Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА ДИПЛОМНА РОБОТА

на тему:

Інструмент інтерфейсу командного рядка для вдосконаленого адміністрування процесів в ОС комп'ютера

ПОЯСНЮВАЛЬНА ЗАПИСКА

Виконав студент 4 курсу, групи 2КІ-18б
спеціальності 123 — Комп'ютерна інженерія
Димпалов Є. І.

Керівник к.т.н., доц. каф. ОТ
Савицька Л. А.

Рецензент кф-мн., доц., доцент каф. МБІС
Шиян А. А.

Допущено до захисту
д.т.н., проф. Азаров О.Д.

" 16 " червня 2022 р.

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітньо-кваліфікаційний рівень – бакалавр
Спеціальність – 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри
обчислювальної техніки
проф. Азарову О.Д.
«08» 02 2022 р.

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ
Димпалову Єгору Ігоровичу

1 Тема проекту «Інструмент інтерфейсу командного рядка для вдосконаленого адміністрування процесів в ОС комп'ютера», керівник роботи к.т.н., доц. каф. ОТ Савицька Л. А. затверджені наказом вищого навчального закладу від «24» березня 2022 року № 66

2 Строк подання студентом проекту 16.06.2022 р.

3 Вихідні дані до проекту: стандарти, монографії, підручники та наукові статті по темі. Існуюче програмне забезпечення, яке стосується теми бакалаврської дипломної роботи

4 Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): вступ, аналіз предметної області, проектування програми, розробка програми.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): рисунки вікна програми, рисунки коду програми.

6 Консультанти розділів роботи приведений в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Спеціальна частина	Савицька Л. А., доцент кафедри ОТ		

7 Дата видачі завдання «24» березня 2022 р.

8 Календарний план виконання БДР приведений в таблиці 2.

Таблиця 2 — Календарний план виконання БДР

№ з/п	Назва етапів БДР	Строк виконання	Примітка
1	Визначення напрямку бакалаврської роботи, формулювання теми	25 березня – 3 квітня 2022 р.	Вик.
2	Аналіз предметної області обраної теми	5-25 квітня 2022 р.	Вик.
3	Розробка алгоритму роботи	26-30 квітня 2022 р.	Вик.
4	Написання бакалаврської роботи на основі розробленої теми	1-17 травня 2022 р.	Вик.
5	Передзахист бакалаврської дипломної роботи	18-19 травня 2022 р.	Вик.
6	Виправлення, уточнення, корегування бакалаврської дипломної роботи	20 травня – 15 червня 2022 р.	Вик.
7	Захист бакалаврської дипломної роботи	16 червня 2022 р.	Вик.

Студент _____ Димпалов Є. І.

Керівник роботи _____ Савицька Л. А.

АНОТАЦІЯ

Димпалов Є. І. Інструмент інтерфейсу командного рядка для вдосконаленого адміністрування процесів в ОС комп’ютера. Бакалаврська дипломна робота зі спеціальності 123 – комп’ютерна інженерія, освітня програма – комп’ютерна інженерія. Вінниця: ВНТУ, 2022. 87 с.

На укр. мові. Бібліогр.: 22 назв; рис.: 18; табл. 3.

У бакалаврській дипломній роботі проведено аналіз існуючих інструментів інтерфейсу командного рядка та бібліотек для створення таких інструментів. Проведено дослідження актуальних технологій, які дозволяють разом із системою комп’ютера на базі ОС Windows.

Дана робота допомагає проаналізувати існуючі можливості інтерфейсу командного рядка та використати їх задля створення нового CLI tool для управління процесами.

Під час проектування програмного модуля було розглянуто різні платформи для подальшої реалізації, побудовано структуру розширення, розроблено алгоритм функціонування програмного модуля.

Під час реалізації інструменту інтерфейсу командного рядка для вдосконаленого адміністрування процесів в ОС комп’ютера було обрано мову програмування C#, середовище розробки Visual Studio 2022. Програмний модуль було протестовано та проаналізовано, отримані результати для оцінки досягнення мети даної бакалаврської дипломної роботи.

Ключові слова: CLI tool, .NET, C#, інтерфейс командного рядка

ABSTRACT

Dympalov Ye. I. Command-line interface tool for improved administration of processes computers OS. The bachelor's thesis in specialty 123 – Computer engineering. Vinnytsa: VNTU, 2022. – 87 p.

In Ukrainian language. Bibliographer: 22 titles; fig.: 18; tabl. 3.

The bachelor's thesis robot analyzed the main tools in the command line interface and libraries for creating such tools. A study was made of current data that allows at once from the system part of a computer based on Windows OS.

The work is given to help analyze the basic capability of the command line interface and its performance to create a new CLI tool for process management.

During the design of the software module, different platforms for further implementation were examined, the structure of the expansion was suggested, and the algorithm for the functioning of the software module was decomposed.

During of implementation of the command-line interface tool for a thorough administration of processes in the computer OS, the C# programming language, the Visual Studio 2022 development environment was processed.

Keywords: CLI tool, .NET, C#, Command-line interface

ЗМІСТ

ВСТУП

Інтерфейс командного рядка (command-line interface, скорочено – CLI) — це текстовий інтерфейс користувача та комп’ютера, в якому комп’ютерні інструкції можна давати лише шляхом введення рядків тексту (команд) з клавіатури.

Класичною формою інтерфейсу командного рядка, що сягає першої половини XX століття, є алфавітно-цифровий пристрій введення-виведення, наприклад, комплект з клавіатури та АЦПУ (телетайпа). Телетайп - електромеханічний друкарський верстат для передачі текстових повідомлень між двома абонентами через простий електронний канал. Згодом замість АЦПУ стали застосовувати монітори, забезпечені знакогенератором, що дозволило швидко та зручно організовувати діалог із користувачем. Подібними пристроями забезпечені або можуть бути забезпечені майже кожен сучасний комп'ютер. Такі комплекти з монітора та клавіатури (іноді з додаванням миші) називаються консоллю комп'ютера.

Відповідно до традиції консольних програм, що використовують клавіатуру та ADPU для введення та виведення відповідно, взаємодія цих програм із користувачем спрощена для зчитування зі стандартного введення та виведення на стандартний вихід. Тому є можливість перенаправляти потоки введення-виводу, взаємодіяти з користувачем через інші пристрої, в тому числі підключені через мережу, і використовувати спеціальні програми — емулятори терміналів, наприклад, малювати вікна з текстом у графічних інтерфейсах (текстові вікна).

Актуальність даної роботи полягає в тому, щоб у використанні специфічних програм, була можливість створювати нові процеси та керувати уже існуючі.

Метою роботи є покращення на технічному та користувацькому рівнях управління процесами в ОС Windows.

Задачі дослідження бакалаврської роботи:

— проаналізувати фундаментальні принципи ООП та макети класів;

— проаналізувати переваги та недоліки звичайного текстового інтерфейсу;

— проаналізувати існуючі бібліотеки для з’єднання програми із операційною системою та обрати ту, що більше усього підходить;

— виконати моделювання програми;

— виконати розробку програми-CLI за допомогою мови програмування C#;

— протестувати програму.

Об’єкт дослідження — процес створення удосконаленого CLI для адміністрування процесів.

Предмет дослідження — удосконалена CLI-програма для адміністрування процесів та створення нових процесів.

Методи дослідження бакалаврської роботи: у роботі використані базові та фундаментальні принципи ООП мови C# на платформі «.NET CORE» для реалізації програми.

Практичне значення отриманих результатів полягає у можливості конфігурувати нові процеси для специфічних програм.

1 ОГЛЯД ТА АНАЛІЗ ДОЦІЛЬНОСТІ РОЗРОБКИ ІНСТРУМЕНТУ ІНТЕРФЕЙСУ КОМАНДНОГО РЯДКА ДЛЯ ВДОСКОНАЛЕННОГО АДМІНІСТРУВАННЯ ПРОЦЕСІВ ОС КОМП'ЮТЕРА

1.1 Аналіз інтерфейсу командного рядка

Інтерфейс командного рядка (системна консоль) — засіб керування системою або програмою користувачем, також використовується для перегляду результатів виконання. Інтерфейс командного рядка можна порівняти з програмними системами управління або різними реалізаціями GUI (graphic user interface — з англ. графічний інтерфейс користувача). Формат вихідної інформації інтерфейсу командного рядка нестандартизований; зазвичай виводиться простий текст, а також графіка, аудіо виведення тощо.

У комп'ютерах раннього покоління, коли операційні системи з урахуванням графічного інтерфейсу не розроблялися, комп'ютери використовували деякі засновані командах операційні системи, такі як MS-DOS, Apple-DOS, Unix тощо. буд. Будь-яке взаємодія людини з цими ОС здійснювалося з допомогою деяких команд, які пізніше були інтерпретовані інтерпретатором, який виконував роль проміжного програмного забезпечення між людськими командами та машинною мовою ОС.

Тим не менш, навіть у сучасному світі операційних систем з графічним інтерфейсом, коли у нас є найбільш інтерактивні та прості способи взаємодії з операційною системою шляхом виконання клацань мишею, відвідування веб-сайтів і т. д. CLI, як і раніше, використовується в багатьох місцях для виконання деяких основних завдань. Найчастіше використання CLI виконується розробниками, які використовують цей інструмент для додавання або встановлення компонентів у свої програми. Вони також використовують інтерфейс командного рядка для автоматичного виконання деяких системних налаштувань на сервері залежно від сценарію, тому їм фактично не потрібно працювати на сервері. Як правило, вони зберігають командний файл або

сценарій PowerShell, який запускає команди CLI на сервері та виконує необхідні операції чи конфігурації. Приклад використання скрипта у Powershell наведенний на рисунку 1.1.

```
function LogDeployment
{
  param([string]$filepath,[string]$deployDestination)
  $datetime = Get-Date
  $filetext = "Deployed package to " + $deployDestination + " on " + $datetime
  $filetext | Out-File -filepath $filepath -Append
}

LogDeployment $args[0] $args[1]
```

Рисунок 1.1 — Приклад скрипта у PowerShell

Інтерфейс командного рядка призначений для виклику певних доступних для користувача команд для налаштування програми, або ж виконання деяких алгоритмів. CLI існує саме в такому вигляді через те, що у давні часи, коли з’явилась перша операційна система (далі ОС), було велике обмеження ресурсів комп’ютера. Тому було прийняте рішення зробити саме у вигляді тексту, з мінімальним використовування графіки. На рисунку 1.2 наведено приклад використання консолі у ОС MS-DOS.

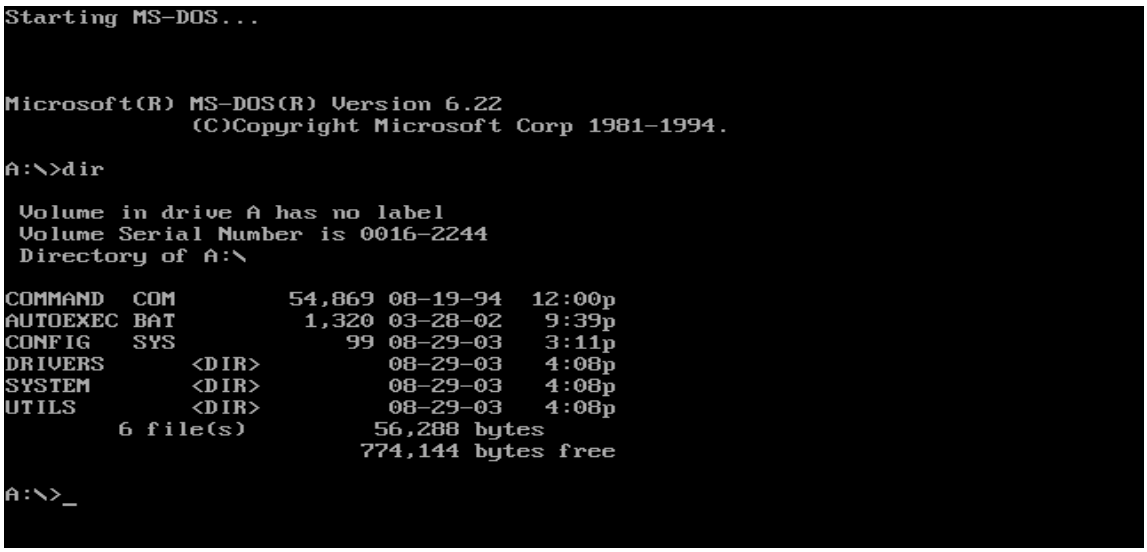


Рисунок 1.2 — Приклад використання консолі

Консоль використовується не тільки в операційних системах, а ще й у комп’ютерних іграх, чатах та інший комп’ютерних програмах. У даних додатках вона слугує як вікном з вводом/виводом певних повідомлень, тобто, не виконуючи ніяких певних алгоритмом потаємних алгоритмів, тільки повідомлення, яке йому прийшло у вигляді тексту, та повідомлення, котре він відправляє також у вигляді тексту.

1.2 Реалізація CLI

Консольним програмам не потрібно обробляти реалізацію взаємодії з користувачем самостійно, і вони обмежуються стандартним пристроєм введення/виводу, використовуючи бібліотеки, такі як «ncurses» або інші інтерфейси програмування. Фактична взаємодія з користувачем зазвичай здійснюється операційною системою або іншим програмним забезпеченням.

В даний час комп'ютери пройшли довгий шлях від інструментів командного рядка до більш просунутого і складного графічного інтерфейсу користувача, який є дуже простим і простим у використанні функціями для будь-якого користувача, що працює з комп'ютером і виконує роботу.

Однак у конфігурації комп'ютерної системи інтерфейс командного рядка все ще знаходить широке застосування. В даний час CLI в основному використовується розробниками додатків або системними адміністраторами для виконання деяких з їхніх важливих завдань, які в іншому випадку зажадали багато часу і зусиль, якби вони виконувались за допомогою графічного інтерфейсу користувача.

У 1970-х і пізніше навіть випускалися спеціальні пристрої, які реалізовували текстовий інтерфейс — текстовий термінал, підключений до комп'ютера безпосередньо через послідовний порт або через модем. З популярністю персональних комп’ютерів функції текстового терміналу зазвичай виконує комп’ютер, на якому запущена консольна програма, або інший комп’ютер. Програми «Telnet» і «SSH» дозволяють користувачам взаємодіяти з консольними програмами, запущеними на віддалених

комп’ютерах (зазвичай під керуванням UNIX) через Інтернет або локальну мережу. «xterm», «rxvt», «konsole» та багато інших програм реалізують текстові інтерфейси через текстові вікна в середовищі X Window System.

