ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр
на тему: **«РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРАХУНКУ ОСНОВНИХ МЕТРИК ПРОГРАМНОГО КОДУ НА МОВІ C#»**

Виконав: студент 5 курсу, групи ППЗ-52
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Дяченко Максим В’ячеславович
(прізвище та ініціали)
Керівник Гаманюк І.М.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

Київ - 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення

_____ О.В. Негоденко
“ ” _____ 2021 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Дяченко Максиму В’ячеславовичу
(прізвище, ім’я, по батькові)

1.Тема роботи: **«Розробка програмного забезпечення для розрахунку основних метрик програмного коду на мові C#»,**
Керівник роботи Гаманюк Ігор Михайлович старший викладач кафедри ,
затверджені наказом вищого навчального закладу від 16.03.2021 року №65
2. Строк подання студентом роботи 01.06.2021 року
3. Вихідні дані до роботи:
Інформаційно-аналітичний метод;
Метрики програмного забезпечення;
Програмне середовище мови програмування C#;
Науково-технічна література з питань, пов’язаних програмуванням на мові C#.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
4.1 Загальний огляд метрик програмного забезпечення;
4.2 Принципи програмування на мові C#;
4.3 Практична реалізація програмного забезпечення для розрахунку основних метрик програмного коду на мові C#.
5.Перелік графічного матеріалу (назва слайдів презентації):
1. Назва роботи;
2. Мета роботи;

3. Вікно додатка Visual Studio Express;
4. Схема типів мови C#;
5. Прості (базові) типи значень;
6. Службові слова мови C#;
7. Показники цикломатичної складності програми;
8. Вікно діалогу для вибору опції меню «Новий проект»;
9. Вікно діалогу для вибору опції меню «Метрики обчислення коду»;
10. Вікно меню для перевірки циклометричної складності коду;
11. Лістинг програми для розрахунку основних метрик програмного коду на мові C#;
12. Лістинг програми для розрахунку основних метрик програмного коду на мові C#;
13. Висновки.

6. Дата видачі завдання 23.04.2021

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1.	Підбір науково-технічної літератури	23.04.2021	Викон.
2.	Загальний огляд метрик програмного забезпечення	24.04.2021	Викон.
3.	Принципи програмування на мові C#	05.05.2021	Викон.
4.	Практична реалізація програмного забезпечення для розрахунку основних метрик програмного коду на мові C#	15.05.2021	Викон.
5.	Висновки, вступ, реферат	20.05.2021	Викон.
6.	Розробка презентації	01.06.2021	Викон.

Студент Дяченко М.В.
(підпис) (прізвище та ініціали)

Керівник роботи Гаманюк І.М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 57 сторінок, 8 рисунків, 3 таблиць, 13 джерел.

Об'єкт дослідження – мова програмування C#

Предмет дослідження – основні метрики програмного коду на мові C#

Мета роботи – Розрахунок основних метрик програмного коду на мові C#

Методи дослідження – інформаційно-аналітичний метод

В роботі досліджено область програмних метрик та виявлено програмні метрики, пов'язані зі складністю коду. Проведено аналіз стосовно автоматичного вимірювання складності вихідного коду, призначення та класифікацію метрик та сфери їх застосування.

Наведено критерії оцінки метрик, а також їх максимальні або мінімальні обмеження.

Розроблено приклад програми для розрахунку основних метрик програмного коду на мові C#.

Галузь використання – інформаційні технології

МЕТРИКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МОВА C#, ТЕХНОЛОГІЯ .NET FRAMEWORK, ОБ'ЄКТНА ОРІЄНТАЦІЯ ПРОГРАМ, ТИПИ, КЛАСИ, ОБ'ЄКТИ

ЗМІСТ

ВСТУП

Під час розробки програми на будь-якій мові програмування і в будь-якому програмному середовищі великого значення набуває питання якості та складності написання програмного коду. Для вирішення цього питання будь-якому програмному середовищі з часом з'являються механізми підрахунку різних метрик. Метрикою називається міра, яка дозволяє отримати числове значення деяких властивостей програмного забезпечення, а також його характеристик. Застосування метрик у загальному випадку дозволяє керівникам проектів і підприємств визначити складність розробленого проекту або проекту, який розробляється, оцінити обсяг виконаних робіт, стилістику розроблюваної програми і зусилля, які докладені кожним розробником для реалізації того чи іншого рішення. Однак, метрики можуть слугувати лише в якості рекомендаційних характеристик, тому що на них не можна повністю покладатись. Це пов'язано з тим, що при розробці програмного забезпечення програмісти з метою мінімізації або максимізації своєї програми, можуть знижувати ефективність роботи програми. Крім того, для прикладу, написання програмістом невеликої кількості рядків коду

або внесенні невеликої кількості структурних змін не означає відсутність ним виконаної роботи а може означати, що дефект програми було дуже складно відшукати. Остання проблема, однак, частково може бути вирішена при використанні метрик складності, тому що значно складніше знайти помилку в більш складній програмі.

Широкий спектр видів діяльності пов'язаний з різними фазами розробки програмного забезпечення. Показниками програмного забезпечення є методи або формули для вимірювання певних властивостей або характеристик програмного забезпечення. У програмній інженерії термін „метрики програми" безпосередньо пов'язаний із вимірюванням. Вимірювання програмного забезпечення відіграє значну роль в управлінні програмним забезпеченням.

Норман Фентон визначає процедуру вимірювання як процес, за допомогою якого цифри або символи призначаються атрибутам сутностей у реальному світі таким чином, щоб описувати їх за чітко визначеними правилами. Метрики програмного забезпечення є кількісним показником щодо продуктивності певного програмного забезпечення стосовно людської взаємодії, необхідної для роботи програмного забезпечення. Ідея створення метрик полягає в тому, що перед тим, як щось можна виміряти або визначити кількісно, це потрібно перевести в цифри.

Існує кілька областей, де метрики програмного забезпечення є корисними. Ці сфери включають все: від планування програмного забезпечення до кроків, які мають на меті покращити продуктивність певного програмного забезпечення. Програмне забезпечення не може працювати самостійно без взаємодії людини. Тому певним чином метрика програмного забезпечення також є мірою відношення
людини до програмного забезпечення.

Саме цій актуальній темі і присвячена бакалаврська робота.

1 ЗАГАЛЬНИЙ ОГЛЯД МЕТРИК ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Використання метрик програмного забезпечення

Метрики програмного забезпечення можуть застосовуватися до кожної вимірюваної сутності програмного забезпечення. Ці метрики застосовуються протягом усього життєвого циклу розробки програмного забезпечення. На початку програмного проекту можна розробити метрики програмного забезпечення, наприклад, для визначення оцінки вартості та потреб у ресурсах. На етапі проектування можна використовувати метрики для підрахунку функціональних
балів програмного забезпечення. Вимірювання розміру програмного забезпечення - ще один аспект, який можна виміряти кількісно. Метрики, зібрані з вихідного коду, можна використовувати для вимірювання розміру програмного забезпечення. Прикладом такої метрики є рядки метрики коду, яка є найосновнішою метрикою для вимірювання розміру програмного забезпечення.

Подібним чином, метрики, зібрані за допомогою детального проекту або структури управління, можуть керувати процесом тестування програмного забезпечення. Аналіз коду може допомогти у покращенні вартості обслуговування програмного забезпечення [1].

Метрики програмного забезпечення дозволяють, за потреби, розробникам програмного забезпечення аналізувати свій код та вносити вдосконалення. Наприклад, якщо певна частина їх коду перевищує межі складності коду, вони можуть переробити його. Ефективне вимірювання може служити таким основним цілям управління, як оцінка витрат та ресурсів, оцінка прогресу та прогнози на майбутнє.

Вимірювання якості програмного забезпечення також можливе за допомогою метрик.

Метрики можна розділити за :

- розміром програмного забезпечення;

- оцінкою витрат;
- оцінкою зусиль;
- якістю програмного забезпечення;
- технічним обслуговуванням;
- моделями надійності;
- аналізом складності та аналізом дефектів;
- тестуванням програмного забезпечення.

1.2 Класифікація метрик програмного забезпечення

З точки зору різних заходів на різних етапах програмного забезпечення, метрики можна згрупувати як :

1.Метрики процесу - міри процесу розробки програмного забезпечення

2. Метрики продукту - міри атрибутів програмного продукту

3. Метрики ресурсів - метрики програмних ресурсів

Метрики програмного забезпечення, які стосуються вимірювання атрибутів процесу програмного забезпечення, згруповані як метрики процесу. Такі метрики включають, наприклад, оцінку тривалості, оцінку витрат, необхідних зусиль, якість процесу та результативність / ефективність процесу.

В залежності від бажаних показників ці метрики можна обчислити на будь-якому етапі розробки програмного забезпечення.

Одним із прикладів метрик процесу є метрика вихідного рядка коду (SLOC). Ця метрика просто підраховує рядки вихідного коду. За цим підрахунком вона може вказувати на зусилля, спрямовані на розробку цього коду.

Метрики продукту вимірювання продукту не включають лише програмний продукт, що постачається замовнику , вони також включають всі дії, здійснені в процесі розробки, наприклад, документацію та прототипи.

Для вимірювання програмного продукту існує широкий діапазон метрик. Метрики продукту - це або міра зовнішніх атрибутів продукту, або його внутрішніх атрибутів [2].

Зовнішні атрибути - міра продуктивності програмного продукту в реальному середовищі, де продукт повинен працювати. Ці заходи включають зручність використання програмного забезпечення та повторне використання, портативність та ефективність.

Прикладами внутрішніх атрибутів продукту є: розмір програмного забезпечення, коректність, складність, помилки та здатність до тестування.

Показники ресурсів и є більш доречними для менеджерів для оцінки ресурсів, необхідних для програмного проекту

Цими ресурсами є розробники, фізичні ресурси для на прикладі комп'ютерів, матеріалів та методів. Ці метрики також можна класифікувати за класом метрик процесу.

1.3 Метрики програмного коду

Метрики програмного забезпечення, які безпосередньо підраховуються з вихідного коду, називаються метриками коду.

Метрики цієї категорії зазвичай належать до класу метрик продукту. Більшість відомих метрик на даний час належить до підкласу метрик коду.

Метрики коду надають розробникам краще уявлення про код, який вони розробляють.

Користуючись перевагами метрик коду метрик коду, розробники можуть зрозуміти, які типи та / або методи слід переробити або ретельніше перевірити. Команди розробників можуть виявити потенційні ризики, зрозуміти поточний стан проекту та відстежувати прогрес під час розробки програмного забезпечення.

На основі різних заходів, що виконуються в коді, ці метрики можна додатково класифікувати за такими підгрупами:

- кількісні метрики;

-метрики складності управління потоком програм

- метрики складності потоку управління даними;

- метрики потоку управління та складності потоків даних;
- об’єктно-орієнтовані метрики;
- метрики безпеки;
- гібридні метрики.

1.3.1 Кількісні метрики

Метрики цієї групи враховують кількісні характеристики коду. Кількісними характеристиками коду можууть бути міра розміру, підрахунок загальної кількості функцій програми, кількість операцій чи операндів у програмі, кількість коментарів та середня кількість рядків на функцію.

Прикладами кількісних показників є: SLOC, Halstead Software Science, метрика ABC та метрики Jilb.

1.3.2 Метрики складності потоку програми

Кількісні показники ігнорують структуру потоку програми. Аналіз структури потоку програм або графік керування програмою надає складність потоку програми. Ці показники вимірюють складність програми на основі аналізу, проведеного за структурою управління потоком програми.

Ці метрики можуть визначати кількість необхідних тестових випадків.

Перший показник такого роду був представлений Маккейбом у 1976 році, відомий як цикломатична складність Маккейба. Цикломатична складність Мак-Кейба досі є однією з найбільш часто використовуваних метрик у багатьох комерційних та некомерційних інструментах для вимірювання складності коду. Інші метрики, основані на цій категорії, - це методологія Г. Майєрса, метод Хансена, топологічний показник Чен, метрики Гаррісона та Мейджала, міра Вудварда, метод граничного значення та метрика Шнайдвінда [5].

1.3.3 Метрики складності потоку управління даними

Ця група метрик базується на аналізі потоку управління даними програми. Ці метрики підходять для програм, керованих даними. Прикладом цього типу метрик є метрика Чепіна. Цей метод досліджує спосіб використання змінних в окремому програмному модулі та дає оцінку інформаційної потужності модуля.

Коефіцієнт зважування - це інший показник з тієї ж групи, який використовується для вимірювання складності програми. Метрика інформаційного потоку Генрі та Кафури також є мірою складності на основі зв'язків інформаційного потоку між процедурою та її середовищем [3].

1.3.4 Метрики потоку управління та складності потоку даних

Метрики цієї групи використовуються для вимірювання складності на основі кількісних властивостей та аналізу структур управління (програмного потоку та потоку даних). Цей клас метрик та метрик потоку програм також класифікується за класом топологічних метрик. Згуртованість модуля, тестування М-міри, міра Колофелло, метрика МакКлура - ось деякі приклади цього типу метрик.

1.3.5 Об'єктно-орієнтовані метрики

Почалася ера програмних метрик до концепції об'єктно-орієнтованого програмування. Різниця у стилі кодування об'єктно-орієнтованого програмування порівняно зі структурованим програмуванням підтвердила необхідність нового класу метрик, які можуть забезпечити ефективність міри складності для об'єктно-орієнтованих мов. Найчастіше використовувані метрики цієї групи - це набір метрик Мартіна, а також метрик Чідамбера та Камерера [4].

1.3.6 Метрики безпеки

Ці метрики мають схожість із кількісними метриками такими, як кількісна метрика помилок та дефектів у програмі, наприклад кількість помилок, виявлених під час перегляду коду або під час його тестування.

Метрики якості програмного забезпечення заслуговують на особливу увагу в SFP, оскільки вони можуть бути використані для вимірювання якості системи.

