwait(false);
  var isUsedOnlyInHierarchy = references.All(reference =>
  {
      return reference.Locations.All(location =>
      {
          var referenceNode = root.FindNode(location.Location.SourceSpan);
          return referenceNode.AncestorsAndSelf().OfType<TypeDeclarationSyntax>().Any(typeDecl =>
              typeDecl.Identifier.ValueText == methodSymbol.ContainingType.Name);
      });
  });
            var privateMethod = methodDeclaration
                .WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PrivateKeyword)))
                .WithTriviaFrom(methodDeclaration)
                .WithAdditionalAnnotations(Formatter.Annotation);

            var newRoot = root.ReplaceNode(methodDeclaration, privateMethod);

            var action = CodeAction.Create("Hide Method", cancellationToken =>
                Task.FromResult(context.Document.WithSyntaxRoot(newRoot)));

            context.RegisterRefactoring(action);
        }

    }


}
```