Інший метод консольного виводу використовувався в персональних комп’ютерах, особливо (але не обмежуючись) IBM PC під керуванням DOS. Програма може не тільки виводити дані за допомогою стандартного виведення, але також може безпосередньо змінювати вміст області пам’яті, пов’язаної з генератором символів монітора, що призводить до негайної зміни даних, видимих на моніторі. Такі програми також можуть працювати в середовищі Microsoft Windows. Крім того, підтримка Windows для текстових вікон у багатьох відношеннях перевершує DOS, включаючи підтримку рідних програм Windows.

Оскільки для створення нашої CLI-програми використовується мова програмування C#, то логічно те, що для реалізації програми були задіяні усі фундаментальні принципи ООП у даній мові, а саме: інкапсуляція, поліморфізм та наслідування.

Інкапсуляція (англ. encapsulation, від лат. in capsula) — у мовах програмування, процес поділу елементів абстракцій, що визначають її структуру (дані) та поведінку (методи); інкапсуляція призначена для ізоляції контрактних зобов'язань абстракції (протокол/інтерфейс) від реалізації. Насправді це означає, що клас повинен складатися з двох частин: інтерфейсу та реалізації. У реалізації більшості мов програмування (C++, C#, Java та інші) забезпечує механізм приховування, що дозволяє розмежовувати доступ до різних частин компонента, але багато видатних розробників наполягає, що інкапсуляція та приховування це різні поняття, які лише тісно взаємодіють між собою. Інкапсуляція найчастіше сприймається як поняття, властиве виключно объектно-ориентированному програмування (ООП), але насправді широко зустрічається й інших. В ООП інкапсуляція тісно пов'язана з принципом абстракції даних. Це, зокрема, тягне у себе розбіжності у термінології у різних джерелах. У співтоваристві C++ або Java прийнято розглядати інкапсуляцію

без приховування як неповноцінну. Однак деякі мови (наприклад, Smalltalk, Python) реалізують інкапсуляцію, але не передбачають можливості приховування в принципі. Інші (Standard ML, OCaml) жорстко поділяють ці поняття як ортогональні та надають їх у семантично різному вигляді (див. приховування у мові модулів ML).

Наслідування (англ. inheritance) — концепція об'єктно-орієнтованого програмування, згідно з якою абстрактний тип даних може успадковувати дані та функціональність певного типу, сприяючи повторному використанню компонентів програмного забезпечення. Спадкування є механізмом повторного використання коду та сприяє незалежному розширенню програмного забезпечення через відкриті класи (англ. public classes) та інтерфейси. Установлення відношення спадкування між класами породжує ієрархію класів. Тобто це пояснює, що наслідування було створенно для того, щоб якось зв’язати схожі по структурі та призначені класи між собою і не дублювати код, коли можна використовувати його із іншого класу, ніби він написаний саме у цьому класі.

Поліморфізм — здатність функції обробляти дані різних типів. Існує кілька різновидів поліморфізму. Дві принципово різні з них були описані Крістофером Стречі у 1967 році: це параметричний поліморфізм та ad-hoc-поліморфізм, інші форми є їх підвидами або поєднаннями. Параметричний поліморфізм є дійсним, т.к. передбачає виконання однієї й тієї ж коду всім допустимих типів аргументів, а ad-hoc-полиморфизм — уявним, т.к. являє собою забезпечення косметичної однорідності потенційно різного коду для кожного конкретного типу аргументу. У цьому існують ситуації, де необхідне використання саме ad-hoc-полиморфизма, а чи не параметричного. Теорія кваліфікованих типів поєднує всі види поліморфізму в єдину модель. Широко поширене визначення поліморфізму, приписуване Бьорну Страуструпу: «один інтерфейс — багато реалізацій», але це визначення підпадає лише ad-hoc-полиморфизм (уявний полиморфизм). Принципова можливість однієї й тієї ж

коду обробляти дані різних типів визначається властивостями системи типів мови. З цієї точки зору розрізняють статичну неполіморфну типізацію (нащадки Алгола і BCPL), динамічну типізацію (нащадки Lisp, Smalltalk, APL) і статичну поліморфну типізацію (нащадки ML). Використання ad-hoc-поліморфізму є найбільш характерним для неполіморфної типізації. Параметричний поліморфізм і динамічна типізація набагато суттєвіше, ніж ad-hoc-поліморфізм, підвищують коефіцієнт повторного використання коду, оскільки певна функція реалізує без дублювання задану поведінку для нескінченного безлічі нововизначуваних типів, що задовольняють необхідним у функції умовам. З іншого боку, часом виникає необхідність забезпечити різну поведінку функції залежно від типу параметра, і тоді необхідним є спеціальний поліморфізм. Параметричний поліморфізм є синонімом абстракції типу. Він повсюдно використовується у функціональному програмуванні, де зазвичай позначається просто як «поліморфізм».

Також у співтоваристві об'єктно-орієнтованого програмування під терміном «поліморфізм» зазвичай мають на увазі спадкування, а використання параметричного поліморфізму називають узагальненим програмуванням, або іноді «статичним поліморфізмом».

1.3 Переваги CLI

Переваги інтерфейсу комп’ютерного рядка полягають у його зручності та ефективності швидкого використання команд. Тобто, будь яку команду можна викликати невеликою кількістю натискань клавіш. Також, завдяки консолі, можна налаштовувати та виконувати певні команди або інструкції на віддалених серверах, які не мають певного графічного інтерфейсу.

Консоль зберігає історію введених команд та виведеної інформації, що дає змогу перечитати дані, які були виведені, потім, доки її не закрити. Це є дуже зручною перевагою, адже більшість графічних інтерфейсів просто виводять повідомлення на екран, які потім неможливо переглянути ще раз, не виконуючи ту саму послідовність команд знову.

Якщо взяти ОС Windows, то можна зрозуміти, що усі дії, які ми можемо виконувати, а саме: створювати новий каталог, новий файл тощо. Це є графічний інтерфейс, який просто звертається до консолі, що є дуже зручним для використання звичайним користувачем. Тобто, через консоль, у текстовому форматі, ми також можемо заходити в каталоги, редагувати їх, видаляти з них щось, додавати нові файли, запускати різні програми та інше.

У ОС Windows є файли з розширенням *.bat. Пакетний файл (*.bat) — це певний набір команд у одному файлі, які виконуються саме командним інтерпретатором, тобто, саме CLI. Пакетні файли були створенні для полегшення створення сценаріїв (скриптів) для їх автоматизації. При запуску пакетного файлу, CLI-типна програма зчитує з нього дані та виконує їх. У даній курсовій роботі .bat файли використовуються для встановлення та видалення програми інтерфейсу командного рядка для вдосконаленого адміністрування процесів.

1.4 Недоліки CLI

На жаль, консоль надто складна для початківців. Без знань певних необхідних команд користувач не зможе користуватися та виконувати необхідні йому задачі, на відміну від графічного інтерфейсу, де візуально показано майже усі потрібні користувачу команди. Звичайно, шукати необхідну команду з довідників до конкретної CLI дуже важко. У графічного інтерфейсу з цим проблем взагалі немає. Також, не знаючи необхідні оптимальні параметри для тієї чи іншої команди майже неможливо налаштувати програму або навіть запустити її на виконання.

Інколи, введення параметрів буває доволі складним. Це, в більшості, стосується комп’ютерних ігор та консольних чатів. Адже користувачі можуть використовувати специфічні нікнейми для себе, тим самим, це ускладнює взаємодію з такими користувачами. Наприклад: користувач може мати нікнейм у якому парні символи мають верхній реєстр, а непарні нижній, що робить дуже незручним дію «вигнати» користувача з серверу чи бесіди у чаті

через консоль. У графічному інтерфейсі було би вікно, де є список усіх учасників і адміністратору потрібно було би тільки обрати необхідного користувача візуальним методом та виконати дію.

Якщо програма має повноцінну скриптову мову, то це робить дуже не зручним взаємодію користувача або програміста з інтерфейсом командного рядка. Адже доводиться перемикатись із консольної мови на скриптову або повністю відмовлятися від консольної мови. Також, є варіант поєднувати ці дві мови, що дуже негативно позначається на зручності програмування.

2 ПРОЕКТУВАННЯ ІНСТРУМЕНТУ ІНТЕРФЕЙСУ КОМАНДНОГО РЯДКА ДЛЯ ВДОСКОНАЛЕНОГО АДМІНІСТРУВАННЯ ПРОЦЕСІВ В ОС КОМП'ЮТЕРА

2.1 Вибір CLI для адміністрування процесів в ОС комп'ютера

У першому розділі ми розглядали що таке взагалі CLI, його переваги і недоліки. Тепер потрібно дізнатися чому саме консоль підходить найкраще для поставленої задачі.

Розробка саме консольного додатку є більш простим. Відпадає необхідність створювати власний фреймворк для користування. Також, навіть немає потреби у графічному інтерфейсі, через простоту використання команд та виводу інформації на екран. У даному додатку був усунен недолік про незнання команд. За допомогою команди «help» на екран консолі буде виведена інформація про усі доступні для користувача у додатку команди з детальним описом як їх використовувати та що саме вони роблять.

Як же саме створити CLI tool в .NET Core? Розглянемо це у середовищі програмування Visual Studio 2022.

Для початку потрібно створити новий проект. Для цієї дії потрібно відкрити початкове вікно у Visual Studio 2022 та натиснути на кнопку «Create new project» (рисунок 2.1). Потім потрібно обрати new project template, тобто, шаблон нового проекта. Існують багато шаблонів для проектів і усі вони служать заради однієї мети — створити необхідний "фундамент" для нової програми або нового проекту, під саме ті потреби, які необхідні розробнику. Тобто якщо ви, наприклад, хочете створити власну віконну програму, то вам необхідно обрати шаблон Windows Form Application який використовує платформу .NET Framework. Він значно відрізняється від Console Application, який використовує платформу .NET Core, у наявності бібліотек класів саме для віконного додатку, що зовсім не потрібне для консольного, і навпаки, консольний шаблон має бібліотеки, котрі стосуються тільки консолі. (рисунок 2.2).

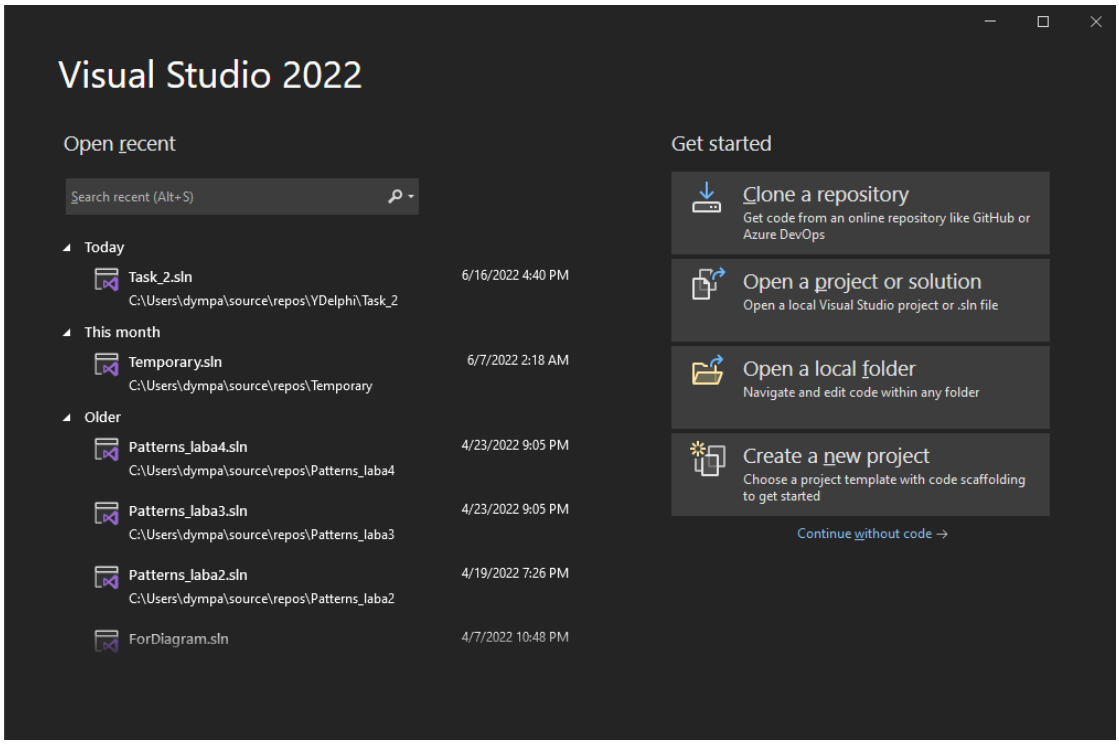


Рисунок 2.1 — Створення нового проекта

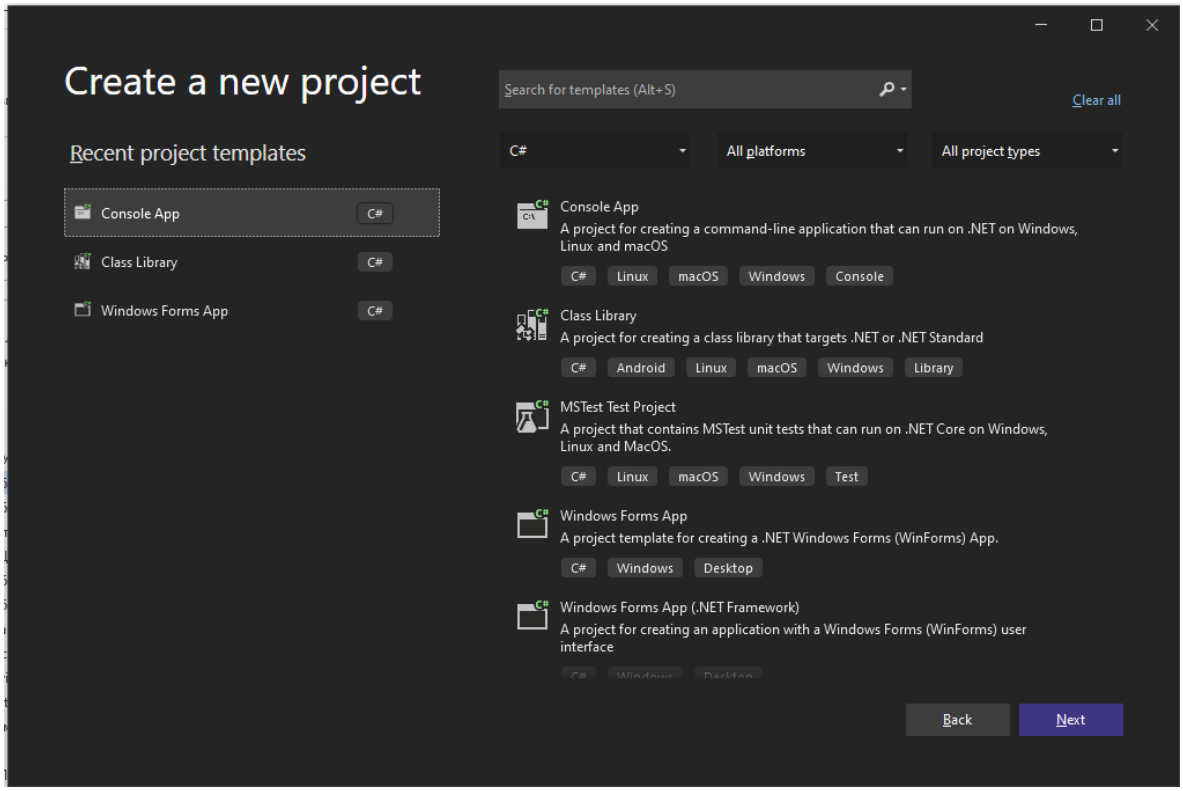


Рисунок 2.2 — Вибір шаблону нового проекту

Після вибору шаблону для нашого нового проекта потрібно обрати ім’я для нього (рисунок 2.3).