На основі вимірювання певних частин системи можуть бути два типи програмних метрик якості:

- статичні;
- динамічні метрики.

Статичні метрики коду підходять для перевірки атрибутів коду, таких як складність програмного забезпечення та доступ до довжини коду. Динамічні метрики коду значною мірою вивчають поведінку системи, представлену як зручність використання, надійність, ремонтопридатність та оцінка ефективності програми.

Статичні метрики коду - це тип метрик якості. Їх зручність помітна, коли йдеться про вимірювання:

- розміру (через рядки коду (LOC));
- складності (з використанням лінійно підрахованого шляху);
- легкого читання (через кількість операторів та різні операнди).

Принцип обчислення метрик статичного коду оснований на синтаксичному аналізі вихідного коду; отже, процес збору метрик автоматизований. Завдяки цій функції можна виміряти метрики всієї системи незалежно від її розміру.

Більш того, можна прогнозувати всю систему на основі метрик - розробники можуть легко знаходити несправні модулі, оскільки вони мають чітке уявлення про

вразливості системи. Статичні метричні коди простіші та широко використовуються на практиці, тому вони представляють безпечний вибір для прогнозування несправного програмного забезпечення [5].

2 ПРИНЦИПИ ПРОГРАМУВАННЯ НА МОВІ C#

2.1 Мова C# і технологія . Net Framework

Мова C # вже багато років незмінно входить в список мов програмування, найбільш затребуваних серед розробників програмного забезпечення. Мова є базовою для технології .Net Framework, розробленої і підтримуваної корпорацією Microsoft.

2.2 Історія створення мови C#

Мова C # створена інженерами компанії Microsoft в 1998-2001 роках. Керував групою розробників Андерс Хейлсберг, який до того трудився в фірмі Borland над створенням компілятора для мови Pascal і брав участь у створенні інтегрованого середовища розробки Delphi. Мова C # з'явилась після мов програмування C ++ і Java. Багатий досвід їх використання був багато в чому враховано розробниками C#.

Синтаксис мови C# схожий на синтаксис мов C ++ і Java. Але схожість зовнішня. У мови C # своя унікальна концепція. Взагалі ж з трьох мов програмування C ++, Java і C # історично першим з'явилась мова C ++, потім Java. І вже після цього з'явилася мова програмування C #.

Для розуміння місця і ролі мови C # на сучасному ринку програмних технологій розумно почати з мови програмування C, який свого часу став потужним стимулом для розвитку програмних технологій як таких. Саме з мови C зазвичай виводять генеалогію мови C# [6].

Мова програмування С з'явилася в 1972 році, його розробив Денис Рітчі. Мова С поступово набрала популярність і в підсумку стала однією з найбільш затребуваних мов програмування. Цьому сприяв ряд обставин. В першу чергу, звичайно, зіграв роль лаконічний і простий синтаксис мови. Та й загальна концепція мови С виявилася виключно вдалою і живучою. Тому коли постало питання про розробку нової мови, який би підтримував
парадигму об'єктно орієнтованого програмування (ООП), то вибір природним чином упав на мову С: мова програмування С ++, що з'явилася в 1983 році, представляла собою розширену версію мови С, адаптовану для написання програм із залученням класів, об'єктів і супутніх їм технологій. У свою чергу, при створенні мови програмування Java відправною точкою
стала мова С ++. Ідеологія мови Java відрізняється від ідеології мови С ++, але при цьому базові керуючі інструкції та оператори в обох мовах схожі.

Мова програмування Java офіційно з'явилася в 1995 році і стала популярною завдяки універсальності програм, написаних на цій мові. Технологія, використовувана в Java, дозволяє писати переносні програмні коди, що виключно важливо при розробці додатків для використання в Internet.

2.3 Особливості мови C#

Мова програмування С # - проста, красива, ефективна і гнучка. За допомогою програм на мові С # вирішують найрізноманітніші завдання. На С # можна створювати як невеликі консольні програми, так і програми з графічним інтерфейсом. Код, написаний на мові С #, лаконічний і зрозумілий (хоча тут, звичайно, багато що залежить від програміста). В цьому відношенні мова програмування С # практично не має конкурентів.

Крім власне переваг мови С #, важливо і те, що мова підтримується компанією Microsoft. Тому вона ідеально підходить, щоб писати програми для виконання під управлінням операційної системи Windows. Операційною системою Windows і технологією .Net Framework компанії Microsoft область

застосування мови C # не обмежується. Існують альтернативні проекти (такі, наприклад, як Mono), що дозволяють виконувати програми, написані на мові C #, під управлінням операційних систем, відмінних від Windows [7-9].

Як зазначалося вище, мова C # є невід'ємною частиною технології (або платформи) .Net Framework. Основу платформи .Net Framework становить середовище виконання CLR (Common Language Runtime) і бібліотека класів, яка використовується при програмуванні на мові C #.

Платформа .Net Framework дозволяє використовувати і інші мови програмування, а не тільки C # - наприклад, C ++ або Visual Basic. Можливості платформи .Net Framework дозволяють об'єднувати «в одне ціле» програмні коди, написані на різних мовах програмування.

При компіляції програмного коду, написаного на мові C #, створюється проміжний код. Це проміжний код реалізований на мові MSIL (Microsoft Intermediate Language). Проміжний код виконується під управлінням системи CLR. Система CLR запускає JIT-компілятор (Just In Time), який, власне, і переводить проміжний код у виконувані інструкції.

Файл з програмним кодом на мові C # зберігається з розширенням .cs. Після компіляції програми створюється файл з розширенням .exe. Але виконати цей файл можна тільки на комп'ютері, де встановлена система .Net Framework. Такий код називається керованим (оскільки він виконується під управлінням системи CLR).

Описана нетривіальна схема компілювання програм із залученням проміжного коду служить важливої мети. Справа в тому, що технологія .Net Framework орієнтована на спільне використання програмних кодів, написаних на різних мовах програмування. Базується ця технологія на тому, що програмні коди з різних мов програмування «переводяться» (в процесі компіляції) в проміжний код на загальній універсальній мові. Простіше кажучи, коди на різних мовах програмування наводяться «до спільного знаменника», яким є проміжна мова MSIL [8].

Програми, написані на Java, теж компілюються в проміжний байт-код. Байт-код виконується під управлінням віртуальної машини Java. Але в порівнянні з мовою C # є принципова відмінність. Байт-код, в який переводиться при компіляції Java-програма, має прив'язку до однієї мови програмування – мови Java. І в Java схема з проміжним кодом потрібна для забезпечення універсальності програм, оскільки проміжний байт-код не залежить від типу операційної системи (і тому переносний). Особливість операційної системи враховується тією віртуальною машиною Java, яка встановлена на комп'ютері і виконує проміжний байт-код. Проміжний код, який використовується в технології .Net Framework, не прив'язаний до конкретної мови програмування. Наприклад, що при компіляції програми на мові C#, що при компіляції програми на мові Visual Basic виходять набори інструкцій на одній і тій ж проміжній мові MSIL. І потрібно це, ще раз підкреслимо, для забезпечення сумісності різних програмних кодів, реалізованих на різних мовах.

Перше, що варто відзначити: мова C # повністю об'єктно орієнтованою. Це означає, що навіть найменша програма на мові C # повинна містити опис хоча б одного класу.

Як зазначалося вище, базові синтаксичні конструкції мови C # нагадують (або просто збігаються) відповідні конструкції в мовах C ++ і / або Java. Але, крім них, мова C # містить безліч цікавих і корисних особливостей [10].

2.4 Програмне забезпечення мови C#

Для роботи з мовою програмування C # на комп'ютері повинна бути встановлена платформа .Net Framework. Компілятор мови C # входить в стандартний набір цієї платформи. Тому, якщо на комп'ютері встановлена платформа .Net Framework, цього достатньо для початку програмування в C #. Зазвичай додатково ще встановлюють середовище розробки, завдяки якому процес програмування на C # стає простим і приємним.

Якщо ми хочемо написати програму (на мові C #), спочатку нам потрібно набрати відповідний програмний код. Теоретично зробити це можна в звичайному текстовому редакторі. В такому випадку набираємо в текстовому редакторі код програми і зберігаємо файл з розширенням .cs (Розширення для файлів з програмами, написаними на мові C #). Після того як код програми набраний і файл з програмою збережений, її слід відкомпілювати. Для цього використовують програму-компілятор csc.exe, яка встановлюється, як зазначено вище, при установці платформи .Net Framework.

Алгоритм дій такий: в командному рядку вказується назва програми-компілятора csc.exe, а потім через пробіл вказується назва файлу зпрограмою на мові C #. Припустимо, ми записали код програми в файл MyProgram.cs. Тоді для компіляції програми в командному рядку використовуємо інструкцію csc.exe MyProgram.cs або csc MyProgram.cs (розширення exe-файла можна не вказувати). Якщо компіляція проходить нормально, то в результаті отримуємо файл з розширенням .exe, а назва файлу збігається з назвою вихідного файлу з програмою (в нашому випадку це MyProgram.exe). Отриманий в результаті компіляції exe-файл запускають на виконання.

Файл csc.exe за замовчуванням знаходиться в каталозі C: \ Windows \ Microsoft.NET \ Framework всередині папки з номером версії - наприклад, v3.5 або v4.0. Також для компілювання програми з командного рядка доведеться, швидше за все, виконати деякі додаткові налаштування - наприклад, в змінних середовища задати шлях для пошуку компілятора csc.exe [9].

Хоча така консольна компіляція цілком робоча, вдаються до неї рідко. Причина в тому, що не дуже зручно набирати код в текстовому редакторі, потім компілювати програму через командний рядок і запускати вручну виконавчий файл. Набагато простіше скористатися спеціальною програмою - інтегрованої середовищем розробки (IDE від Integrated Development Environment). Середовище розробки містить в собі всі найбільш важливі «інгредієнти», необхідні для «приготування» такої «страви», як програма на мові C #. При роботі з середовищем розробки користувач отримує в одному комплекті редактор кодів,

засоби налагодження, компілятор і ряд інших ефективних утиліт, що значно полегшують роботу з програмними кодами.

Завантажити файли можна з сайту компанії Microsoft www.microsoft.com (на сайті слід знайти сторінку завантажень). Після установки середовища розробки ми отримуємо все, що потрібно для

успішної розробки додатків на мові C #.

Як виглядає вікно програми Visual Studio Express (версія 2015), показано на рис. 2.1.

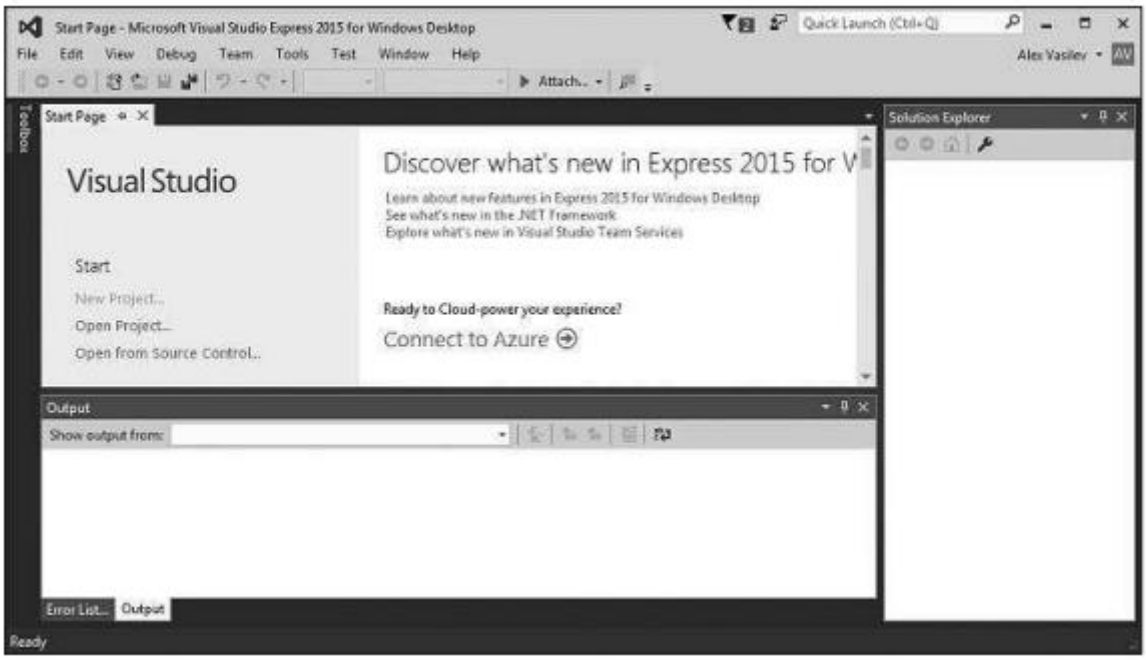


Рисунок 2.1 - Вікно додатка Visual Studio Express 2015

Середовище розробки Visual Studio хоча і краща, але далеко не єдина. Існують інші програми, які використовуються при створення програм на мові C#. Причому масштаби використання альтернативних засобів розробки постійно розширюються. Але, в будь-якому випадку, має сенс знати, які є варіанти крім Visual Studio.

Середовище розробки SharpDevelop є прийнятним вибором і дозволяє створювати додатки різного рівня складності. По суті, дане середовище розробки є інтегрованим відладчиком, що взаємодіє з середовищем .Net Framework. Вікно середовища розробки SharpDevelop показано на рис. 2.2.

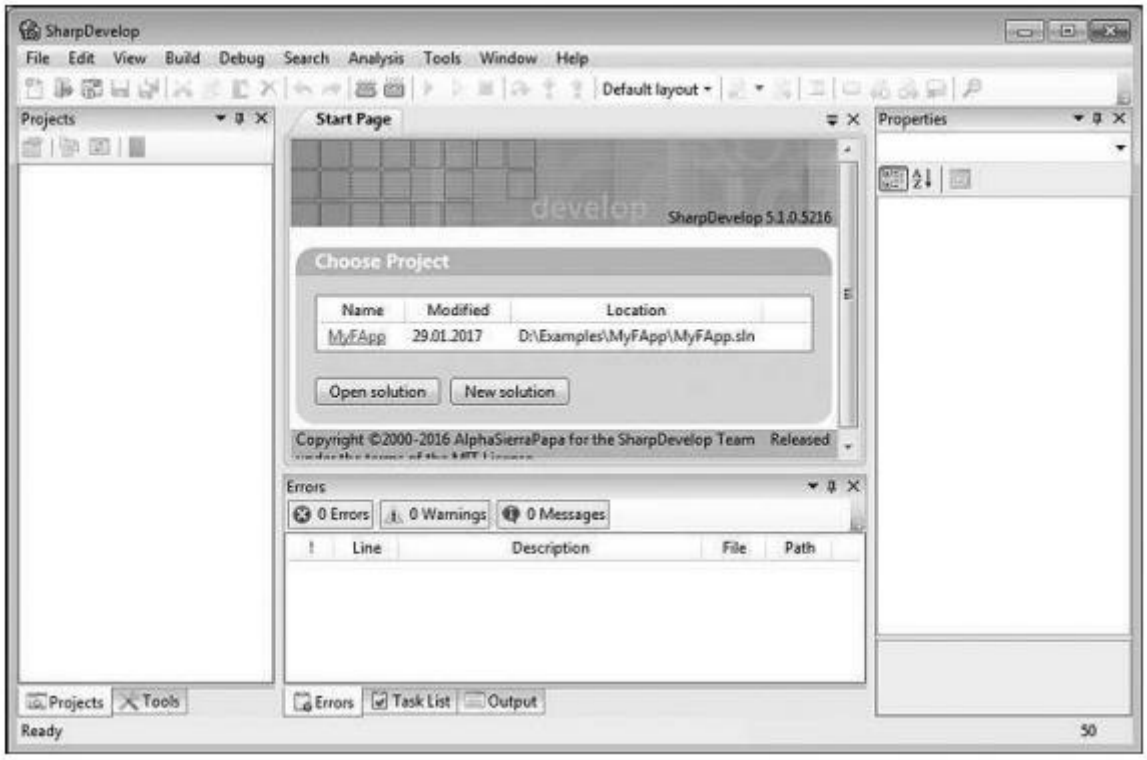


Рисунок 2.2 - Вікно середовища розробки SharpDevelop

Ще один проект, який заслуговує на увагу, називається Mono. Проект розвивається в основному завдяки підтримці компанії Xamarin. В рамках цього проекту була створено середовище розробки MonoDevelop, призначене, крім іншого, для створення додатків на мові C#. На даний момент розробникам для установки пропонується середовище розробки Xamarin Studio, яке позиціонується як продовження проекту MonoDevelop.

Слід зазначити, що середовища розробки не є еквівалентними за своїми функціональними можливостями. Найбільш надійний варіант пов'язаний з використанням середовища розробки Microsoft Visual Studio. Потрібно мати на увазі, що в силу специфіки середовищ розробки SharpDevelop і Xamarin Studio деякі програмні коди, виконувані в Microsoft Visual Studio, можуть не виконуватися в SharpDevelop і Xamarin Studio (і навпаки) [11].

Середовища розробки Microsoft Visual Studio і SharpDevelop орієнтовані на використання операційної системи Windows. Середовище розробки Xamarin

Studio дозволяє працювати з операційними системами Windows, Linux і Mac OS X.

Вікно середовища розробки Xamarin Studio показано на рис. 2. 3.

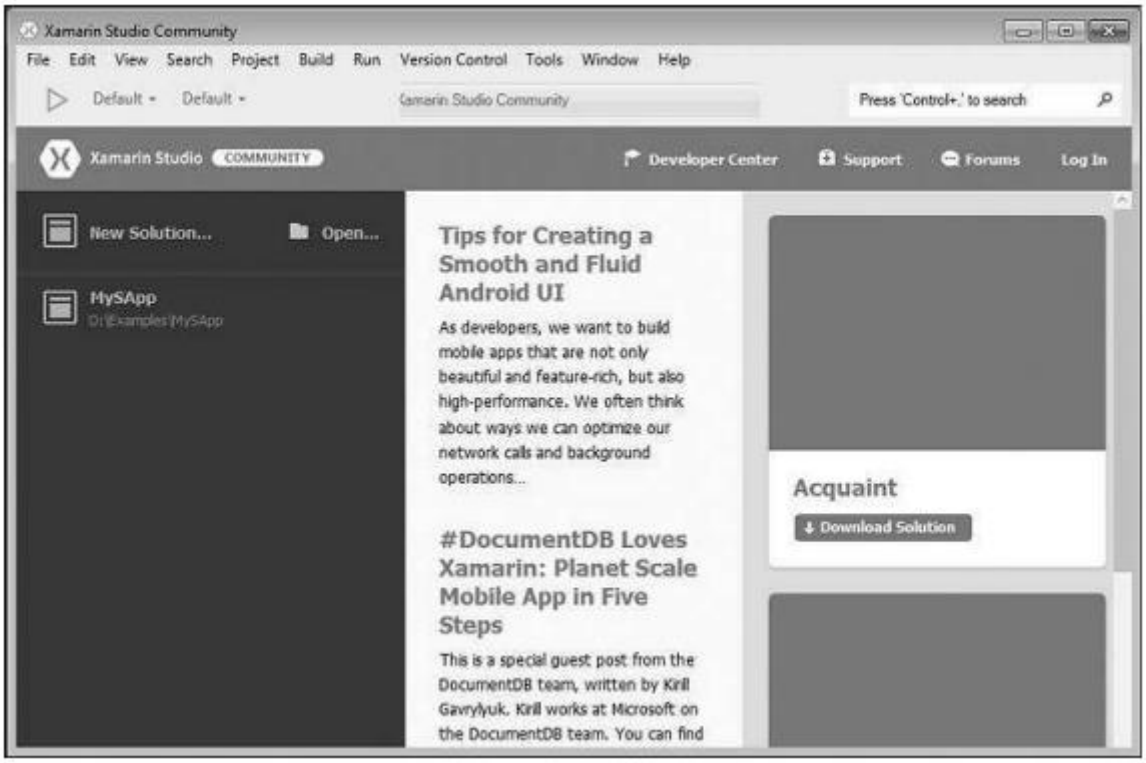


Рисунок 2.3 - Вікно середовища розробки Xamarin Studio

2.5 Об'єктна орієнтація програм на C#

2.5.1 Типи, класи, об’єкти

Мова C# - це об'єктно-орієнтована мова зі строгою типізацією.

В об'єктно-орієнтованих мовах все є об'єктом - від констант і змінних базових типів до даних агрегованих типів будь-якої складності.

Тип в програмуванні поняття первинне. Тип деякої сутності декларує для неї сукупність можливих станів і набір допустимих дій. Сутностями можуть бути константи, змінні, масиви, структури тощо.

Найбільш часто поняття тип в мовах програмування використовують в зв'язку з поняттям «змінна».

Примітивне визначення: змінна це пара «позначення змінної + значення змінної».

Для змінної тип вводить сукупність її можливих значень і набір допустимих операцій над цими значеннями.

Приклад визначення змінної в Сі, Сі ++, C# і деяких інших мовах:

int spot = 16

spot - позначення (ім'я) змінної, 16 - її значення в даний момент (після цього визначення), int - назва типу змінної. Цей тип int визначає для змінної spot граничні значення, сукупність допустимих значень і набір операцій з правилами їх виконання. Наприклад, для spot визначена операція отримання залишку від ділення (%) і особливим чином визначено розподіл на цілу величину (/ в цьому випадку позначає операцію цілочисельного ділення). Результат spot/5 дорівнює 3, а значенням виразу spot% 3 буде 1.

Типи в мові C# введені за допомогою класів (а також структур, перерахувань, інтерфейсів, делегатів і масивів).

Поняття класу в теоретичних роботах, присвячених об'єктно-орієнтованої методології, зазвичай вводиться на основі поняття «об'єкт». Об'єкт визначають по-різному.

«Об'єктом називається сукупність даних (полів), що визначають стан об'єкта, і набір функцій (методів), що забезпечують зміну зазначених даних (стану об'єкта) і доступ до них».

«Об'єкт - інкапсуляція безлічі операцій (методів), доступних для зовнішніх викликів, і стану, що запам'ятовує результати виконання зазначених операцій»

Після подібного визначення в посібниках з об'єктно-орієнтованого програмування перераховуються обов'язкові ознаки об'єктів:

1. розрізнення;

2. можливість одного об'єкта перебувати в різних станах (в різний час);

3. можливість динамічного створення об'єктів;

4. «вміння» об'єктів взаємодіяти один з одним за допомогою обмінів повідомленнями;

5. наявність методів, що дозволяють об'єкту реагувати на повідомлення (на зовнішні для об'єкта впливи);

6. інкапсуляція даних усередині об'єктів.

Ввівши поняття об'єкта, клас визначають як механізм, що задає структуру (поля даних) всіх однотипних об'єктів і функціональність об'єктів, тобто механізм, що визначає всі методи, які стосуються об'єктів.

В процесі розвитку об'єктно-орієнтованого підходу і при його реалізації в мовах програмування стало ясно, що серед даних об'єкта можуть існувати такі, які належать не одиничному об'єкту, а всім об'єктам класу. Те ж було виявлено та для методів - деякі з них могли визначати не функціональність окремого (кожного) об'єкта, а бути загальними для класу (для всіх його об'єктів). У сукупності поля і методи як класу, так і формуючих з його допомогою об'єктів називаються членами класу [12].

Для ілюстрації цих понять і особливо відмінностей полів і методів класу від полів і методів його об'єктів розглянемо приклад: Клас з назвою «студент групи N-го курсу».

поля (дані) об'єкта: ПІБ, оцінки за сесію, взяті в бібліотеці книги і т.д.

методи об'єкта: скласти іспит, отримати книги в бібліотеці і т.д.

поля (дані) класу: номер курсу (N), дати іспитів, кількість дисциплін в семестрі і т.д.

метод класу: перевести групу на наступний курс - зміняться всі дані класу, але не всі об'єкти залишаться в цьому зміненому класі (не обов'язково всі студенти будуть переведені на наступний курс).

Різниця між даними і методами об'єктів і даними і методами їх класу істотно використовується в мові C#. Щоб їх розрізняти в визначенні (в

оголошенні) класу його дані і його методи забезпечуються спеціальним модифікатором static (статичний).

Отже, клас грає дві ролі:

- клас це контейнер для методів класу і даних класу;
- клас це «трафарет», що дозволяє створювати конкретні об'єкти.

Для кожного конкретного об'єкта, клас визначає структуру його стану і поведінку. Стан об'єкта задається сукупністю значень його полів. Поведінка об'єкта визначається набором методів, що відносяться до об'єктів даного класу.

Відповідно до об'єктної орієнтації мови C# - будь-яка програма на мові C # являє собою клас або сукупність класів.

Усередині оголошення кожного класу можуть бути розміщені:

1. дані класу (статичні поля);
2. методи класу (статичні методи);
3. дані об'єктів класу (не статичні поля);
4. методи для роботи з об'єктами класу (не статичні методи);
5. внутрішні класи;
6. додаткові члени, мова про які ще попереду.

2.5.2. Програма C#

Формат найпростішого визначення (інакше декларації або оголошення) класу в C#:

class ім’я_класу

{поля і методи}

Тут class - службове слово. Заключена в обов'язкові фігурні дужки сукупність полів і методів називається тілом класу. Серед полів і методів можуть бути статичні, що відносяться до класу в цілому, і не статичні - визначальні стану конкретних об'єктів і дії над цими об'єктами.

Перед тілом класу - знаходиться заголовок оголошення класу. Заголовок в загальному випадку може мати більш складну форму, але зараз її не потрібно розглядати. Службове слово class авжди входить в заголовок.

ім'я_класу - ідентифікатор, довільно обираний автором класу.

Відзначимо, що ідентифікатор в мові C# - це послідовність букв, цифр і символів підкреслення, що не може починатися з цифри. На відміну від багатьох попередніх мов в ідентифікаторах C # можна використовувати літери різних алфавітів, наприклад, російського чи грецького. У мові C# прописна буква відрізняється від тієї ж самої малої.

Серед методів класів здійсненної програми (додатки) на мові C# обов'язково присутній статичний метод зі спеціальним ім'ям Main. Цей метод визначає точку входу в програму - саме з виконання операторів методу Main () починається виконання її коду. Виконуюча програму система неявно (невидимо для програміста) створює єдиний об'єкт класу, що представляє програму, і передає управління коду методу Main ().

Перш ніж наводити приклади програм, необхідно відзначити, що практично кожна програма на мові C # активно використовує класи бібліотеки з .NET Framework. Через бібліотечні класи програмі доступне те оточення, в якому вона виконується. Наприклад, клас Console представляє в програмі засоби для організації консольного діалогу з користувачем.

Застосування в програмі бібліотеки класів передбачає або створення об'єктів класів цієї бібліотеки, або звернення до статичних полів і методів бібліотечних класів.

Щоб застосовувати методи і поля бібліотечних класів і створювати їх об'єкти, необхідно знати склад бібліотеки і можливості її класів. Бібліотека .NET настільки велика, що на першому етапі вивчення програмування на C# доведеться обмежитися тільки самими скромними відомостей про неї.