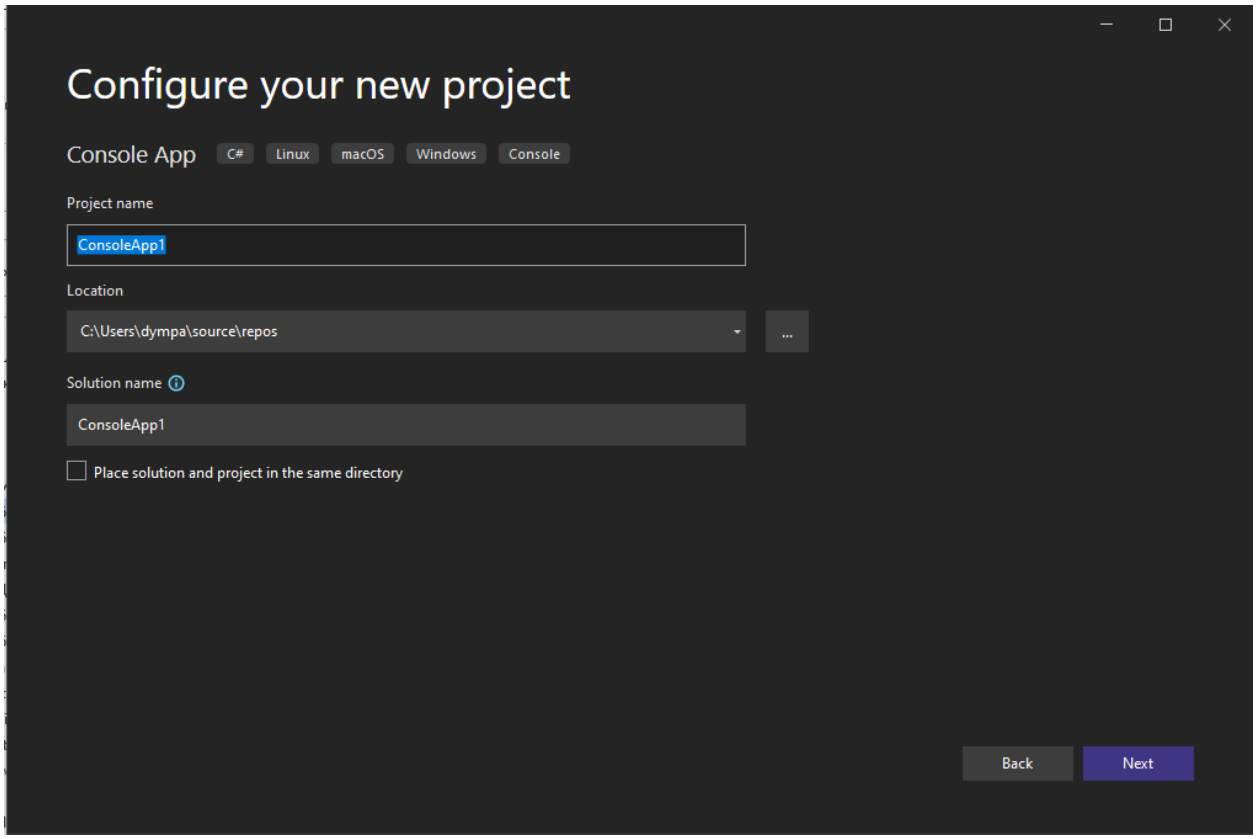


Рисунок 2.3 — Вибір назви для нового проекта

Після створення нового проекту та написання усього необхідного коду для програми, потрібно лише встановити її на комп’ютер, щоб мати можливість користуватись нею у консолі. Для цього потрібно зайти у код основної збірки, де знаходиться клас Program, у властивості «Project sdk» встановити значення «Microsoft.NET.SDK» і після цього ввести у консоль стрічку «dotnet tool install --global --add-source ./nupkg “назва програма”»

Інструмент CLI .NET — це, по суті, консольна програма, яка загорнута в пакет NuGet. Потім можна встановити цей користувацький інструмент глобально (рівень машини) або локально (рівень проекту) залежно від ваших вимог. Інструмент CLI .NET зазвичай приймає певні дані з командного рядка у вигляді аргументів командного рядка та перемикачів командного рядка. Потрібно лише обробити цей вхід у своїй консольній програмі відповідно до

певних вимог, які необхідні користувачу. Наприклад, інструмент CLI може прийняти рядок підключення до бази даних як вхідні дані та створити набір таблиць у базі даних. Перемикачі командного рядка надають інструменту різні параметри. Наприклад, у вас може бути перемикач, який повідомляє інструменту, чи потрібно перезаписувати таблиці бази даних, якщо вони вже є, чи використовувати наявні.

Завдяки саме тому, що додаток побудований на базі CLI, є можливість продивлятися історію виконання команд та навіть не потрібно заходити у лог-файли для того, щоб подивитися якісь помилки, які вийшли у результаті використання додатку, адже вони також виводяться на екран консолі. Також, завдяки інтерфейсу командного рядка додаток можна запускати та використовувати в усіх CLI, що підтримуються ОС Windows, такі, як PowerShell, Windows Terminal, Command Prompt та інші. Приклади використання додатку у PowerShell та Windows Terminal на рисунках 2.4 та 2.5 відповідно.

Така гнучкість у консолі дає можливість користувачу обирати більш зручну, саме для нього консоль і це ніяк не впливає на використовування додатку. Користувач навіть має можливість створити власну CLI в ОС Windows, тим самим, працювати з максимальним комфортом для себе з даною програмою.

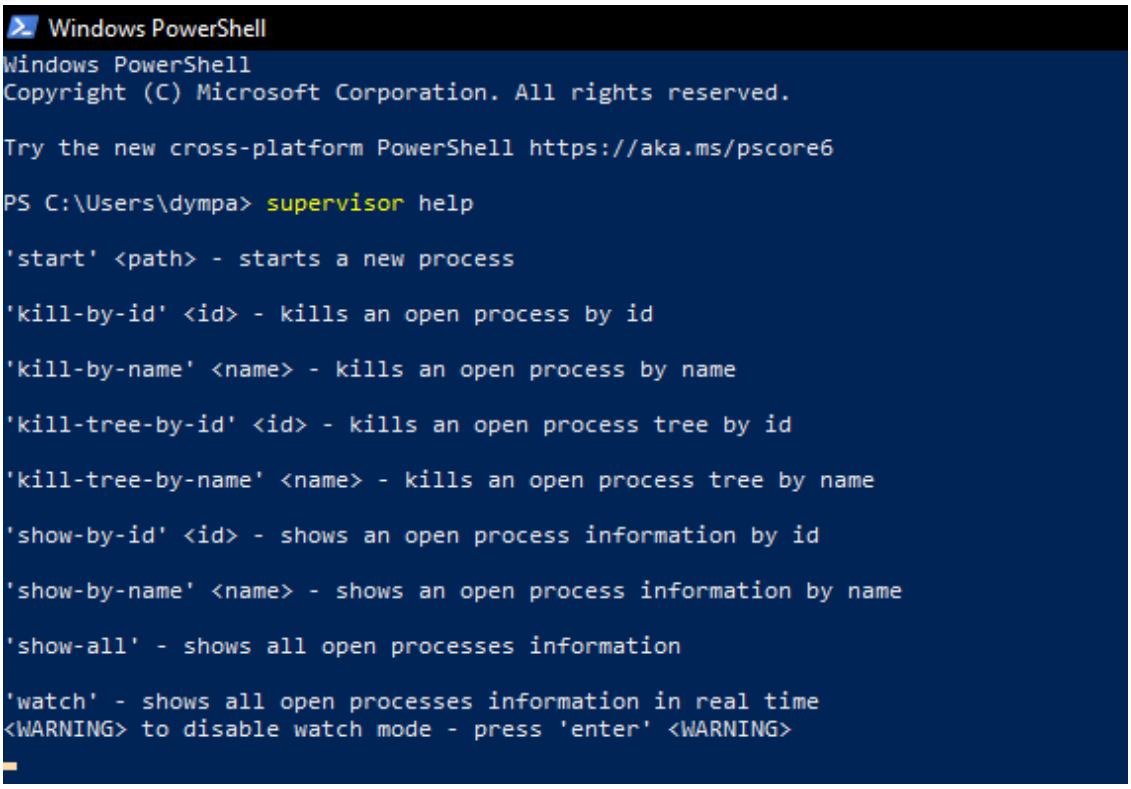


Рисунок 2.4 — Приклад використання додатку у Windows PowerShell

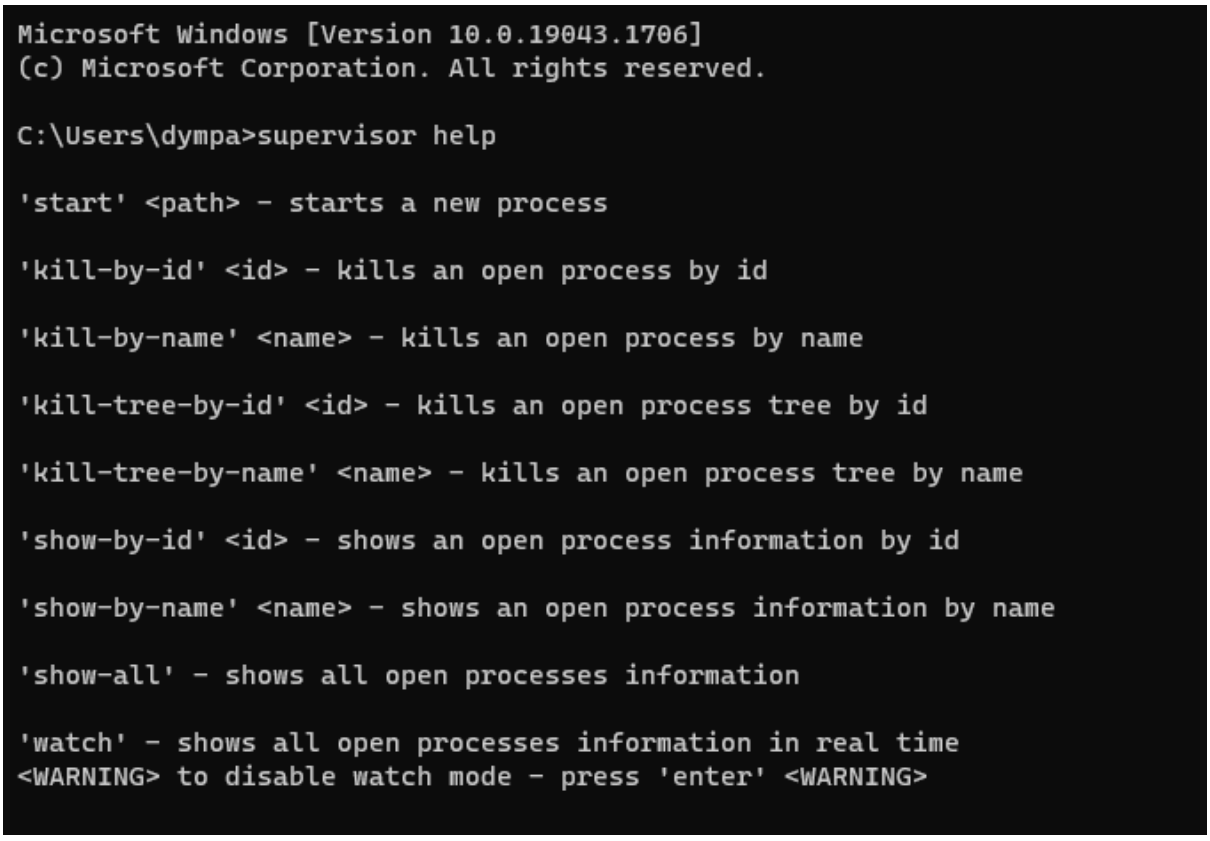


Рисунок 2.5 — Приклад використання додатку у Windows Terminal

Приклад використання програми у Command Prompt наведений на рисунку 2.6.