Для ілюстрації наведених загальних відомостей про програми на C # розглянемо програму, яка виводить в консольне вікно екрану фразу «Введіть Ваше ім'я:», зчитує ім'я, набирається користувачем на клавіатурі, а потім вітає

користувача, використовуючи отримане ім'я.

```
//01_01.cs – Перша програма.
class helloUser
{
    static void Main()
    {
    string name;
    System.Console.WriteLine("Введіть Ваше ім'я:");
    name =  System.Console.ReadLine();
    System.Console.WriteLine("Вітаю Вас, name+"!");
    }
}
```

Перший рядок – однорядковий коментар.

Другий рядок class helloUser - це заголовок визначення класу з ім'ям helloUser. Нагадаємо, що class – службове слово, а ідентифікатор helloUser вибрав автор програми.

Далі в фігурних дужках - тіло класу.

У класі helloUser тільки один метод з заголовком static void Main ()

Як уже сказано, службове слово static - це модифікатор методу класу (який відрізняє його від методів об'єктів). Службове слово void визначає тип, відповідний особливому випадку «відсутність значення». Його використання в заголовку означає відсутність повертаючого методом Main() значення. У заголовку кожного методу після його імені в круглих дужках міститься список параметрів (специфікація параметрів). У нашому прикладі параметри у методу не потрібні, але круглі дужки обов'язкові. Відзначимо, що ім'я Main не є службовим словом C #.

Слідом за заголовком у визначенні кожного методу поміщається його тіло - укладена в фігурні дужки послідовність визначень, описів і операторів. Розглянемо тіло методу Main () в нашому прикладі.

string name; - це визначення (декларація) строкової змінної з обраним програмістом ім'ям name. string - службове слово мови C # - позначення передбаченого типу (System.String) для подання рядків. Детальніше про типи йтиметься нижче.

Для виведення інформації в консольне вікно використовується оператор:

System.Console.WriteLine("Введіть Ваше ім'я:");

Це звернення до статичного методу WriteLine() бібліотечного класу Console, що представляє в програмі консоль. System - позначення простору імен (namespace), до якого віднесено клас Console.

Метод WriteLine () класу Console виводить значення свого аргументу в консольне вікно. У цього методу немає повертаючого значення, але є аргумент. В даному зверненні до методу WriteLine() аргументом служить строкова константа "Введіть Ваше ім'я:". Її значення - послідовність символів, розміщена між лапок. Саме це повідомлення як «запрошення» побачить користувач в командному вікні при виконанні програми [11].

Текст із запрошенням і подальшою відповіддю користувача для нашої програми можуть мати, наприклад, такий вигляд:

Введіть Ваше ім'я:

Тимофій<ENTER>

Тут <ENTER> - умовне позначення (доданий на папері, але відсутнє на консольному екрані) натискання користувачем клавіші ENTER.

Коли користувач набере на клавіатурі деяке ім'я (деякий текст) і натисне клавішу ENTER, то будуть виконані дії, відповідні оператору

name = System.Console.ReadLine();

Тут спочатку викликається метод ReadLine() класу Console з простору імен System. Метод ReadLine() не має аргументів, але у нього є значення, що повертається - рядок символів, прочитаний з стандартного вхідного потоку консолі. В нашому прикладі значення, що повертається - рядок "Тимофій".

Метод ReadLine() виконається тільки після натискання клавіші ENTER. До тих пір користувач може вносити в текст, що набирає будь-які виправлення та зміни.

В даному операторі зліва від знака операції присвоювання = знаходиться ім'я name тієї змінної, якій буде присвоєно отримане від консолі значення. Після наведеного вище діалогу значенням name буде "Тимофій".

Наступний оператор містить звернення до вже знайомого методу:

System.Console.WriteLine("Вітаю Вас, name+"!");

Як аргумент використовується конкатенація (зчеплення, з'єднання) трьох рядків:

1. строкової константи "Вітаю Вас,";
2. строкової змінної з ім'ям name;
3. строкової константи "!".

Як позначення операції конкатенації рядків використовується символ +. (У виразах з арифметичними операндами знак + означає операцію додавання. Ця особливість операцій по-різному виконуватися для різних типів операндів називається поліморфізмом. При конкатенації в нашому прикладі значення строкових констант «приєднуються» до значення змінної з ім'ям name.

Таким чином, результат виконання програми буде таким:

Введіть Ваше ім'я:

Тимофій<ENTER>

Вітаю Вас, Тимофій!

Для продовження натисніть будь-яку клавішу…

Останню фразу додає середовище виконання програм при завершенні консольного застосування. Натискання будь-якої клавіші призводить до закриття консольного вікна. Якщо додаток виконується поза інтегрованого середовища, тобто з консольного рядка, то фраза «Для продовження натисніть будь-яку клавішу» буде відсутньою.

2.5.3 Простір імен

«Простір імен - механізм, за допомогою якого підтримується незалежність використовуваних в кожній програмі імен і виключається можливість їх випадкового взаємовпливу »

«Простір імен визначає декларативну область, яка дозволяє окремо зберігати безлічі імен. По суті, імена, оголошені в одному просторі імен, які не будуть конфліктувати з такими ж іменами, оголошеними в іншому.»

Кожна програма на C # може використовувати або свій власний унікальний простір імен, або розміщує свої імена в просторі, що надається програмою за замовчуванням.

Поняття простору імен з'явилося в програмуванні в зв'язку з необхідністю розрізняти однойменні поняття з різних бібліотек, використовуваних в одній програмі. Простір імен System об'єднує ті класи з .NET Framework, які найбільш часто використовуються в консольних програмах на C#.

Якщо в програмі необхідно багаторазово звертатися до класів з одного і того ж простору імен, то можна спростити складові імена, використовуючи на початку програми (до визначення класу) спеціальний оператор:

using ім’я _простір_імен;

Після такого оператора для звернення до статичного члену класу з даного простору імен можна використовувати скорочене кваліфіковане ім'я

ім’я_класу. ім’я_члена

У нашій програмі використовуються: простір імен System, з цього простору - клас Console і два статичних методи цього класу WriteLine () і ReadLine ().

Помістивши в програму оператор

using System;

можна звертатися до названих методів за допомогою скорочених складових імен Console.WriteLine() і Console.ReadLine ().

2.5.4 Створення консольного додатку

На відміну від мов попередніх поколінь, мову C# неможливо застосовувати, вивчивши лише синтаксис і семантику мовних конструкцій. Навіть такі елементарні дії як введення-виведення тестової інформації вимагають застосування механізмів, що входять в мову програмування. Ці механізми надаються програмісту в вигляді засобів платформи .NET. Платформа .NET підтримує не тільки мову C#, але і десятки інших мов, надаючи їм величезну бібліотеку класів, що спрощують розробку програм різного призначення. .NET дозволяє в одному програмному комплексі об'єднувати програми, написані на різних мовах програмування. .NET в даний час реалізована для різних операційних систем. Для .NET розроблені потужні і найбільш сучасні системи (середовища) програмування. Назвемо дві з них.

Фірма Microsoft (розробник продукту .NET Framework) пропонує програмістам середу програмування Visual Studio. Корпорація Borland випускає систему програмування Borland Developer Studio (BDS). Крім комерційних продуктів із зазначеним найменуванням обидві фірми випускають такі вільно поширювані (безкоштовні) системи програмування, можливостей яких цілком достатньо для вивчення програмування на мові C#:

Visual C# Express Edition

Turbo C# Explorer

Програмуючи на C # в .NET Framework, можна зокрема розробляти:

• консольні додатки;

• Windows-додатки;

• бібліотеки класів.

Незалежно від того, якого виду програма розробляється, в Visual Studio необхідно створити рішення (Solution) і в цьому рішенні проект (Project). Створення порожнього (без проектів) рішення не має сенсу, тому рішення буде автоматично створено при створенні нового проекту. Перш ніж описати послідовність дій, необхідних для створення та виконання простої програми, зупинимося на співвідношенні понять проект і рішення. В одне рішення можуть одночасно входити проекти програм різних видів. Текст (код) програми може

бути оброблений середовищем Visual Studio, коли він поміщений в проект, а проект - включений в рішення. Найчастіше в одне рішення поміщають взаємопов'язані проекти, наприклад, використовують одні і ті ж бібліотеки класів (також поміщені у вигляді проектів в це рішення).

Як тільки середовище розробки запущене, необхідно виконати наступні кроки.

1. Створення нового проекту.

File → New → Project

У вікні New Project на лівій панелі (Project Types) необхідно обрати мову (Visual C#) і платформу (Windows). На центральній панелі вибрати вид додатка Console Application.

В поле Name замість пропонованого за замовчуванням імені ConsoleApplication1 треба надрукувати вибране ім'я проекту, наприклад Program_1. В поле Location ввести повне ім'я папки, в якій буде збережено рішення, наприклад, C:\ Програми. За замовчуванням рішенням приписується ім'я його першого проекту (в нашому прикладі Program_1). Кнопкою ОК запускається процес створення проекту (і рішення).

Середа Visual Studio 2010 створює рішення, проект програми та відкриває вікно редактора з таким текстом заготовки для коду програми:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Program_1
{
    class Program
   {
        static void Main(string[] args)
   }
}
```

2. Незважаючи на те, що ніякого коду в проект ми не додавали, це додаток цілком працездатний. Його можна наступним чином запустити на компіляцію і виконання:

Debug → Start Without Debuggind

(або поєднання клавіш Ctrl + F5)

Відкриється консольне вікно з єдиною фразою:

«Для продовження натисніть будь-яку клавішу…»

Це повідомлення середовища розробки, що завершає виконання консольного додатку.

3. Доповнимо створену середовищем розробки заготовку коду консольного застосування операторами з першої програми:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Program_1
{
    class Program
   {
        static void Main(string[ ] args)
        {
        string name;
        System.Console.WriteLine("Введіть Ваше ім'я: ");
        name = System.Console.ReadLine( );
        System.Console.WriteLine("Вітаю Вас, "
              + name + "!");
        }
    }
```

Тепер після натискання клавіш Ctrl + F5 програма відкомпілюється, почне виконання, виведе запрошення: «Введіть Ваше ім'я: "і, у відповідь на введене ім'я,« привітається ».

На відміну від першої програми 01_01.cs в тексті заготовки, створеної середовищем Visual Studio 2010 присутні зайві для нашої програми оператори. По-перше, замість чотирьох операторів using, можна обійтися одним using System ;.

По-друге, немає необхідності в явній вказівці параметра в заголовку методу:

static void Main(string[ ] args)

Конструкція string [] args ніяк не використовується в нашому додатку і може бути видалена.

Третя особливість заготовки - наявність оголошення простору імен:

namespace Program_1 {…}

Це оголошення вводить для програми власний простір імен з позначенням Program_1. Програма 01_01.cs не містить такого оголошення і тому використовує стандартний простір імен.

2.6 Типи в мові C#

2.6.1 Типи посилань і типи значень

У стандарті C# в зв'язку з типами використовується вираз «уніфікована система типів». Сенс уніфікації полягає в тому, що всі типи походять від класу object, тобто є похідними від цього класу і наслудують його члени.

Типи C# дозволяють представляти, по-перше, ті «скалярні» дані, які використовуються в розрахунках (цілі і дійсні числа, логічні значення) і в обробці текстів (символи, рядки). Друга група типів відповідає специфічним для програмування на мовах високого рівня «агрегуючим» конструкціям: масивам, структурам, об'єктам (класам).

Такий поділ типів на дві названі групи успадковано мовою C# з попередніх йому мов програмування.

Однак, при розробці C# вирішили, що в системі типів доцільно мати ще один поділ. Тому мова C # підтримує два види (дві категорії) типів: типи значень (value types) і типи посилань (reference types).

Принципова відмінність цих двох видів типів полягає в тому, що об'єкт посилального типу можна посилатись як одночасно декількома посиланнями, що абсолютно неприпустимо для об'єктів з типами значень.

Для змінних традиційних мов, наприклад Сі, завжди дотримується однозначна відповідність:

ім’я_змінної → значення_змінної

Точно така ж схема відношення справедлива в мові C # для об'єктів з типами значень:

ім’я_об’єкта → значення_об’єкта

Якщо розглядати реалізацію такого відношення в програмі, то потрібно згадати, що пам'ять комп'ютера організована у вигляді послідовності осередків. Кожен осередок має індивідуальну, зазвичай числову адресу (найменший з адресованих осередків - байт).

При виконанні програми кожному об'єкту виділяється блок (ділянка) пам'яті у вигляді одного або декількох суміжних байтів. Адреса першого з них вважається адресою об'єкта. Код, що знаходиться в виділеному для об'єкта блоці пам'яті, представляє значення об'єкта.

Уявити машинну реалізацію об'єкта з типом значень можна так:

адреса_об’єкта ⇒ код_значення_об’єкта

Змінні, які мають типи значень, безпосередньо представляють в програмі конкретні дані.

Змінні, які мають типи посилань, представляють в програмі конкретні дані побічно, хоча опосередкованість цього уявлення не відображено явно в тексті програми. Доступ до даних по імені змінної з типом посилання ілюструє тріада:

Машинну реалізацію такої тріади можна зобразити так:

адреса_змінної ⟹ код_адреси_даних ⟹ код_значення_даних

Однак при програмуванні доступ до даних за допомогою посилання можна сприймати відповідно до схеми доступу до даних за допомогою традиційної змінної (що має тип значень):

ілюструє тріада:

ім’я_посилання ⟹ значення_даних

Але при використанні такої схеми з'являється нова і принципова відмінність - доступність одних і тих же даних (однієї ділянки пам'яті) за допомогою декількох посилань (рис.2.4):

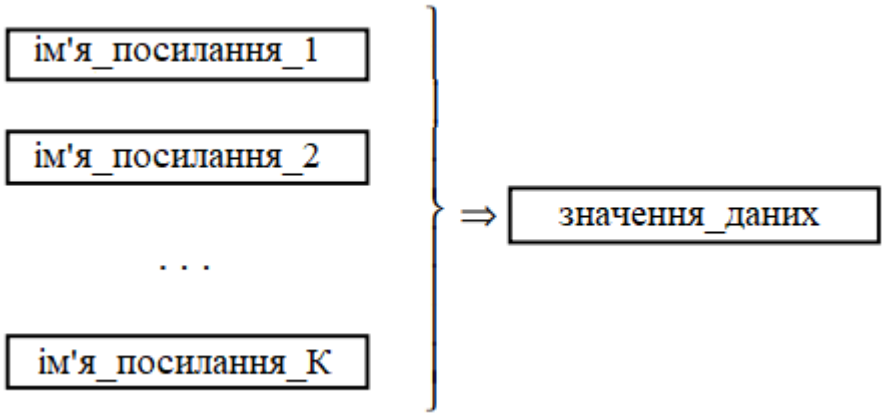


Рисунок 2.4 - Доступ до даних за допомогою декількох посилань

Основна і принципова для програміста-користувача відмінність типів значень від типів посилань складається в наступному. Кожної змінної, яка має тип значень, належить її власна копія даних, і тому операції з однією змінною не впливають на значення інших змінних.

Кілька змінних з типом посилань можуть бути одночасно співвіднесені з одним і тим же об'єктом. Тому операції, що виконуються з однією з цих змінних,

можуть змінювати об'єкт, на який в цей момент посилаються інші змінні (з типом посилань).

Відмінності між типами значень і типами посилань ілюструє ще одна особливість. Об'єкт посилального типу ніколи не має свого власного імені.

Якщо звернути увагу на принципи організації пам'яті комп'ютера, то слід зазначити, що на логічному рівні вона розділена на дві частини: стек і керовану пам'ять - "купу" (manager heap).

Об'єкти з типами значень, як такі, завжди при реалізації отримують пам'ять в стеку. При привласненні їх значення копіюються. Об'єкти посилальних типів розміщуються в купі [12].

Як і об'єкти, змінні можуть бути посилальних типів (посилання) і типів значень.

Можна сказати, що типи значень - це ті типи, змінні яких безпосередньо зберігають свої дані, тоді як посилальні типи - це ті типи, змінні яких зберігають посилання, за якими відповідні дані можуть бути доступні.

2.7 Класифікація типів C#

Загальне відношення між типами ілюструє ієрархічна схема, наведена на рис. 2.5. Як уже згадувалося і як показано на схемі, всі типи мови C# мають загальний базовий тип - клас object.

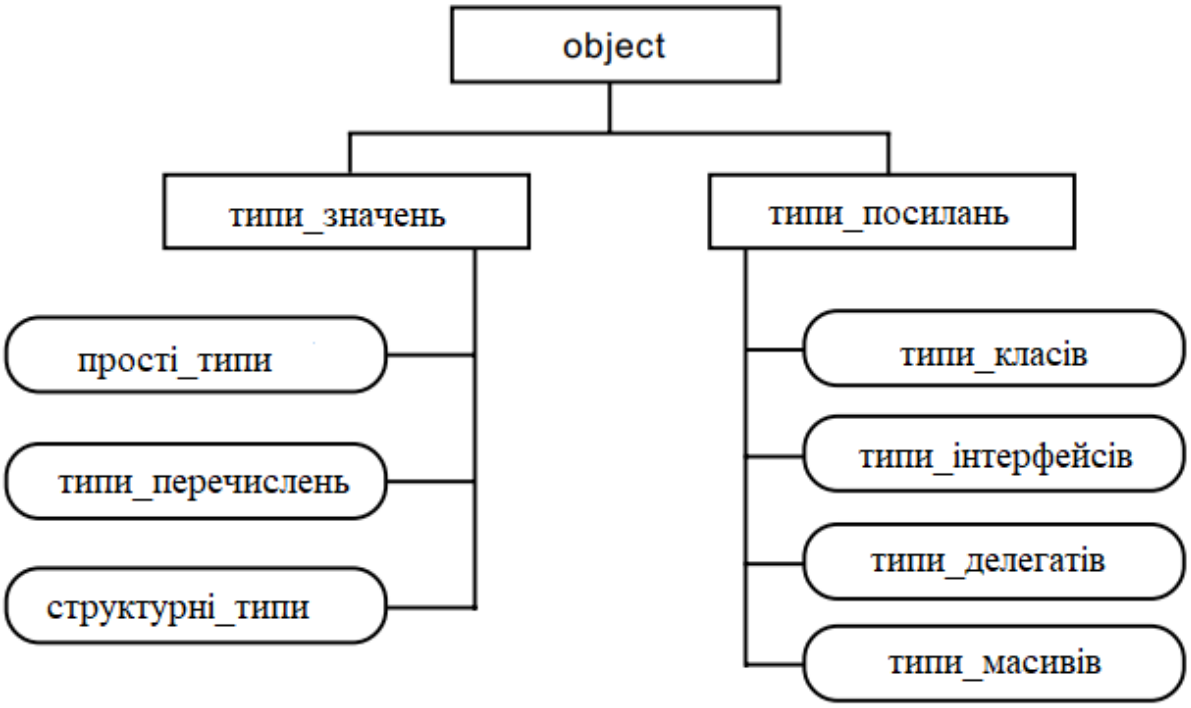


Рисунок 2.5 - Схема типів мови C#

Всі типи, які можуть використовуватися в програмах на C#, діляться на три групи:

- обумовлені в мові C# (в Стандарті вони позначені терміном Built_In, який можна перекласти як «вбудовані»);
- бібліотечні (зазвичай зі стандартної бібліотеки .NET Framework);
- визначені програмістом (користувацькі).

Обумовлені типи завжди включені в мову C#. До них відносяться:

1. object - тип посилань (клас), який є початковим (єдиним вихідним) базовим для всіх інших типів мови C#, тобто всі інші типи є похідними від цього типу;

2. прості (базові або фундаментальні) типи;

3. string - тип посилань (клас) для подання рядків - послідовностей символів в кодуванні Unicode ...

Бібліотечні і користувацькі типи можуть бути як типами значень, так і типами посилань. Щоб користуватися бібліотечним типом, потрібно знати його ім'я і можливості (поля, методи), а також назву того простору імен, яким він належить.

Дляскорочення кваліфікованого (повного) імені потрібного нам класу (типу) в програму включають директиву

using назва_простору_імен;

Наприклад, щоб написати програму на C# для роботи з файлами, в ній використовується директива:

using System.IO;

Після цього в програмі стають доступні за допомогою скорочених імен класів типи, необхідні для організації введення-виведення.

Нові типи значень можуть бути введені в програму як перерахування та структури. Для додавання нових типів посилань використовують класи, інтерфейси, делегати.

2.8 Прості типи. Константи-літерали

Прості типи значень мови C # можна поділити на наступні групи:

• числові (арифметичні) типи;

• логічний (булєвський) тип;

• символьний тип.

До числових типів відносяться: знакові і беззнакові цілочисельні типи, дійсні типи і десятковий тип.

Числові значення представляються в програмі за допомогою констант (літералів), і змінних трьох різних видів:

- цілі числа (знакові типи: sbyte, short, int, long, беззнакові типи: byte, ushort, uint, ulong);
- дійсні числа (типи float, double);
- десяткові числа (тип decimal).

Приклади цілочисельних літералів:

48 - знакового цілого типу (int);

43L - знакового довгого цілого типу (long);

49U - беззнакового цілого типу (uint);

93UL - беззнакового довгого цілого типу (ulong).

Константи (літерали) речових типів можуть бути записані у вигляді з фіксованою точкою:

101.284 - тип double;

-0.221F - тип float;

12.0f - тип float;

Крім того, широко використовується експоненціальний запис - наукова нотація, при якій явно виписуються мантиса і експонента, а між ними розміщується розділити E або e.

Приклади:

-0.24E-13 - тип double

1.44E + 11F - тип float

-16.3E + 02f - тип float

Тип decimal спеціально введений в мову C#, щоб забезпечити обчислення, при виконанні яких неприпустимі (точніше, повинні бути мінімізовані) помилки округлення. Наприклад, при фінансових обчисленнях з великими сумами навіть мінімальні похибки за рахунок округлення можуть призводити до помітних втрат.

Змінні і константи типу decimal дозволяють представляти числові значення в діапазоні від 10^{-28} до 7,9. 10^{-28}. Для кожного числа виділяється 128 двійкових розрядів, причому число зберігається в формі з фіксованою точкою. За допомогою цього типу можна представляти числа, які мають до 28-ми десяткових розрядів.

У записі десяткової константи використовується суфікс m (або M).

Приклади десяткових літералів:

308.0008M

12.6m

123456789000m

Для представлення логічних значень використовуються константи типу bool:

true - істина;

false - брехня.

У порівнянні з попередніми мовами, наприклад, Сі і С ++ в С # для подання кодів окремих символів (для даних типу char) використовується 1 байт, а 2 байти і для кодування використовується Unicod. Символьні літерали обмежені обов'язковими апострофами:

'A', 'z', '2', 'О', 'Я'.

У символьних літералах для представлення одного символу можуть використовуватися ескейп-послідовності, кожна з яких починається зі зворотним косою риси \. У вигляді ескейп-послідовностей зображуються керуючі символи:

\ '- апостроф;

\ " - лапки;

\\ - зворотна коса риска;

\ a - звуковий сигнал;

\ b - повернення на крок (забій);

\ n - новий рядок;

\ r - повернення каретки;

\ t - табуляція (горизонтальна);

\ 0 - нульове значення;

\ f - переклад сторінки;

\ v - вертикальна табуляція.

За допомогою явної записи числового значення коду ескейп-послідовністю можна уявити будь-який символ Юнікоду.

Формат такого уявлення:

'\uhhhh',

де h - шістнадцяткова цифра, u - обов'язковий префікс.

Граничні значення від '\ u0000' до '\uFFFF'.

2.9 Оголошення змінних і констант базових типів

Успадкований від мов Сі і С ++, новий екземпляр (змінна) простого типу вводиться за допомогою оголошення:

type name = expression;

де type - назва типу;

name - ім'я екземпляра (змінної);

expression - ініціалізуючий вираз (наприклад, константа).

Оголошення обов'язково завершується крапкою з комою.

Назви базових типів з прикладами оголошень наведені в табл. 2.1.

В одному оголошенні можуть бути визначені кілька змінних одного типу:

type name1 = expression1, name2 = expression 2;

Змінні одного оголошення відокремлюються одна від одної комами.

Таблиця 2.1

Прості (базові) типи значень

Тип	Опис	Приклади оголошень
sbyte	8-бітовий знаковий цілий (1 байт)	**sbyte** val = 12;
short	16-бітовий беззнаковий цілий (2 байт)	**short** val1 = 12;
int	32-бітовий беззнаковий цілий (4 байт)	**int** val1 = 12;
long	64-бітовий беззнаковий цілий (8 байт)	**long** val1 = 12; **long** val2 = 34L;
byte	8-бітовий беззнаковий цілий (1 байт)	**byte** val1 = 12;
ushort	16-бітовий беззнаковий цілий (2	**ushort** val1 = 12;

	байт)	
uint	32-бітовий беззнаковий цілий (4 байт)	**uint** val1 = 12; **uint** val2 = 34U;
ulong	64-байтовий беззнаковий цілий (8 байт)	**ulong** val1 = 12; **ulong** val2 = 34U; **ulong** val3 = 56L; **ulong** val4 = 78UL;
float	Дійсний з плаваючою точкою з одинарною точністю (4 байт)	**float** val = 1.23F;
double	Дійсний з плаваючою точкою з подвійною точністю (8 байт)	**double** val1 = 1.23; **double** val2 = 4.56D;
bool	Логічний тип; значення або **false** або **true**	**bool** val1 = true; **bool** val2 = false;
char	Символьний тип; значення – один символ Юнікода (2 байт)	**char** val = ‘h’;
decimal	Точний грошовий тип, як найменше 28 значущих десяткових розрядів (12 байт)	**decimal** val = 1.23M;

Ввівши формат оголошення змінних, слід зупинитися на питанні вибору їхніх імен. Ім'я змінної – обраний програмістом ідентифікатор. Ідентифікатор - це послідовність букв, десяткових цифр і символів підкреслення, яка починається з

цифри. У мові C# в якості букв допустимо застосовувати літери національних алфавітів.

В якості імен, що вводяться програмістом, заборонено використовувати службові (ключові) слова мови C#.

Таблиця 2.2.

Службові слова мови C#

abstract	**as**	**base**	**bool**	**break**
byte	**case**	**catch**	**char**	**checked**
class	**const**	**continue**	**decimal**	**default**
delegate	**do**	**double**	**else**	**enum**
event	**explicit**	**extern**	**false**	**finally**
fixed	**float**	**for**	**foreach**	**goto**
if	**implicit**	**in**	**int**	**interface**
internal	**is**	**lock**	**long**	**namespace**
new	**null**	**object**	**operator**	**out**
override	**params**	**private**	**protected**	**public**
readonly	**ref**	**return**	**sbyte**	**sealed**
short	**sizeof**	**stackalloc**	**static**	**string**
struct	**switch**	**this**	**throw**	**true**
try	**typeof**	**uint**	**ulong**	**unchecked**
unsafe	**ushort**	**using**	**virtual**	**void**
volatile	**while**			

Слід зазначити, що в табл. 2.2 не включені ідентифікатори, які грають роль ключових слів тільки в конкретному контексті. Приклади: into, set, yield, from.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРАХУНКУ ОСНОВНИХ МЕТРИК ПРОГРАМНОГО КОДУ НА МОВІ C#

Обслуговування програмного забезпечення є одним із ключових напрямків у будь-якому процесі програмної інженерії, в якому аналіз вихідного коду відіграє вирішальну роль. Через високу вартість технічного обслуговування стало абсолютно необхідним створювати високоякісне програмне забезпечення. З часом для визначення складності та інших показників було проведено численні аналізи вихідного коду. Підвищена складність сучасних програмних додатків також підвищує складність забезпечення надійності та супроводу коду. Метрики коду

являють собою набір оцінок програмного забезпечення, які дають розробникам більш глибоке уявлення про розроблюваний код. Метрики коду - це інструмент, який аналізує розроблений проект, вимірює складність та забезпечує кращу інформацію про програмний код. Використовуючи переваги метрик коду, розробники можуть зрозуміти, які типи і методи повинні бути перероблені або ретельно протестовані. Групи розробників можуть виявити потенційні ризики, розуміння поточного стану проекту і відстеження ходу виконання під час розробки програмного забезпечення.