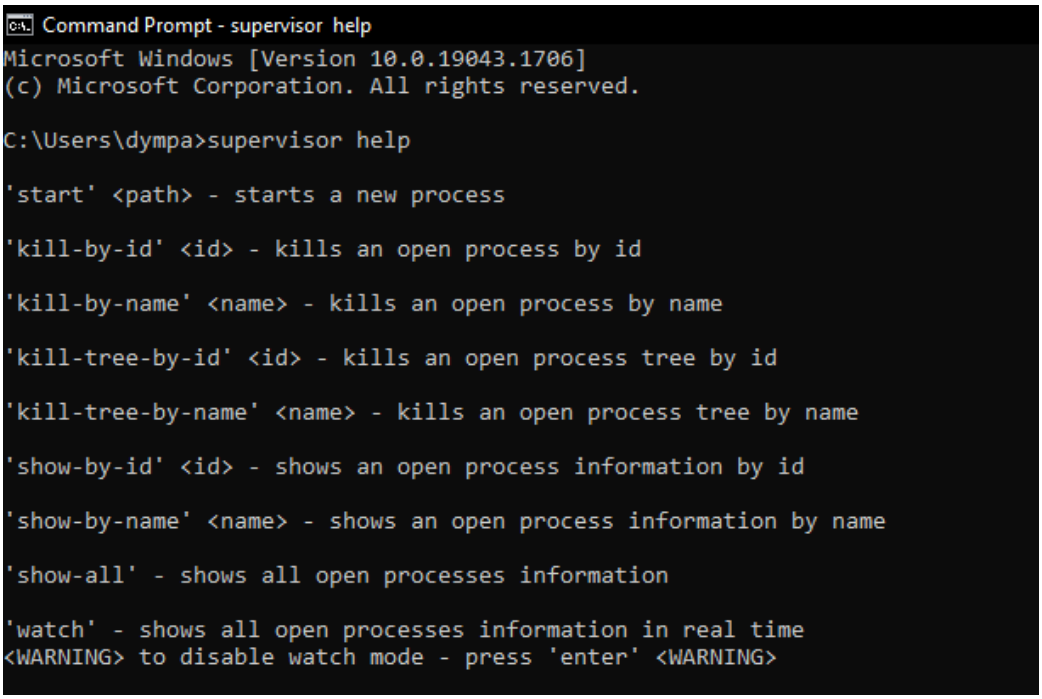


Рисунок 2.6 — Приклад використання додатку у Command Prompt

2.2 Вибір бібліотеки «ConsoleAppFramework» для CLI додатку

.NET має великий стандартний набір бібліотек класів, які називаються бібліотеками базових класів (основний набір) або бібліотеками класів платформи (повний набір). Ці бібліотеки забезпечують реалізації багатьох загальних і конкретних типів, алгоритмів і функціональних можливостей додатків. Як комерційні, так і громадські бібліотеки будуються на основі бібліотек класів фреймворків, забезпечуючи прості у використанні готові бібліотеки для широкого набору обчислювальних завдань.

Підмножина цих бібліотек надається з кожною реалізацією .NET. API бібліотеки базових класів (BCL) очікується з будь-якою реалізацією .NET, оскільки вони знадобляться розробникам, і тому, що вони знадобляться для запуску популярним бібліотекам. Спеціальні для програми бібліотеки вище BCL, такі як ASP.NET, не будуть доступні в усіх реалізаціях .NET.

BCL надає основні типи та функціональні можливості утиліт і є основою всіх інших бібліотек класів .NET. BCL має на меті забезпечити загальні реалізації без упередження будь-якого робочого навантаження. Продуктивність є важливим фактором, оскільки додатки можуть віддавати перевагу певній політиці, як-от низька затримка до високої пропускної здатності або низький рівень пам’яті до низького використання ЦП. Загалом BCL має бути високопродуктивним і використовувати середній підхід відповідно до цих різних проблем продуктивності. Для більшості програм цей підхід був досить успішним.

Для створення даної CLI-програми було використано бібліотеку «ConsoleAppFramework». Вона значно полегшує підв’язку до консолі комп’ютера на ОС Windows, що дозволяє створювати нові команди та задавати їм певний алгоритм дій. Використання цієї бібліотеки є найнеобхіднішим для робочого стану додатку, адже саме вона є тією ланкою, що зв’язує між собою алгоритми команд та виконавче середовище в ОС Windows.

Дана бібліотека була створення саме для подібних CLI додатків, що, звичайно, значно спрощує розробку програми та її реалізацію в ОС Windows.

ConsoleAppFramework побудовано на .NET Generic Host, також є можливість використовувати конфігурацію, ведення журналів, dependency injection (DI, з англ. — взаємозалежність), керування у реальному часі пакетами бібліотеки Microsoft.Extensions. ConsoleAppFramework виконує прив’язування параметрів із аргументів змінних типу string, маршрутизує багато команд, конструктор довідки в стилі .NET тощо.

Завдяки бібліотеці «ConsoleAppFramework» можна легко створювати багато командних програм. Також увімкнення використання конфігурації GenericHost — найкращий спосіб поділитися конфігурацією або робочим процесом під час створення пакетної програми для іншої веб-програми .NET. Якщо інструмент призначений для CI (continuous integration — з англ. неперервна інтеграція), то використовувати «git pull and run by dotnet run — [Command] [Option]» дуже корисно.

Стандартний API командного рядка dotnet — System.CommandLine низького рівня, вимагає багато стандартних кодів. ConsoleAppFramework схожий на ASP.NET Core в програмах CLI, не потребує шаблону. Однак завдяки потужності Generic Host це просто і легко, але набагато потужніше.

2.3 Проектування архітектури інструменту інтерфейсу командного рядка

Щоб зпроектувати архітектуру програми потрібно сформувати представлення про її роботу.

Можна описати наступним чином — є певний набір команд, кожна з яких, має певну логіку та принцип дії; також є система сповіщення про певні помилки та успішні операції, котрі виводяться у консоль та записуються у лог-файл.

Ці команди повинні працювати з процесами, які уже запущені на комп’ютері та виконуються. Є можливість запускати процеси з файлів з розширенням .exe та закривати активні процеси по їхнім ID або назвам. Також, можна створювати свої процеси, записувати їх потрібно у папці у файли з розширенням .xml (кожен процес — це окремий файл, заповнювати файл потрібно по спеціальному шаблону). Шлях до папки з усіма .xml файлами вказується у спеціальному файлі appsettings.json.

Для цього потрібно створити інтерфейси з певними методами та властивостями та розділити між ними певні алгоритми. Тобто кожен інтерфейс і, як наслідок, кожний клас, що буде реалізовувати конкретний інтерфейс, буде відповідати за щось одне, але, при цьому, вони усі будуть взаємозв’язані між собою. Наприклад: один клас буде отримувати дані від процесів, до яких програма звертається і передавати ці данні іншому класу, він буде обробляти ці дані і передавати уже готове повідомлення, що є об’єктом ще одного класу далі по ланцюгу. І це лише один блок програми, а їх буде реалізовано декілька. Тобто це означає, що наша CLI-програма буде декілько рівнева та виконувати багато дій паралельно між собою, які будуть все ж таки перетанись в одній

кінцевій точці і уже в консоль на екрані будуть виводитись усі необхідні дані, а в лог-файл уже буде виводитись більш детальна інформація про певні запущені дії. З цього погляду можна навіть сказати, що це неначе конвеєр, де на кожній зупинці робиться певна дія і уже на виході ми отримуємо готовий до експлуатації продукт.

Звичайно, для того, щоб перевірити працездатність програми та виявити усі можливі помилки у ході її експлуатації, потрібно її протестувати якимось чимось. Для вирішення цієї проблеми були обрані 2 рівні тестування програми: unit-tests та тестування безпосередньо самої програми на користувацькому рівні, але, для початку треба розібратись що таке взагалі unit-tests.

Unit-tests (модульні тести) — процес у програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми. Ідея полягає в тому, щоб писати тести для кожної нетривіальної функції чи методу. Це дозволяє досить швидко перевірити, чи не привела чергова зміна коду до регресії, тобто до появи помилок у вже відтестованих місцях програми, а також полегшує виявлення та усунення таких помилок. Наприклад, оновити бібліотеку, що використовується в проекті, до актуальної версії можна в будь-який момент, прогнавши тести і виявивши несумісності. Складність написання модульних тестів залежить від організації коду. Сильне зачеплення або велика зона відповідальності окремих сутностей (класи для об'єктно орієнтованих мов) можуть ускладнити тестування. Для об'єктів здійснюють зв'язок із зовнішнім світом (мережева взаємодія, файлове введення-виведення і т. д.) слід створювати заглушки. У термінології виділяють «просунуті» заглушки — Mock-об'єкти, які несуть у собі логіку. Також спростити тестування може виділення якнайбільшої частини логіки в чисті функції. Вони ніяк не взаємодіють із зовнішнім світом і їхній результат залежить тільки від вхідних параметрів.

Наша програма буде працювати на конкретному комп’ютері. Тобто, задля її використання потрібно буде скачати та встановити CLI tool та

під’єднати її до консолі операційної системи за допомогою файлу з розширенням *.bat, у якому буде стрічка, котра була описана у пункті 2.2.

Задля зручності назви команд будуть короткі та інтуїтивно зрозумілими, але, щоб CLI ОС зрозуміла, що ми хочемо викликати команди саме з нашої програми, перед назвою команди та параметри, що передаються у неї, потрібно вказати ключове слово «supervisor».

Чому користувачі хотіли би використовувати дану CLI tool?

По-перше, така програма є більш розширена навіть у порівняні зі стандартним Task Manager у Windows від компанії Microsoft.

По-друге, дана CLI tool використовує менше оперативної пам’яті у своїй роботі та менше завантажує процесор.

По-третє, використовуючи простоту текстового редактору та особливості консолі, є можливість переглядати історію запитів та історію сповіщень. Також, є функціонал як перегляду окремих процесів, так і всіх разом у режимі реального часу або ж тільки у даний момент.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНСТРУМЕНТУ ІНТЕРФЕЙСУ КОМАНДНОГО РАДКА ДЛЯ ВДОСКОНАЛЕНОГО АДМІНІСТРУВАННЯ ПРОЦЕСІВ В ОС КОМП'ЮТЕРА

3.1 Мова програмування для програмування

Для CLI програми була обрана мова програмування C#. C# — об'єктно орієнтована мова програмування від компанії Microsoft.

Мова програмування C# була розроблена для власних платформ Microsoft .NET Framework та .NET Core. Приналежність до сімейства мов .NET дає переваги для комбінування цих мов, адже вони усі використовують CLR (common language runtime) та CIL (або MSIL) – Common (Microsoft) intermediate language. Це дозволяє коду написаному на C# компілювати код написаний, наприклад, на Visual Basic.

Також, приналежність до сімейства мов .NET дає доступ до великої сильної бібліотеки класів, адже це сімейство є одним з найпопулярніших у світі, тим самим, звичайні програмісти поповнюють базу бібліотек власними бібліотеками класів завдяки платформі .NET Core з відкритим вихідним кодом (open source platform). Завдяки цьому, стає набагато простіше створювати специфічні бібліотеки класів для специфічних задач.

C# належить до сімейства мов із C-подібним синтаксисом, з яких його синтаксис найбільш близький до C++ та Java. Мова статично типізована і підтримує поліморфізм, перевантаження операторів (включаючи явні та неявні оператори перетворення типів), делегати, властивості, події, змінні, властивості, загальні типи та методи, ітератори, анонімну підтримку замикань Функція, LINQ, Exception, Comment XML формат.

Запозичивши багато у своїх попередників: C++, Delphi, Modula, Smalltalk і особливо Java — C#, спираючись на практику їх використання, виключаючи деякі моделі, які виявилися проблематичними при розробці програмної системи, наприклад, C#, на відміну від C++, робить не підтримує

множинну спадковість класів (одночасно допускаючи кілька реалізацій інтерфейсів).

Особливості мови програмування C# полягають у тому, що вона була розроблена як мова програмування на рівні програми для CLR, і, таким чином, значною мірою залежить від можливостей самого CLR. Спочатку це стосується системи типів C#, що відображає BCL. Наявність чи відсутність певних виразних мовних особливостей залежить від того, чи можна перекласти певні мовні особливості у відповідні структури CLR. Отже, оскільки CLR розвивався з версії 1.1 до версії 2.0, сам C# був значно збагачений; подібні взаємодії повинні з'явитися в майбутньому (однак із випуском C# 3.0 цей шаблон було порушено, так як ця версія C# не опиралася на розширення платформи .NET). Як і всі інші мови, орієнтовані на .NET, CLR надає C# багато функцій, яких не вистачає «класичним» мовам програмування. Наприклад, збір сміття не реалізується в самому C#, а виконується CLR для програм, написаних на C# так само, як програми, написані на VB.NET, J# тощо.

3.2 Середовище програмування

Інтегроване середовище програмування (IDE) —— Комплексні програмні рішення для розробки програмного забезпечення. Зазвичай воно складається з редактора вихідного коду, інструментів для автоматичної компіляції та налагодження програм. Більшість сучасних середовищ розробки здатні до автоматичного заповнення коду.

Деякі середовища розробки включають компілятор, інтерпретатор або обидва (наприклад, NetBeans та Eclipse), а інші не включають жодного (SharpDevelop та Lazarus). Деякі інтегровані середовища розробки включають системи контролю версій або інструменти для полегшення розробки графічного інтерфейсу користувача (GUI) (XCode, Embarcadero Delphi). Багато сучасних ISR включають засоби перевірки класів, перевірки об'єктів, схеми ієрархії класів для полегшення розробки об'єктно-орієнтованого програмного

забезпечення. Інтегровані середовища програмування розроблені з метою максимізації продуктивності програміста, забезпечуючи відповідні інструменти розробки з подібними інтерфейсами як єдину програму, яка запускає весь процес розробки та забезпечує необхідну функціональність для модифікації, компіляції, розгортання та налагодження програмного забезпечення. Протилежним є метод розробки програмного забезпечення, який використовує окремі інструменти, такі як vi, GCC або make.

Одним із завдань інтегрованого середовища програмування є скорочення часу, необхідного для налаштування різних інструментів розробки, замість того, щоб надавати той самий набір інструментів у цілому. Це підвищує продуктивність розробників, коли вивчення роботи IDE відбувається швидше, ніж вивчення всіх інструментів. Крім того, більша інтеграція між вбудованими інструментами може ще більше підвищити продуктивність. Наприклад, код можна проаналізувати безпосередньо під час його редагування, таким чином виявляючи помилки перед перекладом коду.