При цьому можна використовувати програмне середовище Visual Studio для створення даних метрик коду, які вимірюють складність і зручність обслуговування керованого коду. Дані метрик коду можуть створюватися для всього рішення або окремого проекту.

Visual Studio обчислює метрики коду, результати яких є наступні:

- Індекс зручності обслуговування - обчислює значення індексу від 0 до 100, який представляє відносну простоту обслуговування коду. Високе значення означає кращу супроводжуваність. Для швидкого виявлення проблем в коді можна використовувати кольорове маркування. Зелена Оцінка знаходиться в діапазоні від 20 до 100 і вказує на те, що код має гарну супроводжуваність. Жовта Оцінка знаходиться в діапазоні від 10 до 19 і вказує, що код є помірно підтримуваним. Червона Оцінка - це оцінка між 0 і 9 і вказує на низьку супроводжуваність.
- Складність організації циклів вимірює структурну складність коду. Він створюється шляхом обчислення кількості різних шляхів коду в потоці програми. Програма, що має складний потік управління, вимагає більше тестів для досягнення хорошого об'єму протестованого коду і менш супроводжуваної.
- Глибина успадкування – вказує на кількість різних класів, які успадковують один від одного, аж до базового класу. Глибина успадкування аналогічна взаємозв'язку класів в тому сенсі, що зміна базового класу може вплинути на будь-які з його успадкованих класів. Чим вище це число, тим глибше спадкування і тим вище ймовірність внесення змін до базового класу, що

призводить до критичної зміни. Для більш глибокого успадкування низьке значення є добрим, а високе значення є неприпустимим [13].

- Взаємозв'язок класів - вимірює зв'язок з унікальними класами через параметри, локальні змінні, типи, які повертаються, виклики методів, універсальні екземпляри або шаблони шаблонів, базові класи, реалізації інтерфейсу, поля, певні в зовнішніх типах, і декорування атрибутів. Хороша розробка програмного забезпечення визначає, що типи і методи повинні мати високу зв'язність і низьку зв'язок. Високий зв'язок вказує на проект, який важко використовувати і підтримувати через безліч взаємозалежностей від інших типів.
- Рядки вихідного коду - вказує точне число рядків вихідного коду, наявних у вихідному файлі, включаючи порожні рядки. –
- Рядки виконуваного коду – вказує на приблизну кількість рядків або операцій виконуваного коду. Це кількість операцій в виконуваному коді.

Деякі програмні засоби і компілятори створюють код, який додається до проекту, і розробник проекту не може бачити або не повинен змінювати його. В основному метрики коду ігнорують сформований код при обчисленні значень метрик. Це дозволяє значенням метрик відображати, що може бачити і змінювати розробник.

Підвищена складність сучасних програмних додатків також підвищує слкладність забезпечення надійності та забезпечення коду.

Цикломатична складність - це програмна метрика (вимірювання), що використовується для позначення складності програми. Це кількісний показник кількості лінійно незалежних шляхів через вихідний код програми. Він був розроблений Томасом Дж. Маккейбом-старшим у 1976 році.

Загалом, цикломатична складність говорить про те, наскільки складним є розроблений код.

На основі цифр, наведених для кожного методу у вихідному коді, можна легко визначити, чи є код складним чи ні. Цикломатична складність також впливає на інші показники програмного забезпечення, такі як індекс

ремонтопридатності коду. По суті, при цикломатичній складності вищі числа є «поганими», а менші - «хорошими».

Цикломатична складність є більшою у класах або методах, в яких існує багато умовних операторів (наприклад, if..Else, while, switch statement).

Кількість рядків у класі або методі також впливає на цикломатичну складність. Більша кількість рядків означає поєднання декількох логік разом, що явно порушує SRP (принцип єдиної відповідальності).

Цикломатична складність допомагає розробнику, вимірюючи складність коду. Чим вища складність коду, тим складнішим є код. Кількість цикломатичної складності залежить від того, скільки різних шляхів виконання або потоку керування кодом можна виконати, залежно від різних входів.

Прийнятними є наступні показники для визначення цикломатичної складності програми, наведені в таблиці 3.1.

Таблиця 3.1

Показники цикломатичної складності програми

Оцінка	Цикломатична складність	Тип ризику
Від 1 до 10	Проста	Невеликий ризик
Від 11 до 20	Складна	Низький ризик
Від 21 до 50	Занадто складна	Середній ризик, увага
Більше 50	Занадто складна	Високий ризик, не вдається протестувати

Метрика «Цикломатична складність» має такі переваги:

- дозволяє легко знайти складний код;
- допомогає зосередити зусилля на тестуванні;
- дозволяє легко обчислювати та обслуговувати код в майбутньому.

Для запуску в дію метрики «Цикломатична складність» необхідно створити нову консольну програму і обчислити метрики коду, перейшовши до меню Analyze | Calculate Code Metrics for Solution. На рис. Представлено вікна діалогу для вибору опцій меню « Новий проект» та « Метрики обчислення коду»

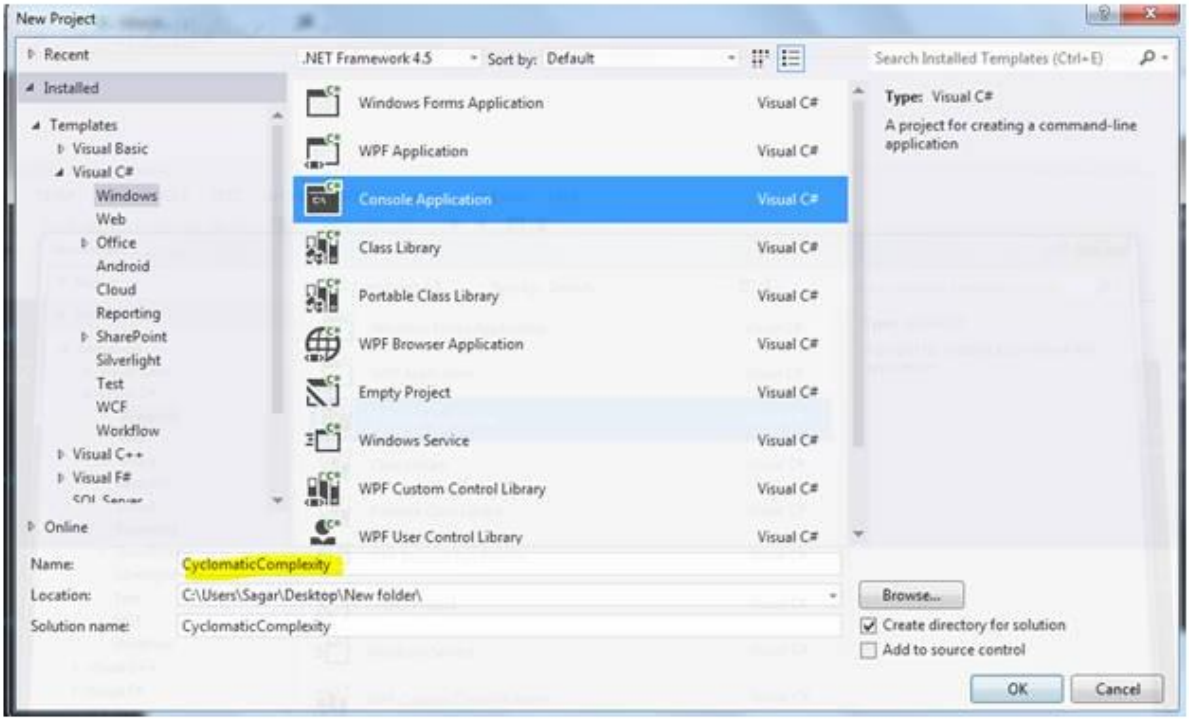


Рисунок 3.1 - Вікно діалогу для вибору опції меню « Новий проект»

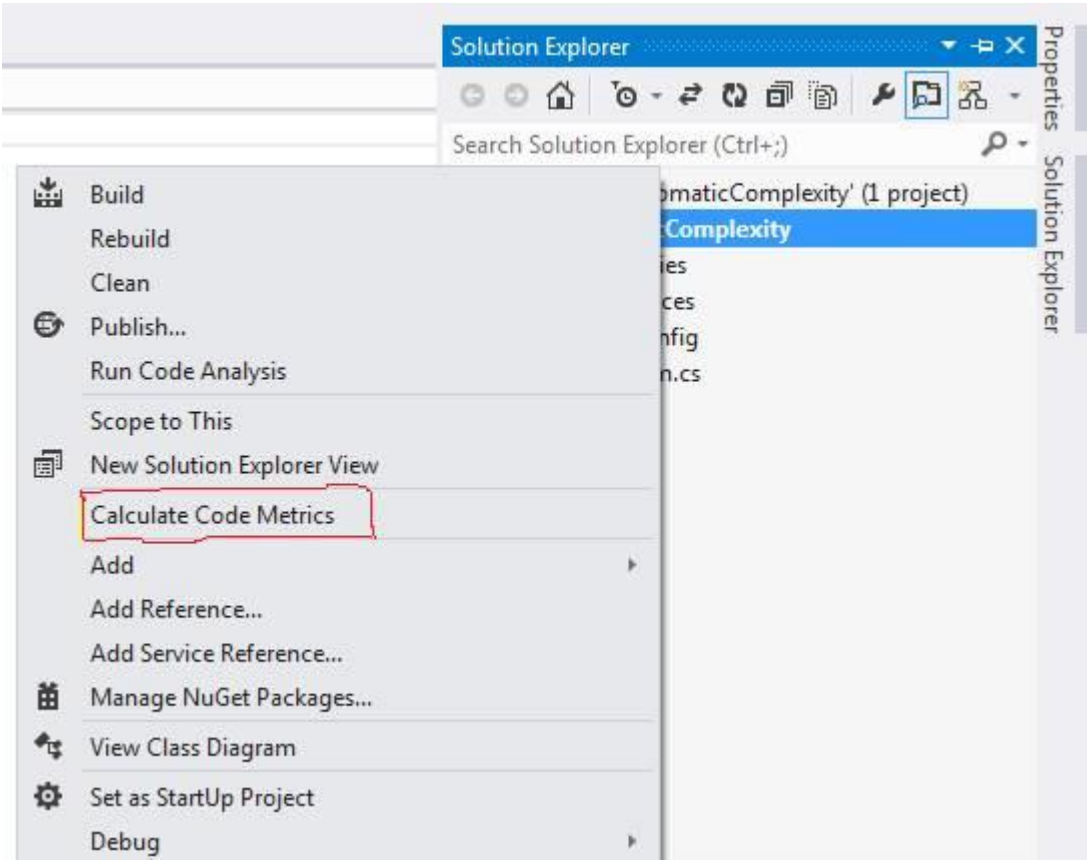


Рисунок 3.2 - Вікно діалогу для вибору опції меню « Метрики обчислення коду»

На рис. 3.3 представлено вікно для перевірки циклометричної складності коду.

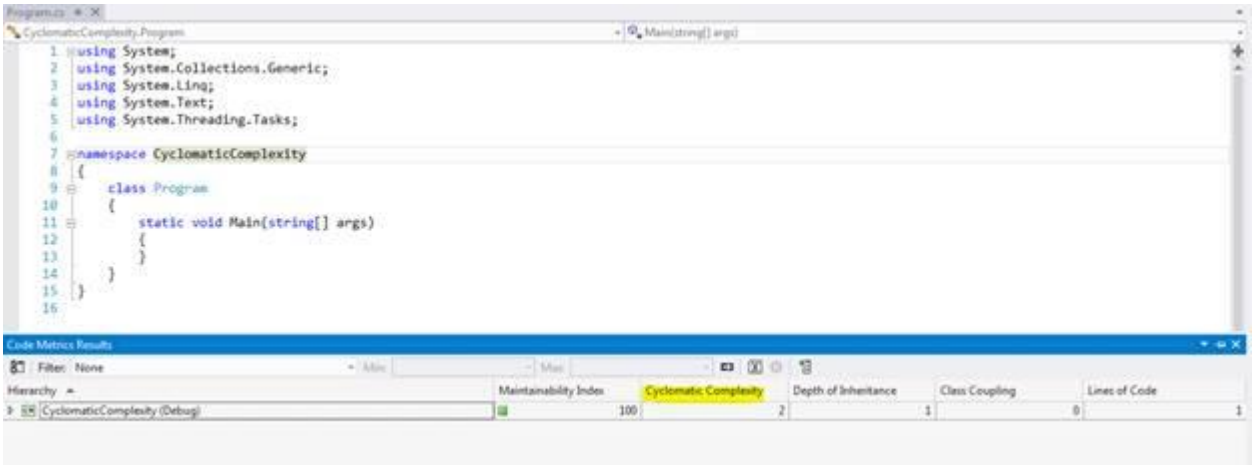


Рисунок 3.3 – Вікно меню для перевірки циклометричної складності коду

Нижче наведено лістинг програми для розрахунку основних метрик програмного коду на мові C#.