В ході аналізу та підбору IDE, було обрано найпопулярніше та найзручніше IDE – Visual Studio 2022 від студії Microsoft.

Microsoft Visual Studio — це сімейство продуктів Microsoft, яке включає інтегроване середовище розробки програмного забезпечення та багато інших інструментів. Ці продукти дозволяють розробляти консольні та графічні додатки, включаючи підтримку технології Windows Forms, а також веб-сайти, веб-додатки, веб-сервіси у рідному та керованому коді для Microsoft Windows, Windows Mobile, Windows Phone, Windows Усі підтримувані платформи CE , .NET Framework, .NET Compact Framework і Microsoft Silverlight.

Visual Studio підтримує 36 різних мов програмування та дозволи редагування коду, а також надає підтримку розробки (включаючи пов’язані кроки) приблизно для однієї мови програмування, розробленої для певних мовних служб. Вбудовані мови включають C, C++, C++/CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML і CSS. Додаткова

інформація для інших мов, таких як Python, Ruby, Node.js та багатьох інших, доступна завдяки плагінам.

Хвилясті лінії (рисунок 3.1) вказують на помилки в процесі введення або потенційні проблеми з кодом. Ці візуальні підказки допомагають негайно вирішувати проблеми, не чекаючи створення чи виконання помилок. Якщо ви вкажете на хвилясту лінію, на екрані з'являться додаткові повідомлення про помилки. Ви також можете побачити світловий індикатор у полі зліва, щоб отримати інформацію про швидкі дії для усунення помилки.

Рисунок 3.1 — Візуальні підказки

IntelliSense — це набір функцій, які відображають інформацію про код безпосередньо в редакторі, а в деяких випадках автоматично генерують невеликі фрагменти коду. По суті, це основна документація, вбудована в редактор, що усуває необхідність пошуку інформації в інших джерелах.

На наступному зображенні показано, як IntelliSense відображає список учасників для типу:

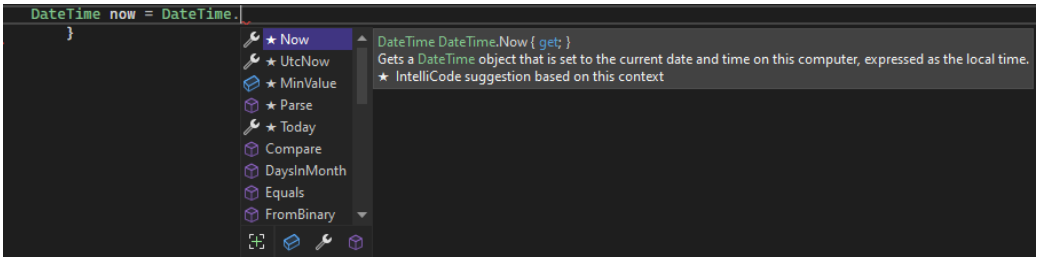


Рисунок 3.2 — IntelliSense

Для швидкого пошуку функцій або елементів коду IDE Visual Studio надає один компонент пошуку (рисунок 3.3) (CTRL + Q).

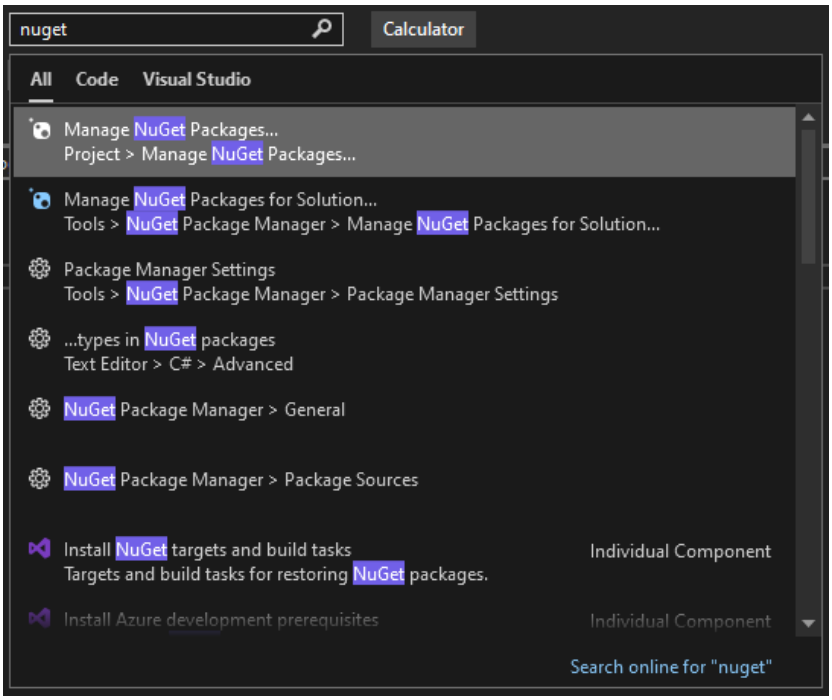


Рисунок 3.3 — пошук

Завдяки таким перевагам, саме Microsoft Visual Studio 2022 було обрано для розробки програми.

3.3 Програмна реалізація CLI програми для керування процесами

Реалізацію CLI tool буде розпочато з розробки інтерфейсів та їхні реалізації у відповідних класах-нащадках.

```
namespace Supervisor.Common.Interfaces
{
    /// <summary> Handles folder which has a configuration files
    4 references
    public interface IFolderHandler
    {
        /// <summary> Determines all new created files
        public event FileEventConfiguration ComponentAdded;
        /// <summary> Determines all deleted files
        public event FileEventConfiguration ComponentDeleted;
        /// <summary> Determines all changed files
        public event FileEventConfiguration ComponentChanged;
        /// <summary> Executes all configuration files in the chosen folder
        4 references
        public IEnumerable<Message> ProcessFiles();
        /// <summary> Tracks the creation of a new files in the chosen folder
        4 references
        public Message OnCreated(object sender, FileSystemEventArgs e);
        /// <summary> Tracks the deletion of files in the chosen folder
        4 references
        public Message OnDeleted(object sender, FileSystemEventArgs e);
        /// <summary> Tracks the changing of existing files in the chosen folder
        4 references
        public Message OnChanged(object sender, FileSystemEventArgs e);
        /// <summary> Returns instances of Supervisor.Utilities.Message with all configu ...
        2 references
        public IEnumerable<Message> GetAll();
    }
}
```

Рисунок 3.4 — Реалізація інтерфейсу IFolderHandler

Уся бізнес-логіка кожної команди поділена та розподілена по ієрархії класів, таких як Commands, ProcessHandler та ProcessReceiver та допоміжних класах, наприклад: Message, ProcessParameter та інші. Також по ієрархічній системі об’єкти цих класів створюються завдяки Dependency Injection, як приватні поля у тілах класів. Усі класи розподілені у різні збірки із різними namespace (простір імен), тобто є загальний namespace «Supervisor» і в залежності від того, яку збірку ми хочемо підключи або до якої збірки хочемо додати новий клас, ми після слова «Supervisor» вказуємо конкретну приналежність. Наприклад: усі інтерфейси розміщені у папці «Interfaces» та мають відповідний namespace «Supervisor.Common.Interfaces». Класи, що відповідають за процеси, мають namespace «Supervisor.Processes» і так далі.

Таким чином, CLI програму можна поділити на 3 блоки:

— «Supervisor.Processes»;

— «Supervisor.Services»;

— «Supervisor.Utilities».

Розглянемо кожен блок більш детальніше.

3.3.1 Supervisor.Processes

Клас ProcessReceiver – це врапер для методів інтегрованого класу Process. Він був створений для можливості створення Unit-tests для класу ProcessHandler.

Adapter (або Wrapper) – шаблон структурного оформлення, призначений для організації функціонального використання об'єктів, за допомогою спеціально розробленого інтерфейсу, котрі не підлягають зміні.

У класі ProcessHandler виконуються усі алгоритми та дії, котрі маніпулюють процесами: закривають їх, відкривають нові, показують інформацію про окремі процеси та інше.

```
public Message Start(string path)
{
    try
    {
        var process = _processReceiver.Start(path);
        if (process != null)
        {
            _logger.Information($"Started a new process {process.Name} (Time: {DateTime.Now})");
            return new Message($"{process.Name} has been started successfully: Id({process.Id})," +
                $" Used memory - {process.Memory / 1024 / 1024} MB", TypeOfMessage.Success);
        }
        return new Message("There is no any executable files by input path." +
            " Check out the the correctness of the path", TypeOfMessage.Error);
    }
    catch (Exception ex)
    {
        _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
        return new Message($"The process can not be started because of error:" +
            $" {ex.Message}", TypeOfMessage.Error);
    }
}
```

Рисунок 3.5 — Демонстрація методу Start() у класі ProcessHandler

Деякі методи приймають вхідні параметри, як наприклад метод Start(), він приймає параметр типу string «path», тобто, це шлях до файлу, процес якого ми хочемо запустити. І в залежності від правильності шляху та файлу, котрий розташований за цим шляхом, ви отримаємо результат виконання команди. Якщо все вказано вірно, то в консоль виведеться інформація про виконаний процес, якщо шлях граматично вказаний вірно, але процес не може бути запущеним, то у консоль виводиться повідомлення про це, але, якщо станеться якась помилка, при виконанні команди, то виведеться повідомлення, що сталась помилка, та текст самої помилки, який був згенерований компілятором.

Подібним кшталтом реалізовані абсолютно усі команди у даному класі. На рисунку 3.5 можна помітити, що повертаючий тип методу – Message, це зроблено в усіх методах даного класу, адже клас Commands реалізований таким чином, що він просто приймає об’єкт типу Message, в якому знаходиться повідомлення, яке треба вивести у консоль, від об’єкту класу ProcessHandler та виводить отриманий текст різними кольорами на екран, в залежності від типу повідомлення. Таким чином, успішні дії виводяться у консоль зеленим кольором, помилки – червоним, а звичайний – білим.

```
/// <summary> Starts any process by file's path
[Command("start")]
0 references
public void Start([Option(0)] string path)
{
    Print(_processHandler.Start(path));
}
```

Рисунок 3.6 — Демонстрація методу Start() у класі Commands

```
/// <summary> Prints the message after executing of command
12 references
private void Print(Message message)
{
    if (message == null)
    {
        return;
    }
    switch (message.TypeOfMessage)
    {
        case TypeOfMessage.Error:
            Console.ForegroundColor = ConsoleColor.Red;
            break;
        case TypeOfMessage.Success:
            Console.ForegroundColor = ConsoleColor.Green;
            break;
        case TypeOfMessage.Common:
            Console.ForegroundColor = ConsoleColor.Gray;
            break;
        case TypeOfMessage.Finished:
            Console.ForegroundColor = ConsoleColor.Yellow;
            break;
    };
    Console.WriteLine(message.ToString());
    Console.ForegroundColor = ConsoleColor.Gray;
}
```

Рисунок 3.7 — Демонстрація методу Print() у класі Commands

Як видно на рисунку 3.6, метод Start() складається взагалі з одного рядка, це зроблено для того, щоб не дублювати логіку для кожного окремого методу. Через це, створено метод Print(), котрий приймає в параметрах об’єкт типу Message і, в залежності від рівня повідомлення, обирає колір яким вивести повідомлення, яке зберігається у цьому об’єкті (код даного методу зображено на рисунку 3.7).

Підсумовуючи даний блок програми, можна сказати, що усю «роботу» з уже існуючими процесами (або з тими, котрі створили не ми завдяки *.xml файлам) робить саме клас ProcessHandler, а клас Commands лише обробляє результат виконання цього класу, та виводить його на екран. Даний принцип розподілення «обов’язків» є реалізація патерну MVC.

MVC (Model-View-Controller) — це архітектурний шаблон проектування програми, що використовується під час проектування та розробки програмного забезпечення. Шаблон розбиває систему на три взаємопов’язані частини: модель даних, представлення (інтерфейс користувача) і модуль керування. Він використовується для відокремлення даних (моделі) від інтерфейсу користувача (зовнішнього вигляду), щоб зміни в інтерфейсі користувача мали мінімальний вплив на обробку даних, а зміни можна було вносити в модель даних без зміни інтерфейсу користувача.

3.3.2 Supervisor.Services

Даний блок складається із семи класів (5 взаємопов’язаних та 2 це класи-врапери): ConfigurationHolder, FileConfigurationWatcher, FolderHandler, OptionsConfig, ThreadHandler, DirectoryReceiver, XmlHandler.

ConfigurationHolder — клас, в якому зберігається конфігурація усіх згенерованих процесів у файлах з розширенням .xml.

FileConfigurationWatcher — клас, завдяки якому у реальному часі спостерігаються зміни у папці із конфігурацією.

FolderHandler — клас, котрий обробляє запити об’єкту класу FileConfigurationWatcher та здійснює маніпуляції з конфігурацією і записує їх у об’єкт класу ConfigurationHolder.

OptionsConfig — клас, в якому зберігається налаштування для об’єкта класу FileConfigurationWatcher, котрі зчитуються з файлу з розширенням .json.

ThreadHandler — клас, який керує багатопоточністю для оновлювання інформації у консолі у режимі реального часу.

DirectoryReceiver — wrapper для метода Directory.GetFiles().

XmlHandler — wrapper для класу XmlDocument та його метода Load().

У класі FolderHanlder виконується уся логіка для обробки усіх змін, які були помічені у папці з конфігурацією процесів, а також, при старті програми, він збирає усю інформацію із папки з конфігурацією та виконує певні дії за вказаним алгоритмом. Тобто якщо якийсь файл був видалений, доданий або

змінений у середині, FolderHandler про це дізнається та виконає наступні алгоритми, котрі будуть підходити до утвореної ситуації.

```
4 references
public Message OnCreated(object sender, FileSystemEventArgs e)
{
    if (e.ChangeType == WatcherChangeTypes.Created)
    {
        ManipulationOfConfiguration(e.Name, TypeOfAction.Add);
        ComponentAdded?.Invoke(sender, e);
        return new Message($"{e.Name} has been deleted", TypeOfMessage.Common);
    }
    return null;
}
4 references
public Message OnDeleted(object sender, FileSystemEventArgs e)
{
    if (e.ChangeType == WatcherChangeTypes.Deleted)
    {
        ManipulationOfConfiguration(e.Name, TypeOfAction.Remove);
        ComponentDeleted?.Invoke(sender, e);
        return new Message($"{e.Name} has been deleted", TypeOfMessage.Common);
    }
    return null;
}
4 references
public Message OnChanged(object sender, FileSystemEventArgs e)
{
    if (e.ChangeType == WatcherChangeTypes.Changed)
    {
        ManipulationOfConfiguration(e.Name, TypeOfAction.Change);
        ComponentChanged?.Invoke(sender, e);
        return new Message($"{e.Name} has been changed", TypeOfMessage.Common);
    }
    return null;
}
```

Рисунок 3.8 — Демонстрація методів, котрі реагують на зміні у папці з конфігурацією

На рисунку 3.8 можна помітити, що в тілі усіх трьох методів є виклик метода ManipulationOfConfiguration(), який приймає два параметри: ім’я файлу, з якими відбулись зміни та зміну типу enum TypeOfAction. Ця зміна дозволяє встановити який саме тип маніпуляції був здійснений: файл доданий, файл видалений або файл змінений. Цей метод був створений, щоб не дублювати код у кожний новий метод, тобто, зменшення розмірів тіла цього класу, та через принцип “якщо код повторюється, винеси його у окремий метод”.

Підсумовуючи даний блок, можна сказати, що клас FolderHandler є саме тим класом, що робить усю “брудну роботу” та передає дані та результати виконання іншим класам. Наприклад: методи OnCreated(), OnChanged(), OnDeleted() повертають об’єкт типу Message, який використовує тільки клас

Commands. Це означає, що саме через ці методи йде передача повідомлень, котрі виводяться у консолі.

```
/// <summary> Loads the XML document with the configuration and determines an ac ...
3 references
private Message ManipulationOfConfiguration(string fileName, TypeOfAction action)
{
    var uri = _fileConfigurationWatcher.Path + "\\" + fileName;
    _xmlHandler.Load(uri);
    foreach (XmlDocument document in _xmlHandler.Document)
    {
        if (document != null)
        {
            ProcessConfiguration processConfiguration;
            var root = document.DocumentElement;
            foreach (XmlNode node in root.ChildNodes)
            {
                if (action == TypeOfAction.Add)
                {
                    processConfiguration = new ProcessConfiguration(
                        node.Attributes.GetNamedItem("Name").Value, uri);
                    _configurationHolder.Add(processConfiguration);
                    return new Message($"{fileName} has been added successfuly", TypeOfMessage.Success);
                }
                if (action == TypeOfAction.Remove)
                {
                    processConfiguration = _configurationHolder.FindConfigurationByName(
                        node.Attributes.GetNamedItem("Name").Value);
                    _configurationHolder.Remove(processConfiguration);
                    return new Message($"{fileName} has been removed successfuly", TypeOfMessage.Success);
                }
                if (action == TypeOfAction.Change)
                {
                    processConfiguration = _configurationHolder.FindConfigurationByURI(uri);
                    _configurationHolder.Change(processConfiguration);
                    return new Message($"{fileName} has been changed successfuly", TypeOfMessage.Success);
                }
            }
        }
    }
    return null;
}
```

Рисунок 3.9 — Демонстрація методу ManipulationOfConfiguration

3.3.3 Supervisor.Utilities

Останній блок складається із допоміжний класів, котрі мають від сили 2 методи або навіть не мають їх, а просто зберігають певні дані у властивостях різних типів.

Delegates — це зовсім не клас, а звичайний файл з розширенням .cs, котрий містить спеціальні делегати, котрі використовуються для налаштування подій у класі FolderHandler. Цей файл був створений для зручності розробника, щоб мати їх у окремому доступі.

Message – клас, котрий містить в собі повідомлення та “ранг” цього повідомлення, як уже було вказано вище.

ProcessConfiguration – клас, котрий містить в собі тільки 2 властивості – Name та BaseURI. Name – ім’я самого файлу, BaseURI – повний шлях файлу.

ProcessExtension – статичний клас, створений для методів-розширення для зручності у написані коду.

ProcessParameter – клас, що містить в собі ім’я, ID та займану процесом пам’ять у комп’ютері.

Підсумовуючи останній блок, можна сказати, що в ньому містяться додаткові класи, котрі були створенні тільки для зручності розробника і вони не мають вагомого впливу на саму програму.

3.4 Тестування програми

Щоб протестувати дану програму, були розроблені спеціальні Unit-tests із перевірками на усі очікувані результати, такі як: вдале виконання методу із відповідним повідомленням на екран, вдале повідомлення про помилку, коли щось було введено неправильно або некоректно та також, неконтрольовані виключення (exceptions).

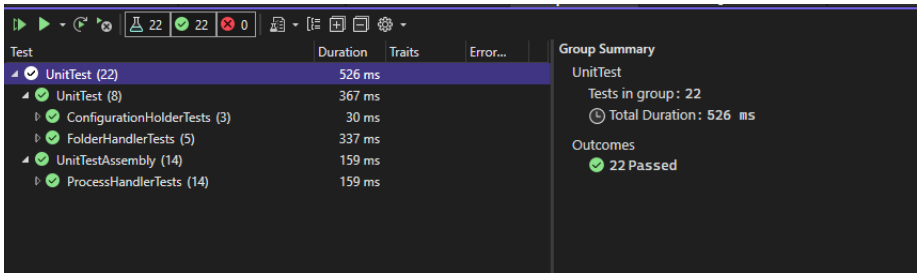


Рисунок 3.10 — результат виконання Unit-Tests

Також, програма була протестована вручну, тобто у реальній роботі з реальними процесами, в результаті чого, помилок помічено не було.

Отже, тестування пройшло успішно та програма готова до релізу та використанню користувачами.

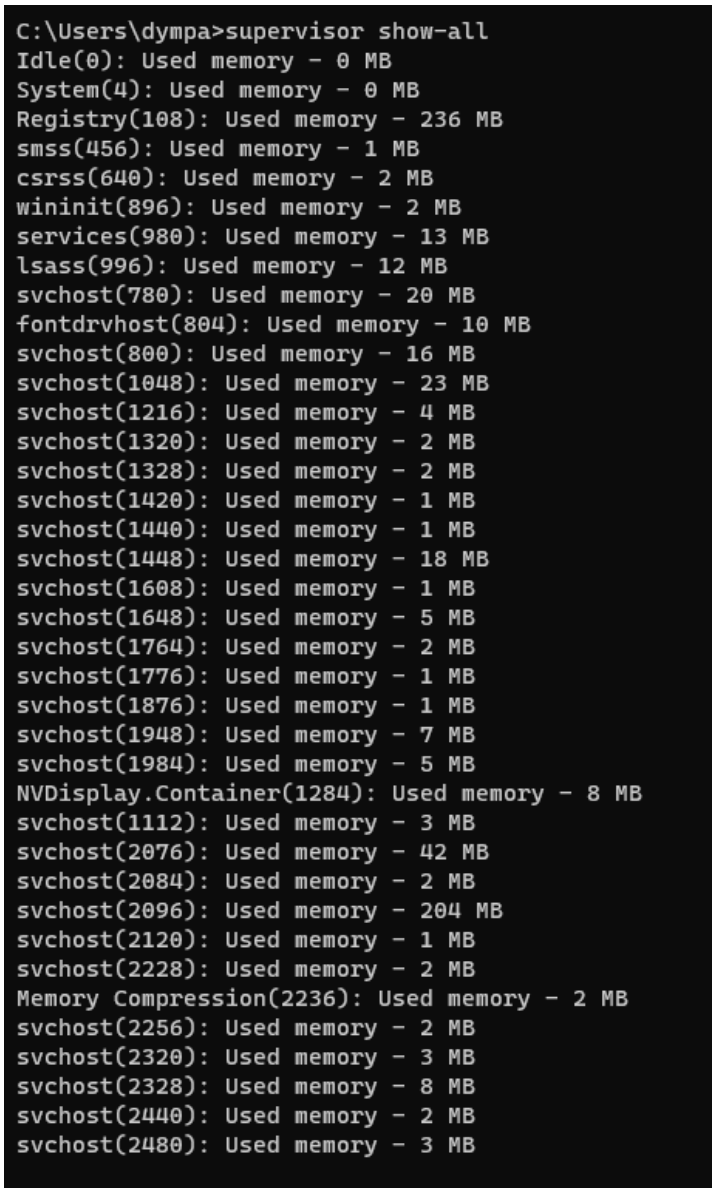


Рисунок 3.11 — демонстрація команди supervisor show-all

ВИСНОВКИ

У даній дипломній роботі був розроблений інструмент інтерфейсу для вдосконаленого адміністрування процесів у ОС комп’ютера. Для реалізації задачі була обрана мова програмування C#, оскільки вона найбільше підходила для вирішення задачі, були проаналізовані та використанні основні принципи ООП саме в цій мові для створення CLI-програми та вона виявилась найзручнішою серед інших мов, також вона має сильну бібліотеку класів.

Були розглянуті доцільні середовища програмування (IDE) для створення CLI програми та на основі переваг та унікальних функцій була обрана IDE Microsoft Visual Studio 2022, котра ідеально підійшла для виконання задачі.

Після вибору IDE та мови програмування була спроектована архітектура моделі програми, прораховано усе що потрібно для максимально компактного та вірного з точки зору менеджменту оперативної пам’яті вирішення задачі. Реалізована уся архітектура програми разом із допоміжними класами та методами.

Не менше уваги було приділено також бібліотеки яка буде поєднувати CLI tool з консоллю системи для можливості використовувати саме системну консоль для користування програмою, що в значній мірі спрощує це використання, адже нічого зайвого скачувати не потрібно. З цього також витікає, що не потрібно розробляти власний фреймворк, що значно спрощує розробку програми та зменшує використання оперативної пам’яті.

Була змодельованна архітектура програми у повному представленні, в якому і була реалізована та створена. Після усіх тестів, які підтвердили відсутність помилок, програма вийшла у реліз та може використовуватись будь яким користувачем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Димпалов Є. І., Савицька Л. А. «РОЗРОБКА CLI TOOLS НА ПЛАТФОРМІ .NET CORE»

2. Руководство по ASP.NET Core 6 [Електронний ресурс] — Режим доступу: https://metanit.com/sharp/aspnet6/

3. .NET Core – что это такое, возможности [Електронний ресурс] — Режим доступу: https://www.xelent.ru/blog/vozmozhnosti-net-core/

4. How to track a process [Електронний ресурс] — Режим доступу: https://stackoverflow.com/questions/26796907/how-to-track-a-process

5. Мартін Р. С. Чистий код. Створення і рефакторинг коду [Текст]. — 2008. — С. 230-257.

6. Ріхтер Д. CLR via C# 4-те видання. Програмування на платформі .NET Core 3.1, на мові c#. 2013. — С. 134-287.

7. Троелсен Е. Мова програмування C# 6-те видання. 2013. — С. 30-100.

8. .NET CLI [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/core/tools/

9. Visual Studio [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/ru-ru/visualstudio/get-started/visual-studio-ide?view=vs-2022

10. ConsoleAppFramework [Електронний ресурс] — Режим доступу: https://github.com/Cysharp/ConsoleAppFramework

11. dotnet command - .NET CLI [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet

12. .NET Core Command-line Interface [Електронний ресурс] — Режим доступу: https://www.tutorialsteacher.com/core/net-core-command-line-interface

13. .NET (and .NET Core) - introduction and overview [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/core/introduction

14. ASP.NET MVC Pattern [Електронний ресурс] — Режим доступу: https://dotnet.microsoft.com/en-us/apps/aspnet/mvc

15. Object Oriented Programming [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/ru-ru/dotnet/csharp/fundamentals/tutorials/oop

16. Введение в .NET Core [Електронний ресурс] — Режим доступу: https://habr.com/ru/company/microsoft/blog/245901/

17. Фленов М. Библия C# (2-е издание). 2012. —– С. 250.

18. C# docs - get started, tutorials, reference. [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/csharp/

19. C# — Объектно-ориентированный язык программирования [Електронний ресурс] —– Режим доступу: https://habr.com/ru/hub/csharp/

20. Что такое C# [Електронний ресурс] — Режим доступу: http://web.spt42.ru/index.php/chto-takoe-c

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

проф., д.т.н.. Азаров О.Д..

" " 2022 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської дипломної роботи

“ Інструмент інтерфейсу командного рядка для вдосконаленого адміністрування процесів в ОС комп’ютера ”

08-23.БДР.025.00.000 ТЗ

Науковий керівник: к.т.н., доцент каф. ОТ

_____ Савицька Л. А.

Студент групи 2КІ-18Б

_____ Димпалов Є. І.

ВНТУ 2022

1 Підстава для виконання бакалаврської дипломної роботи (БДР)

1.1 Актуальність дослідження у напрямку дипломної роботи, яка полягає у тому, щоб тестування та експлуатація специфічний програм або навіть звичайні користувачі мали повний доступ для керування процесами та конфігурувати нові власні процеси для певних завдань та потреб.

1.2 Наказ про затвердження теми БДР.

2 Мета БДР і призначення розробки

2.1 Мета роботи — є розробка CLI tool для вдосконаленого адміністрування процесів у середовищі програмування Visual Studio 2022;

2.2 Призначення розробки — CLI tool для вдосконаленого адміністрування процесів.

3 Вихідні дані для виконання БДР

3.1 Проведення аналізу існуючих CLI tool;

3.2 Розробка структури CLI tool;

3.3 На основі структурних та функціональних схем здійснено розрахунок та моделювання елементів CLI tool.

4 Вимоги до виконання БДР

Головна вимога — реалізувати CLI tool для вдосконаленого адміністрування процесів.

5 Етапи БДР та очікувані результати

Етапи роботи та очікувані результати приведено в таблиці етапів БДР.

6 Матеріали, що подаються до захисту БДР

До захисту подаються: пояснювальна записка БДР, графічні і ілюстративні матеріали, протокол попереднього захисту БДР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до БДР українською та іноземною мовами, довідка про відповідність оформлення БДР діючим вимогам.