153 lines (153 sloc) 6.16 KB

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import
Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Commo
n.props"
Condition="Exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microso
ft.Common.props')" />
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == " ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == " ">AnyCPU</Platform>
    <ProjectGuid>{A8A05693-B329-4DF3-B37B-
31D90FDD89DF}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>Software_Metric_Tools</RootNamespace>
    <AssemblyName>Software Metric Tools</AssemblyName>
    <TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
    <FileAlignment>512</FileAlignment>
    <IsWebBootstrapper>false</IsWebBootstrapper>
    <PublishUrl>publish\Software Project Management\</PublishUrl>
    <Install>true</Install>
    <InstallFrom>Disk</InstallFrom>
    <UpdateEnabled>false</UpdateEnabled>
    <UpdateMode>Foreground</UpdateMode>
    <UpdateInterval>7</UpdateInterval>
    <UpdateIntervalUnits>Days</UpdateIntervalUnits>
    <UpdatePeriodically>false</UpdatePeriodically>
    <UpdateRequired>false</UpdateRequired>
    <MapFileExtensions>true</MapFileExtensions>
    <ApplicationRevision>3</ApplicationRevision>
    <ApplicationVersion>1.0.0.%2a</ApplicationVersion>
    <UseApplicationTrust>false</UseApplicationTrust>
    <PublishWizardCompleted>true</PublishWizardCompleted>
    <BootstrapperEnabled>true</BootstrapperEnabled>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU'
">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
```

```
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' ==
'Release|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup>

<ManifestCertificateThumbprint>D46B6F42DFA45D0FFCF627D243F462CC865
9A9D9</ManifestCertificateThumbprint>
  </PropertyGroup>
  <PropertyGroup>
    <ManifestKeyFile>Software Project
Management_TemporaryKey.pfx</ManifestKeyFile>
  </PropertyGroup>
  <PropertyGroup>
    <GenerateManifests>true</GenerateManifests>
  </PropertyGroup>
  <PropertyGroup>
    <SignManifests>true</SignManifests>
  </PropertyGroup>
  <PropertyGroup>
    <ApplicationIcon>favicon.ico</ApplicationIcon>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="GraphLib">

<HintPath>..\..\..\..\..\Desktop\GraphDisplay\GraphLib\bin\Release\GraphLib.dll</
HintPath>
    </Reference>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="System.Xml.Linq" />
    <Reference Include="System.Data.DataSetExtensions" />
    <Reference Include="Microsoft.CSharp" />
    <Reference Include="System.Data" />
    <Reference Include="System.Deployment" />
```

```
  <Reference Include="System.Drawing" />
  <Reference Include="System.Windows.Forms" />
  <Reference Include="System.Xml" />
 </ItemGroup>
 <ItemGroup>
  <Compile Include="Main.cs">
   <SubType>Form</SubType>
  </Compile>
  <Compile Include="Main.Designer.cs">
   <DependentUpon>Main.cs</DependentUpon>
  </Compile>
  <Compile Include="Program.cs" />
  <Compile Include="Properties\AssemblyInfo.cs" />
  <EmbeddedResource Include="Main.resx">
   <DependentUpon>Main.cs</DependentUpon>
  </EmbeddedResource>
  <EmbeddedResource Include="Properties\Resources.resx">
   <Generator>ResXFileCodeGenerator</Generator>
   <LastGenOutput>Resources.Designer.cs</LastGenOutput>
   <SubType>Designer</SubType>
  </EmbeddedResource>
  <Compile Include="Properties\Resources.Designer.cs">
   <AutoGen>True</AutoGen>
   <DependentUpon>Resources.resx</DependentUpon>
   <DesignTime>True</DesignTime>
  </Compile>
  <None Include="Properties\Settings.settings">
   <Generator>SettingsSingleFileGenerator</Generator>
   <LastGenOutput>Settings.Designer.cs</LastGenOutput>
  </None>
  <Compile Include="Properties\Settings.Designer.cs">
   <AutoGen>True</AutoGen>
   <DependentUpon>Settings.settings</DependentUpon>
   <DesignTimeSharedInput>True</DesignTimeSharedInput>
  </Compile>
  <None Include="Software Project Management_TemporaryKey.pfx" />
 </ItemGroup>
 <ItemGroup>
  <None Include="App.config" />
 </ItemGroup>
 <ItemGroup>
  <ProjectReference Include="..\spm_core\spm_core.csproj">
   <Project>{72bf5934-1c4a-48db-8c84-001aa25bc41e}</Project>
   <Name>spm_core</Name>
```

```
    </ProjectReference>
  </ItemGroup>
  <ItemGroup>
    <BootstrapperPackage Include=".NETFramework,Version=v4.5">
      <Visible>False</Visible>
      <ProductName>Microsoft .NET Framework 4.5 %28x86 and
x64%29</ProductName>
      <Install>true</Install>
    </BootstrapperPackage>
    <BootstrapperPackage Include="Microsoft.Net.Client.3.5">
      <Visible>False</Visible>
      <ProductName>.NET Framework 3.5 SP1 Client Profile</ProductName>
      <Install>false</Install>
    </BootstrapperPackage>
    <BootstrapperPackage Include="Microsoft.Net.Framework.3.5.SP1">
      <Visible>False</Visible>
      <ProductName>.NET Framework 3.5 SP1</ProductName>
      <Install>false</Install>
    </BootstrapperPackage>
  </ItemGroup>
  <ItemGroup>
    <WCFMetadata Include="Service References\" />
  </ItemGroup>
  <ItemGroup>
    <Content Include="favicon.ico" />
  </ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
  <!-- To modify your build process, add your task inside one of the targets below
and uncomment it.
       Other similar extension points exist, see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
</Project>
```

ВИСНОВКИ

В бакалаврській роботі було досліджено область програмних метрик та виявлено програмні метрики, пов'язані зі складністю коду. У цій роботі проведено ретельне вивчення та визначення стосовно автоматичного вимірювання складності вихідного коду.

Ця робота узагальнює теорію про метрики програмного забезпечення, призначення та класифікацію метрик та області, де метрики можуть бути корисними для використання.

Метрики програмного забезпечення можуть слугувати лише рекомендаціями, але на них не можна повністю покладатися.

Вказівка мети для збору метрик є життєво важливим для забезпечення відповідних зібраних метрик. Велика кількість метрик стає заплутаною, якщо вони не потрібні. Один показник може бути сам по собі безглуздим, але разом із іншим
може дати щось корисне.

Нові власні метрики можна розробити, використовуючи деякі з існуючих.

Максимальні та мінімальні метрики слід визначати в залежності від мови програмування, думки програміста та вимог до конкретного коду програмування, який піддається аналізу. При цьому успішна реалізація аналізу складності для одного проекту може бути невдалою для іншого проекту.

Зібрані метрики слід більш ретельно оцінювати для знаходження можливої кореляції між певними метриками.

Результати зібраних метрик слід ретельно перевірити за допомогою інших метрик.

Проміжний інструмент, що використовується для збору метрик, можна опустити, а інструмент – повністю розроблений всередині компанії, а не покладатися на сторонні інструменти, де функціональність інструменту може змінитися для нових версій.

Метод відображення джерела файлу коду для реальних функцій потрібно визначити, але якщо цей метод визначений, цей інструмент повинен розширити свою функціональність щоб показати складність реальних функцій та порівняння з попередніми версіями.

Критерії оцінки метрик та максимальні або мінімальні обмеження повинні бути перевизначені для проблемних метрик.

ПЕРЕЛІК ПОСИЛАНЬ

1. ДСТУ 3008:2015 Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. Київ ДП «УкрНДНЦ , 2016.

2. Положення про кваліфікаційну роботу бакалавра. Методичні вказівки до виконання та захисту кваліфікаційної роботи бакалавра для студентів денної та заочної форм навчання/ – Київ: ДУТ, 2021. – 32 с.

3. C#. Language Specification. Version 4.0.: Microsoft Corporation, 2010. – 505 pp.

4. ECMA-334. C# Language Specification. 4th Edition / June 2006. – Geneva (ISO/IEC 23270:2006). – 553 pp.

5. Бадд Т. Объектно-ориентированное программирование в действии. – СПб.: Питер, 1997. – 796 с.

6. Гросс К. C# 2008 и платформа NET 3.5 Framework: базовое руководство, 2 е изд. – СПб.: БХВ-Петербург. 2009. – 576 с.

7. Либерти Д. Программирование на C#. – СПб: Символ-Плюс, 2003. – 688 с.

8. Нейгел К. и др. C# 2008 и платформа .NET 3.5 для профессионалов. – М.: ООО «И.Д. Вильямс», 2009. – 1392 с.

9. Нэш Т. C# 2008: ускоренный курс для профессионалов. – М. : ООО «И.Д. Вильямс», 2008. – 576 с.

10. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер класс. 2-е изд. исправ. – М.: Русская Редакция; СПб.: Питер, 2008. – 656 с.

11. Скит Дж. C#: программирование для проффесионалов. 2-е изд. – М.: ООО «И.Д. Вильямс», 2011. – 544 с. 991 с.

12. Фролов А.В., Фролов Г.В. Язык C#. Самоучитель. – М.: ДИАЛОГ-МИФИ, 2003. – 560 с.

13. Шилдт Г. Полный справочник по C#. – М.: «И.Д. Вильямс», 2004. – 752 с.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРАХУНКУ ОСНОВНИХ МЕТРИК ПРОГРАМНОГО КОДУ НА МОВІ C#

Виконав студент 5 курсу
Групи ППЗ-52
Дяченко Максим Вячеславович
Керівник роботи старший викладач
Гаманюк Ігор Михайлович

Київ – 2021

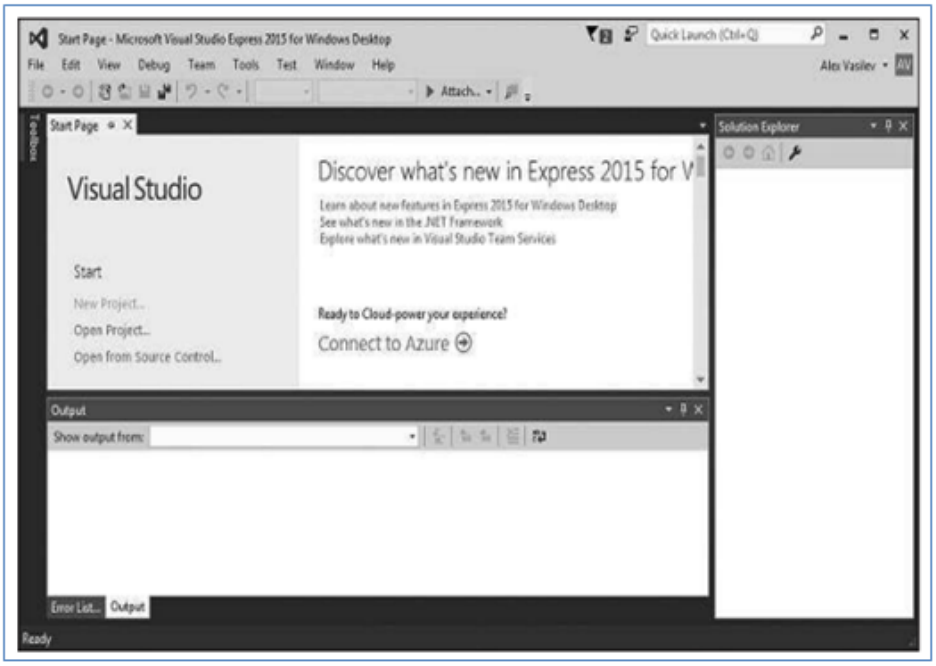
МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи:** Розрахунок основних метрик на мові програмного коду на мові C#
- **Об’єкт дослідження:** мова прогамування C#
- **Предмет дослідження:** основні метрики програмного коду на мові C#

ТЕХНІЧНІ ЗАВДАННЯ

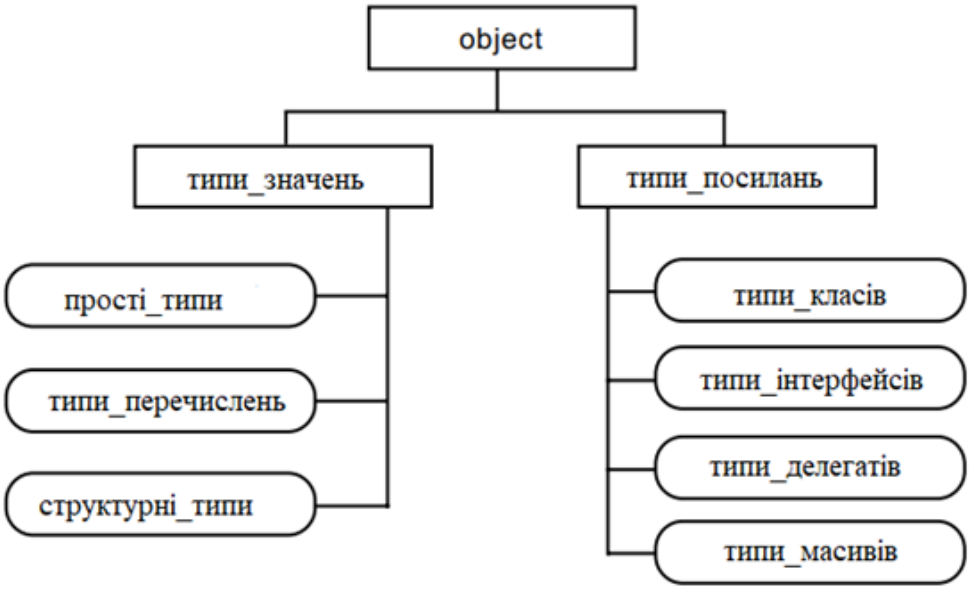
- *1.* Загальний огляд метрик програмного забезпечення
- *2.* Принципи програмування на мові C#
- *3* Практична реалізація програмного забезпечення для розрахунку основних метрик програмного коду на мові C#

Вікно додатка Visual Studio Express



4

Схема типів мови C#



5

Прості (базові) типи значень

Тип	Опис	Приклади оголошень
sbyte	8-бітовий знаковий цілий (1 байт)	**sbyte** val = 12;
short	16-бітовий беззнаковий цілий (2 байт)	**short** val1 = 12;
int	32-бітовий беззнаковий цілий (4 байт)	**int** val1 = 12;
long	64-бітовий беззнаковий цілий (8 байт)	**long** val1 = 12; **long** val2 = 34L;
byte	8-бітовий беззнаковий цілий (1 байт)	**byte** val1 = 12;
ushort	16-бітовий беззнаковий цілий (2 байт)	**ushort** val1 = 12;
uint	32-бітовий беззнаковий цілий (4 байт)	**uint** val1 = 12; **uint** val2 = 34U;
ulong	64-байтовий беззнаковий цілий (8 байт)	**ulong** val1 = 12; **ulong** val2 = 34U; **ulong** val3 = 56L; **ulong** val4 = 78UL;
float	Дійсний з плаваючою точкою з одинарною точністю (4 байт)	**float** val = 1.23F;
double	Дійсний з плаваючою точкою з подвійною точністю (8 байт)	**double** val1 = 1.23; **double** val2 = 4.56D;
bool	Логічний тип; значення або **false** або **true**	**bool** val1 = true; **bool** val2 = false;
char	Символьний тип; значення – один символ Юнікода (2 байт)	**char** val = 'h';
decimal	Точний грошовий тип, як найменше 28 значущих десяткових розрядів (12 байт)	**decimal** val = 1.23M;

6

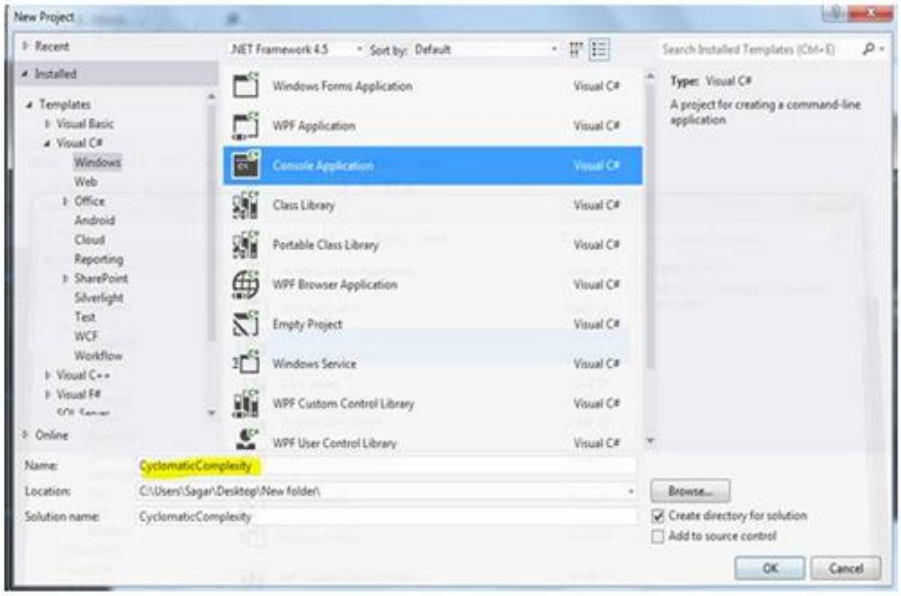
Службові слова мови C#

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

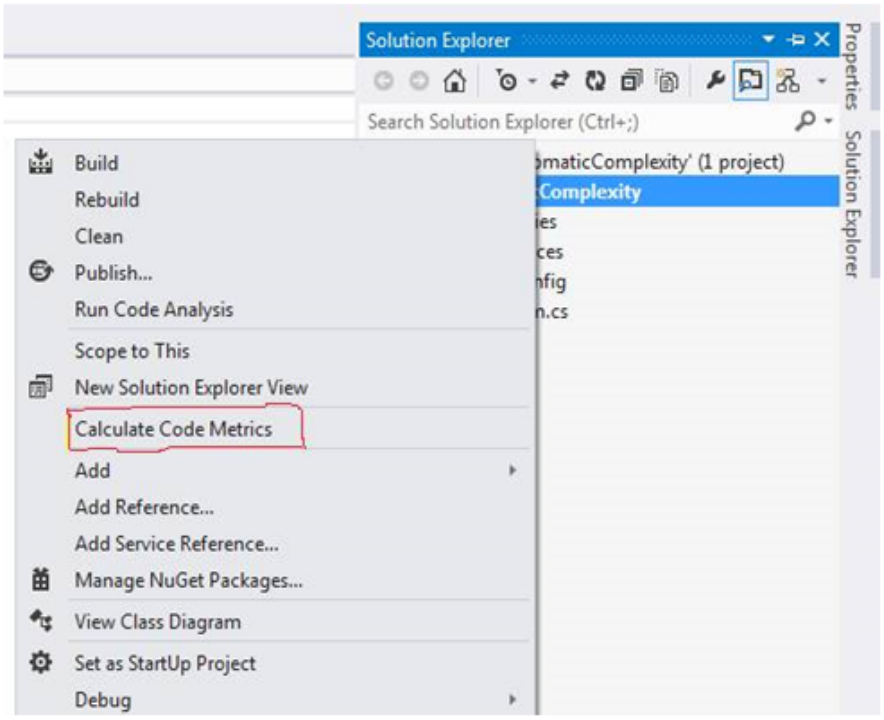
Показники цикломатичної складності програми

Оцінка	Цикломатична складність	Тип ризику
Від 1 до 10	Проста	Невеликий ризик
Від 11 до 20	Складна	Низький ризик
Від 21 до 50	Занадто складна	Середній ризик, увага
Більше 50	Занадто складна	Високий ризик, не вдається протестувати

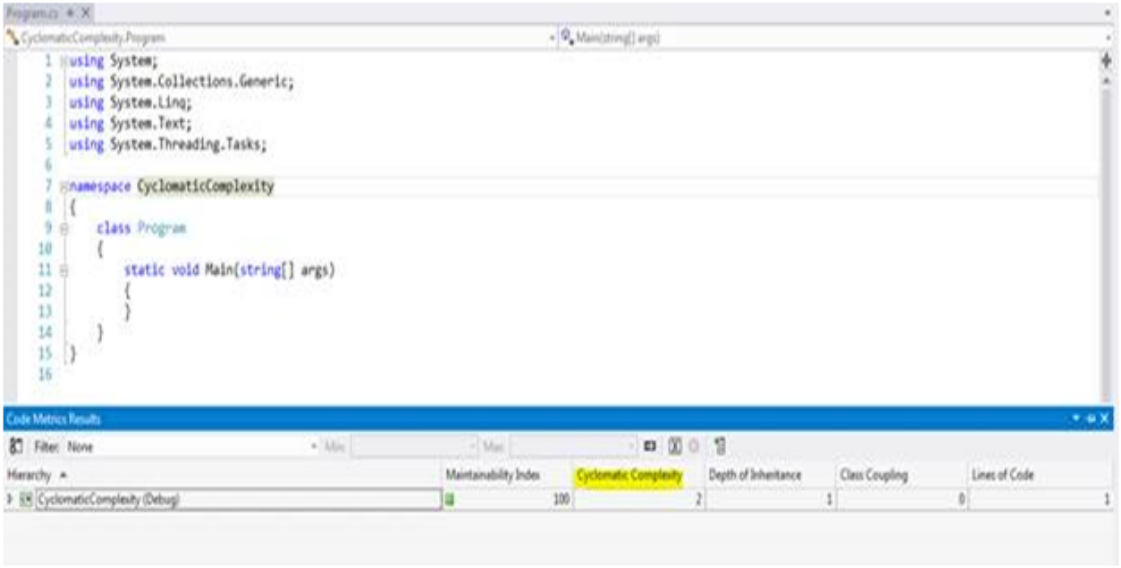
Вікно діалогу для вибору опції меню «Новий проект»



Вікно діалогу для вибору опції меню «Метрики обчислення коду»



Вікно меню для перевірки циклометричної складності коду



Лістинг програми для розрахунку основних метрик програмного коду на мові C#