Таблиця — Етапи БДР

№ етапу	Назва етапу	Термін виконання		Примітка
		початок	кінець	
1	Отримання завдання	08.02.22		
2	Огляд і аналіз інтерфейсу командного рядка та його реалізації	09.02.22	16.02.22	
3	Аналіз фундаментальних принципів ООП в C#	17.02.22	19.02.22	
4	Дослідження методів розробки CLI tool	19.03.22	05.03.22	
5	Аналіз існуючих бібліотек класів в платформі .NET Core	06.03.22	20.03.22	
6	Моделювання архітектури програми	21.03.22	05.04.22	
7	Реалізація першого блоку програми	06.04.22	10.04.22	
8	Реалізація другого блоку програми	11.04.22	02.05.22	
9	Реалізація третього блоку програми	03.05.22	07.05.22	
10	Тестування програми на виявлення помилок	08.05.22	25.05.22	
11	Оформлення пояснювальної записки та ілюстративного матеріалу	26.05.22	10.06.22	
12	Перевірка якості виконання бакалаврської роботи та усунення недоліків	13.06.22		

7 Порядок контролю виконання та захисту БДР

Виконання етапів графічної та розрахункової документації БДР контролюється науковим керівником згідно зі встановленими термінами.

Захист БДР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання БДР

8.1 При оформлюванні БДР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— Методичні вказівки. Кафедра обчислювальної техніки 2022;

— Документами на які посилаються у вище вказаних.

8.2 Порядок виконання БДР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21»