```
    <Reference Include="Microsoft.CSharp" />
    <Reference Include="System.Data" />
    <Reference Include="System.Deployment" />
    <Reference Include="System.Drawing" />
    <Reference Include="System.Windows.Forms" />
    <Reference Include="System.Xml" />
  </ItemGroup>
  <ItemGroup>
    <Compile Include="Main.cs">
      <SubType>Form</SubType>
    </Compile>
    <Compile Include="Main.Designer.cs">
      <DependentUpon>Main.cs</DependentUpon>
    </Compile>
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
    <EmbeddedResource Include="Main.resx">
      <DependentUpon>Main.cs</DependentUpon>
    </EmbeddedResource>
    <EmbeddedResource Include="Properties\Resources.resx">
      <Generator>ResXFileCodeGenerator</Generator>
      <LastGenOutput>Resources.Designer.cs</LastGenOutput>
      <SubType>Designer</SubType>
    </EmbeddedResource>
    <Compile Include="Properties\Resources.Designer.cs">
      <AutoGen>True</AutoGen>
      <DependentUpon>Resources.resx</DependentUpon>
      <DesignTime>True</DesignTime>
    </Compile>
    <None Include="Properties\Settings.settings">
      <Generator>SettingsSingleFileGenerator</Generator>
      <LastGenOutput>Settings.Designer.cs</LastGenOutput>
    </None>
    <Compile Include="Properties\Settings.Designer.cs">
      <AutoGen>True</AutoGen>
      <DependentUpon>Settings.settings</DependentUpon>
      <DesignTimeSharedInput>True</DesignTimeSharedInput>
    </Compile>
    <None Include="Software Project Management_TemporaryKey.pfx" />
  </ItemGroup>
  <ItemGroup>
    <None Include="App.config" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\spm_core\spm_core.csproj">
      <Project>{72bf5934-1c4a-48db-8c84-001aa25bc41e}</Project>
      <Name>spm_core</Name>
    </ProjectReference>
  </ItemGroup>
  <ItemGroup>
    <BootstrapperPackage Include=".NETFramework,Version=v4.5">
      <Visible>False</Visible>
      <ProductName>Microsoft .NET Framework 4.5 %28x86 and
x64%29</ProductName>
      <Install>true</Install>
    </BootstrapperPackage>
    <BootstrapperPackage Include="Microsoft.Net.Client.3.5">
      <Visible>False</Visible>
      <ProductName>.NET Framework 3.5 SP1 Client
Profile</ProductName>
      <Install>false</Install>
    </BootstrapperPackage>
    <BootstrapperPackage Include="Microsoft.Net.Framework.3.5.SP1">
      <Visible>False</Visible>
      <ProductName>.NET Framework 3.5 SP1</ProductName>
      <Install>false</Install>
    </BootstrapperPackage>
  </ItemGroup>
  <ItemGroup>
    <WCFMetadata Include="Service References\" />
  </ItemGroup>
  <ItemGroup>
    <Content Include="favicon.ico" />
  </ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
  <!-- To modify your build process, add your task inside one of the
targets below and uncomment it.
       Other similar extension points exist, see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
</Project>
```

Лістинг програми для розрахунку основних метрик програмного коду на мові C#

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import
Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microso
ft.Common.props"
Condition="Exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion
)\Microsoft.Common.props')" />
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == "
">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == ' ">AnyCPU</Platform>
    <ProjectGuid>{A8A05693-B329-4DF3-B37B-
31D90FDD89DE}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>Software_Metric_Tools</RootNamespace>
    <AssemblyName>Software Metric Tools</AssemblyName>
    <TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
    <FileAlignment>512</FileAlignment>
    <IsWebBootstrapper>false</IsWebBootstrapper>
    <PublishUrl>publish\Software Project Management\</PublishUrl>
    <Install>true</Install>
    <InstallFrom>Disk</InstallFrom>
    <UpdateEnabled>false</UpdateEnabled>
    <UpdateMode>Foreground</UpdateMode>
    <UpdateInterval>7</UpdateInterval>
    <UpdateIntervalUnits>Days</UpdateIntervalUnits>
    <UpdatePeriodically>false</UpdatePeriodically>
    <UpdateRequired>false</UpdateRequired>
    <MapFileExtensions>true</MapFileExtensions>
    <ApplicationRevision>3</ApplicationRevision>
    <ApplicationVersion>1.0.0.%2a</ApplicationVersion>
    <UseApplicationTrust>false</UseApplicationTrust>
    <PublishWizardCompleted>true</PublishWizardCompleted>
    <BootstrapperEnabled>true</BootstrapperEnabled>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' ==
'Debug|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' ==
'Release|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup>

<ManifestCertificateThumbprint>D46B6F42DFA45D0FFCF627D243F
462CC8659A9D9</ManifestCertificateThumbprint>
  </PropertyGroup>
  <PropertyGroup>
    <ManifestKeyFile>Software Project
Management_TemporaryKey.pfx</ManifestKeyFile>
  </PropertyGroup>
  <PropertyGroup>
    <GenerateManifests>true</GenerateManifests>
  </PropertyGroup>
  <PropertyGroup>
    <SignManifests>true</SignManifests>
  </PropertyGroup>
  <PropertyGroup>
    <ApplicationIcon>favicon.ico</ApplicationIcon>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="GraphLib">

<HintPath>..\..\..\..\Desktop\GraphDisplay\GraphLib\bin\Release\Gra
phLib.dll</HintPath>
    </Reference>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="System.Xml.Linq" />
    <Reference Includ="System.Data.DataSetExtensions" />
```

Висновки

- Застосування метрик у загальному випадку дозволяє визначити складність розробленого проекту, оцінити обсяг виконаних робіт, стилістику розроблюваної програми і зусилля, які докладені кожним розробником для реалізації того чи іншого рішення.
- Метрики програмного забезпечення є кількісним показником щодо продуктивності певного програмного забезпечення стосовно людської взаємодії, необхідної для роботи програмного забезпечення.
- В роботі наведено приклад програми для розрахунку основних метрик програмного коду на мові C#.

ДЯКУЮ ЗА УВАГУ!