ДОДАТОК Б

Лістинг програми

```
public class Commands : ConsoleAppBase
    {
        private readonly IProcessHandler _processHandler;
        private readonly IFolderHandler _folderHandler;
        private readonly IThreadsHandler _threadHandler;
        private readonly ManagementEventWatcher _eventStarted;
        private readonly ManagementEventWatcher _eventStoped;
        /// <summary>
        /// Boolean instance for "Watch" mode
        /// </summary>
        private bool _watchMode;
        public Commands(IProcessHandler processHandler, IFolderHandler folderHandler, IThreadsHandler threadHandler)
        {
            _processHandler = processHandler;
            _folderHandler = folderHandler;
            _threadHandler = threadHandler;
            _eventStarted = new ManagementEventWatcher(new WqlEventQuery("SELECT * FROM Win32_ProcessStartTrace"));
            _eventStoped = new ManagementEventWatcher(new WqlEventQuery("SELECT * FROM Win32_ProcessStopTrace"));
            Help();
        }
    /// <summary>
        /// Starts any process by file's path
        /// </summary>
        /// <param name="path"> Path which links on the file on the PC</param>
        [Command("start")]
```

```
        public void Start([Option(0)] string path)
        {
            Print(_processHandler.Start(path));
        }
        /// <summary>
        /// Finishes any started process on the PC
        /// </summary>
        /// <param name="processName"> A name of chosen process which started on the PC</param>
        /// <returns></returns>
        [Command("kill-by-name")]
        public void KillByName([Option(0)] string processName)
        {
            Print(_processHandler.Kill(processName));
        }
        /// <summary>
        /// Finishes any started process on the PC
        /// </summary>
        /// <param name="id">An id of chosen process which started on the PC</param>
        /// <returns></returns>
        [Command("kill-by-id")]
        public void KillById([Option(0)] int id)
        {
            Print(_processHandler.Kill(id));
        }
        /// <summary>
        /// Finishes any started process tree on the PC
        /// </summary>
        /// <param name="processName"></param>
```

```
        /// <returns></returns>
        [Command("kill-tree-by-name")]
        public void KillTreeByName([Option(0)] string processName)
        {
            Print(_processHandler.KillTree(processName));
        }
        /// <summary>
        /// Finishes any started process tree on the PC
        /// </summary>
        /// <param name="id"></param>
        [Command("kill-tree-by-id")]
        public void KillTreeById([Option(0)] int id)
        {
            Print(_processHandler.KillTree(id));
        }
        /// <summary>
        /// Shows all chosen process values
        /// </summary>
        /// <param name="processName">Name of chosen process which started on the PC</param>
        /// <returns></returns>
        [Command("show-by-name")]
        public void ShowById([Option(0)] string processName)
        {
            Print(_processHandler.Show(processName));
        }
        /// <summary>
        /// Shows all processes
        /// </summary>
        /// <returns></returns>
```

```
        [Command("show-all")]
        public void ShowAllProcesses()
        {
            foreach (var message in _processHandler.ShowAllProcesses())
            {
                Print(message);
            }
        }
        /// <summary>
        /// Shows all chosen processes values
        /// </summary>
        /// <param name="id"></param>
        /// <returns></returns>
        [Command("show-by-id")]
        public void ShowByName([Option(0)] int id)
        {
            Print(_processHandler.Show(id));
        }
        /// <summary>
        /// Real time update of active processes
        /// </summary>
        [Command("watch")]
        public void Watch()
        {
            _watchMode = true;
            _threadHandler.StartThreads(EnableWatchMode,
DisableWatchMode);
        }
        /// <summary>
        /// Executes configurated processes
```

```
/// </summary>
[Command("Get-all")]
public void GetAll()
{
    var items = _folderHandler.GetAll();

    #region Figures out if there's any configurated process

    items.GetEnumerator().MoveNext();
    var isNotEmpty = items.GetEnumerator().Current;
    if (isNotEmpty == null)
    {
        Print("There's no any configurated process");
        return;
    }
    items.GetEnumerator().Reset();

    #endregion

    foreach (var item in items)
    {
        Print(item);
    }
    return;
}
/// <summary>
/// Shows all available commands
/// </summary>
[Command("help")]
public void Help()
```

```
            {
                Print("\n'start' <path> - starts a new process\n");
                Print("'kill-by-id' <id> - kills an open process by id\n");
                Print("'kill-by-name' <name> - kills an open process by name\n");
                Print("'kill-tree-by-id' <id> - kills an open process tree by id\n");
                Print("'kill-tree-by-name' <name> - kills an open process tree by name\n");
                Print("'show-by-id' <id> - shows an open process information by id\n");
                Print("'show-by-name' <name> - shows an open process information by name\n");
                Print("'show-all' - shows all open processes information\n");
                Print("'watch' - shows all open processes information in real time\n" +
                    "<WARNING> to disable watch mode - print 'exit' <WARNING>\n");
                Print("'get-all' - shows all configurated processes (this command may be used only while 'watch mode' is active)\n");
            }
        private void Print(Message message)
            {
                if (message == null)
                {
                    return;
                }
                switch (message.TypeOfMessage)
                {
                    case TypeOfMessage.Error:
                        Console.ForegroundColor = ConsoleColor.Red;
                        break;
                    case TypeOfMessage.Success:
```

```
                Console.ForegroundColor = ConsoleColor.Green;
                break;
            case TypeOfMessage.Common:
                Console.ForegroundColor = ConsoleColor.Gray;
                break;
            case TypeOfMessage.Finished:
                Console.ForegroundColor = ConsoleColor.Yellow;
                break;
        };
        Console.WriteLine(message.ToString());
        Console.ForegroundColor = ConsoleColor.Gray;
    }
    /// <summary>
    /// --Created only for method "Help"-- Prints message
    /// </summary>
    /// <param name="message">Message which will be printed</param>
    private void Print(string message)
    {
        Console.WriteLine(message);
    }
    /// <summary>
    /// Disables watch mode
    /// </summary>
    private void DisableWatchMode()
    {
        while (Console.ReadLine().ToString() != "exit") { }
        UnSubscribeEvents();
        Print(new StringBuilder().Append('*', 10).Append(" Watch mode
disabled ").Append('*', 10)
            .AppendLine().ToString());
```

```
        }
        /// <summary>
        /// Execution of the "Watch mode"
        /// </summary>
        private void EnableWatchMode()
        {
            SubscribeEvents();
            Print(new StringBuilder().Append('*', 10).Append(" Watch mode ").Append('*', 10)
                .AppendLine().ToString());
            PrintXMLExceptions();
            while (_watchMode)
            {
                Print($"Time: {DateTime.Now}");
                GetAll();
                Thread.Sleep(12000);
                Print("\n*Update*\n");
            }
        }
        /// <summary>
        /// Catches every single process what's been started in "Watch mode"
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void ProcessStarted(object sender, EventArrivedEventArgs e)
        {
            Print(new Message($"A new process has been started: " +
                $"{e.NewEvent.Properties["ProcessName"].Value}", TypeOfMessage.Success));
            Thread.Sleep(10);
```

```
        }
        /// <summary>
        /// Catches every single process what's been finished in the "Watch mode"
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void ProcessStoped(object sender, EventArrivedEventArgs e)
        {
            Print(new Message($"The process has been stoped: " +
                $"{e.NewEvent.Properties["ProcessName"].Value}", TypeOfMessage.Finished));
            Thread.Sleep(10);
        }
        /// <summary>
        /// Subscribes all necessaries events for Watch Mode
        /// </summary>
        private void SubscribeEvents()
        {
            _eventStarted.EventArrived += ProcessStarted;
            _eventStoped.EventArrived += ProcessStoped;
            _folderHandler.ComponentAdded += _folderHandler.OnCreated;
            _folderHandler.ComponentChanged += _folderHandler.OnChanged;
            _folderHandler.ComponentDeleted += _folderHandler.OnDeleted;
            _eventStarted.Start();
            _eventStoped.Start();
        }
        /// <summary>
        /// Unsubscribes all necessaries events for Watch Mode
        /// </summary>
```

```
private void UnSubscribeEvents()
{
    _watchMode = false;
    _eventStarted.EventArrived -= ProcessStarted;
    _eventStoped.EventArrived -= ProcessStoped;
    _folderHandler.ComponentAdded -= _folderHandler.OnCreated;
    _folderHandler.ComponentChanged -= _folderHandler.OnChanged;
    _folderHandler.ComponentDeleted -= _folderHandler.OnDeleted;
    _eventStarted.Stop();
    _eventStoped.Stop();

}
/// <summary>
/// Prints an XML exceptions
/// </summary>
private void PrintXMLExceptions()
{
    var messages = _folderHandler.ProcessFiles();
    if (messages != null)
    {
        Print("Exceptions: \n");
        foreach (var message in messages)
        {
            Print(message);
        }
        Print("");
    }
}
```

```
public class ProcessHandler : IProcessHandler
```

```
{
    private readonly IProcessReceiver _processReceiver;
    private static ILogger _logger;
    public ProcessHandler(IProcessReceiver processReceiver)
    {
        _processReceiver = processReceiver;
        _logger = Log.Logger;
    }
    public Message Start(string path)
    {
        try
        {
            var process = _processReceiver.Start(path);
            if (process != null)
            {
                _logger.Information($"Started a new process {process.Name} (Time: {DateTime.Now})");
                return new Message($"{process.Name} has been started successfully: Id({process.Id})," +
                    $" Used memory - {process.Memory / 1024 / 1024} MB", TypeOfMessage.Success);
            }
            return new Message("There is no any executable files by input path." +
                " Check out the the correctness of the path", TypeOfMessage.Error);
        }
        catch (Exception ex)
        {
            _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
```

```
                return new Message($"The process can not be started because of error:" +
                    $" {ex.Message}", TypeOfMessage.Error);
            }
        }
        public Message Kill(string processName)
        {
            try
            {
                _logger.Information($"{_processReceiver.Kill(processName, false).Name} was killed" +
                    $" (Time: {DateTime.Now})");
                return new Message($"The process ({processName}) has been successfully exited",
                    TypeOfMessage.Success);
            }
            catch (Exception ex)
            {
                _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
                return new Message($"The process can not be exited because of error:" +
                    $" {ex.Message}", TypeOfMessage.Error);
            }
        }
        public Message Kill(int id)
        {
            try
            {
                var process = _processReceiver.Kill(id, false);
```

```
                _logger.Information($"{process.Name} was killed (Time: {DateTime.Now})");
                return new Message($"The process ({process.Name}) has been successfully exited",
                    TypeOfMessage.Success);
            }
            catch (Exception ex)
            {
                _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
                return new Message($"The process can not be exited because of error: {ex.Message}",
                    TypeOfMessage.Error);
            }
        }
        public Message KillTree(string processName)
        {
            try
            {
                _logger.Information($"{_processReceiver.Kill(processName, true)} tree was killed" +
                    $" (Time: {DateTime.Now})");
                return new Message($"The process tree ({processName}) has been successfully exited",
                    TypeOfMessage.Success);
            }
            catch (Exception ex)
            {
                _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
                return new Message($"The process can not be exited because of error:" +
```

```
                $" {ex.Message}", TypeOfMessage.Error);
            }
        }
        public Message KillTree(int id)
        {
            try
            {
                var process = _processReceiver.Kill(id, true);
                _logger.Information($"{process.Name} tree was killed (Time: {DateTime.Now})");
                return new Message($"The process tree ({process.Name}) has been successfully exited",
                    TypeOfMessage.Success);
            }
            catch (Exception ex)
            {
                _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
                return new Message($"The process can not be exited because of error: {ex.Message}",
                    TypeOfMessage.Error);
            }
        }
        public Message Show(string processName)
        {
            try
            {
                var process = _processReceiver.GetProcess(processName);
                return new Message($"{processName}: Id ({process.Id})," +
                    $" Used memory - {process.Memory / 1024 / 1024}MB", TypeOfMessage.Common);
```

```
            }
            catch (Exception ex)
            {
                _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
                return new Message($"The process {processName.ToUpper()}" +
                    $" is not open or your input of process name is not correct",
TypeOfMessage.Error);
            }
        }
        public Message Show(int id)
        {
            try
            {
                var process = _processReceiver.GetProcess(id);
                return new Message($"{process.Name}: Id ({process.Id})," +
                    $" Used memory - {process.Memory / 1024 / 1024}MB",
TypeOfMessage.Common);
            }
            catch (Exception ex)
            {
                _logger.Error($"{ex.Message} (Time: {DateTime.Now})");
                return new Message("The process is not open or your input of
process name is not correct",
                    TypeOfMessage.Error);
            }
        }
        public IEnumerable<Message> ShowAllProcesses()
        {
            var processes = _processReceiver.GetProcesses();
            foreach (var process in processes)
```

```
                {
                    yield return new Message($"{process.Name}({process.Id}):" +
                        $" Used memory - {process.Memory / 1024 / 1024} MB", TypeOfMessage.Common);
                }
            }
        }
        /// <summary>
        /// A wrapper for System.Diagnostics.Process
        /// </summary>
        public class ProcessReceiver: IProcessReceiver
        {
            public ProcessParameter Start(string path)
            {
                var process = Process.Start(path);
                return new ProcessParameter(process.ProcessName, process.Id, process.PeakPagedMemorySize64);
            }
            public ProcessParameter GetProcess(string processName)
            {
                var processes = Process.GetProcesses();
                foreach (var process in processes)
                {
                    var name = process.ProcessName;
                    if (processName.ToLower() == process.ProcessName.ToLower())
                    {
                        return new ProcessParameter(name, process.Id, process.PeakPagedMemorySize64);
                    }
                }
```

```
            return null;
        }
        public ProcessParameter GetProcess(int id)
        {
            var process = Process.GetProcessById(id);
            return new ProcessParameter(process.ProcessName, process.Id,
process.PeakPagedMemorySize64);
        }
        public ProcessParameter[] GetProcesses()
        {
            var processes = Process.GetProcesses();
            var parameters = new ProcessParameter[processes.Length];
            for (int i = 0; i < processes.Length; i++)
            {
                parameters[i] = new ProcessParameter(processes[i].ProcessName,
processes[i].Id, processes[i].PeakPagedMemorySize64);
            }
            return parameters;
        }
        public ProcessParameter Kill(int id, bool tree)
        {
            var process = Process.GetProcessById(id);
            var parameter = new ProcessParameter(process.ProcessName,
process.Id, process.PagedMemorySize64);
            process.Kill(tree);
            return parameter;
        }
        public ProcessParameter Kill(string processName, bool tree)
        {
            var process = Process.GetProcessesByName(processName);
```

```
                var parameter = new ProcessParameter(process[0].ProcessName, process[0].Id, process[0].PagedMemorySize64);
                process[0].Kill(tree);
                return parameter;
            }
        }
    /// <summary>
        /// Holding the configuration of the new processes into Supervisor.Utilities.ProcessConfiguration instances
        /// </summary>
        public class ConfigurationHolder : IConfigurationHolder
        {
            private readonly List<ProcessConfiguration> _configuration;
            public ConfigurationHolder()
            {
                _configuration = new List<ProcessConfiguration>();
            }
            public int Count => _configuration.Count;
            public ProcessConfiguration this[int index] => _configuration[index];
            public List<ProcessConfiguration> ProcessConfigurations => _configuration;
            public void Add(ProcessConfiguration configuration)
            {
                if (configuration == null)
                {
                    return;
                }
                _configuration.Add(configuration);
            }
```

```
        public ProcessConfiguration Remove(ProcessConfiguration configuration)
        {
            if (configuration == null)
            {
                return null;
            }
            _configuration.Remove(configuration);
            return configuration;
        }
        public ProcessConfiguration Change(ProcessConfiguration processConfiguration)
        {
            if (processConfiguration == null)
            {
                return null;
            }
            for (var i = 0; i < _configuration.Count; i++)
            {
                if (_configuration[i].BaseURI == processConfiguration.BaseURI)
                {
                    _configuration[i] = processConfiguration;
                    return _configuration[i];
                }
            }
            return null;
        }
        public ProcessConfiguration FindConfigurationByName(string value)
        {
            foreach (var configuration in _configuration)
```

```
        {
            if (configuration.BaseURI == value)
            {
                return configuration;
            }
        }
        return null;
    }
    public ProcessConfiguration FindConfigurationByURI(string value)
    {
        foreach (var configuration in _configuration)
        {
            if (configuration.BaseURI == value)
            {
                return configuration;
            }
        }
        return null;
    }
    public IEnumerator GetEnumerator()
    {
        return _configuration.GetEnumerator();
    }
}
/// <summary>
/// Wrapper for System.IO.Directory class
/// </summary>
public class DirectoryReceiver: IDirectoryReceiver
{
    public string[] GetFiles(string path, string searchPattern)
```

```
            {
                return Directory.GetFiles(path, searchPattern);
            }
        }

    public class FileConfigurationWatcher : IFileConfigurationWatcher
        {
            private readonly FileSystemWatcher _watcher;
            private readonly IOptionsMonitor<OptionsConfig> _optionsMonitor;
            public      FileConfigurationWatcher(IOptionsMonitor<OptionsConfig>
optionsMonitor)
            {
                _optionsMonitor = optionsMonitor;
                _watcher = new FileSystemWatcher();
                LoadData();
            }
            FileSystemWatcher  IFileConfigurationWatcher._fileSystemWatcher  {
get; }
            public string Path
            {
                get
                {
                    return _optionsMonitor.CurrentValue.Path;
                }
            }
            public string Filter
            {
                get
                {
                    return "*.xml";
```

```
            }
        }
        private void LoadData()
        {
            _watcher.Path = Path;
            _watcher.NotifyFilter = NotifyFilters.Attributes
                                    | NotifyFilters.CreationTime
                                    | NotifyFilters.DirectoryName
                                    | NotifyFilters.FileName
                                    | NotifyFilters.LastWrite
                                    | NotifyFilters.Security
                                    | NotifyFilters.Size;
            _watcher.Filter = Filter;
            _watcher.IncludeSubdirectories = true;
            _watcher.EnableRaisingEvents = true;
        }
    }

/// <summary>
    /// Provides folder handling
    /// </summary>
    public class FolderHandler : IFolderHandler
    {
        private readonly IFileConfigurationWatcher _fileConfigurationWatcher;
        private readonly IConfigurationHolder _configurationHolder;
        private readonly IDirectoryReceiver _directoryReceiver;
        private readonly IXmlHandler _xmlHandler;
        public FolderHandler(IConfigurationHolder configurationHolder,
IDirectoryReceiver directoryReceiver, IFileConfigurationWatcher
fileConfigurationWatcher, IXmlHandler xmlHandler)
```

```
        {
            _configurationHolder = configurationHolder;
            _directoryReceiver = directoryReceiver;
            _fileConfigurationWatcher = fileConfigurationWatcher;
            _xmlHandler = xmlHandler;
        }
        public event FileEventConfiguration ComponentAdded;
        public event FileEventConfiguration ComponentDeleted;
        public event FileEventConfiguration ComponentChanged;
        public IEnumerable<Message> ProcessFiles()
        {
            var xmlFiles =
_directoryReceiver.GetFiles(_fileConfigurationWatcher.Path,
_fileConfigurationWatcher.Filter);
            var xmlDocuments = new List<XmlDocument>();
            if (xmlFiles != null)
            {
                foreach (var file in xmlFiles)
                {
                    var message = _xmlHandler.Load(file);
                    if (message == null)
                    {
                        xmlDocuments.Add(_xmlHandler.Document);
                    }
                    else
                    {
                        yield return message;
                    }
                }
            }
```

```
            ExecuteConfigurations(xmlDocuments);
        }
        public Message OnCreated(object sender, FileSystemEventArgs e)
        {
            if (e.ChangeType == WatcherChangeTypes.Created)
            {
                ManipulationOfConfiguration(e.Name, TypeOfAction.Add);
                ComponentAdded?.Invoke(sender, e);
                return new Message($"{e.Name} has been deleted",
TypeOfMessage.Common);
            }
            return null;
        }
        public Message OnDeleted(object sender, FileSystemEventArgs e)
        {
            if (e.ChangeType == WatcherChangeTypes.Deleted)
            {
                ManipulationOfConfiguration(e.Name, TypeOfAction.Remove);
                ComponentDeleted?.Invoke(sender, e);
                return new Message($"{e.Name} has been deleted",
TypeOfMessage.Common);
            }
            return null;
        }
        public Message OnChanged(object sender, FileSystemEventArgs e)
        {
            if (e.ChangeType == WatcherChangeTypes.Changed)
            {
                ManipulationOfConfiguration(e.Name, TypeOfAction.Change);
                ComponentChanged?.Invoke(sender, e);
```

```
                return new Message($"{e.Name} has been changed",
TypeOfMessage.Common);
            }
            return null;
        }
        public IEnumerable<Message> GetAll()
        {
            if (_configurationHolder.Count == 0)
            {
                yield return null;
            }
            foreach (ProcessConfiguration item in _configurationHolder)
            {
                var process = ProcessExtension.Start(item);
                yield return new Message($"The process:
{process.ProcessName}({process.Id})",
                    TypeOfMessage.Common);
            }
        }
        private ICollection<ProcessConfiguration>
ExecuteConfigurations(ICollection<XmlDocument> collection)
        {
            if (collection.Count == 0)
            {
                return null;
            }
            foreach (var document in collection)
            {
                if (document != null)
                {
```

```
                    var root = document.DocumentElement;
                    foreach (XmlNode node in root.ChildNodes)
                    {
                        _configurationHolder.Add(new ProcessConfiguration
                            (node.Attributes.GetNamedItem("Name").Value,
document.BaseURI));
                    }
                }
            }
            return _configurationHolder.ProcessConfigurations;
        }
        private enum TypeOfAction
        {
            Add,
            Remove,
            Change
        };
        /// <summary>
        /// Loads the XML document with the configuration and determines an
action to do
        /// </summary>
        /// <param name="path">The path of current XML document</param>
        /// <param name="action">Determines an action: "add", "delete" or
"change" -- other parameters rerurns NULL</param>
        /// <returns>Supervisor.Utilites.ProcessConfiguration isntance if XML
document exists and filled correctly
        /// -- otherwise returns NULL --</returns>
        private Message ManipulationOfConfiguration(string fileName,
TypeOfAction action)
        {
```

```
                var uri = _fileConfigurationWatcher.Path + "\\" + fileName;
                _xmlHandler.Load(uri);
                foreach (XmlDocument document in _xmlHandler.Document)
                {
                    if (document != null)
                    {
                        ProcessConfiguration processConfiguration;
                        var root = document.DocumentElement;
                        foreach (XmlNode node in root.ChildNodes)
                        {
                            if (action == TypeOfAction.Add)
                            {
                                processConfiguration = new ProcessConfiguration(
                                    node.Attributes.GetNamedItem("Name").Value, uri);
                                _configurationHolder.Add(processConfiguration);
                                return new Message($"{fileName} has been added
successfuly", TypeOfMessage.Success);
                            }
                            if (action == TypeOfAction.Remove)
                            {
                                processConfiguration                                =
_configurationHolder.FindConfigurationByName(
                                    node.Attributes.GetNamedItem("Name").Value);
                                _configurationHolder.Remove(processConfiguration);
                                return new Message($"{fileName} has been removed
successfuly", TypeOfMessage.Success);
                            }
                            if (action == TypeOfAction.Change)
                            {
```

```
                    processConfiguration =
_configurationHolder.FindConfigurationByURI(uri);
                    _configurationHolder.Change(processConfiguration);
                    return new Message($"{fileName} has been changed
successfuly", TypeOfMessage.Success);
                }
            }
        }
    }
    return null;
  }
}

/// <summary>
  /// Realize a settings for real time watching for chosen folder with
configuration
  /// </summary>
  public class OptionsConfig : IOptionsConfig
  {
    public string Path { get; set; }
  }
}

/// <summary>
  /// Handling of System.Threads instances
  /// </summary>
  public class ThreadsHandler: IThreadsHandler
  {
    private Thread _enableThread;
    private Thread _disableThread;
```

```
        public ThreadsHandler()
        {

        }
        public void StartThreads(ThreadStart enableThread, ThreadStart disableThread)
        {
            _enableThread = new Thread(enableThread);
            _disableThread = new Thread(disableThread);
            _enableThread.Start();
            _disableThread.Start();
        }
    }

    public class XmlHandler : IXmlHandler
    {
        public XmlDocument Document { get; private set; }
        public string Name { get; private set; }
        public XmlHandler(){}
        public Message Load(string path)
        {
            Name = null;
            Document = new XmlDocument();
            var splitted = path.Split("\\");
            Name = splitted[splitted.Length - 1];;
            try
            {
                Document.Load(path);
                return null;
            }
```

```
                catch (Exception e)
                {
                    return new Message($"{Name} has thrown an exception: {e.Message}", TypeOfMessage.Error);
                }
            }
        }

    public delegate Message FileEventConfiguration(object sender, FileSystemEventArgs e);
        public delegate Message FileEventChangedConfiguration(object sender, FileSystemEventArgs e);
    /// <summary>
        /// Outputs the messages for logs
        /// </summary>
        public class Message
        {
            private readonly string _message;
            public Message(string message, TypeOfMessage typeOfMessage)
            {
                _message = message;
                TypeOfMessage = typeOfMessage;
            }
            /// <summary>
            /// Type of message
            /// </summary>
            public TypeOfMessage TypeOfMessage { get; }
            public override string ToString()
            {
                return _message;
```

```
        }
    }
    /// <summary>
    /// Types of output messages
    /// </summary>
    public enum TypeOfMessage
    {
        Common,
        Error,
        Success,
        Finished
    };

public class ProcessConfiguration
    {
        public ProcessConfiguration(string name, string baseURI)
        {
            Name = name;
            BaseURI = baseURI;
        }
        public string Name { get; }
        public string BaseURI { get; }
    }

public static class ProcessExtension
    {
        public static Process Start(ProcessConfiguration processConfiguration)
        {
            return Process.Start(processConfiguration.Name);
        }
```

```
    }

/// <summary>
    /// Contains parameters of process --- this class uses only for testing
    /// </summary>
    public class ProcessParameter
    {
        public ProcessParameter(string name, int id, long memory)
        {
            Name = name;
            Id = id;
            Memory = memory;
        }
        /// <summary>
        /// The name of process
        /// </summary>
        public string Name { get; }
        /// <summary>
        /// The id of process
        /// </summary>
        public int Id { get; }
        /// <summary>
        /// The used memory of process
        /// </summary>
        public long Memory { get; }
    }
```

ДОДАТОК В

Ілюстративний матеріал

ІНСТРУМЕНТ ІНТЕРФЕЙСУ КОМАНДНОГО РЯДКА ДЛЯ ВДОСКОНАЛЕНОГО АДМІНІСТРУВАННЯ ПРОЦЕСІВ В ОС КОМП'ЮТЕРА

Бакалаврська дипломна робота
спеціальність 123 — "Комп'ютерна інженерія"

Виконав: Димпалов Є. І.
Ст. групи 2КІ-18б
Науковий керівник:
Савицька Людмила Анатоліївна

1

АКТУАЛЬНІСТЬ

Інтерфейс командного рядка (command-line interface, скорочено - CLI) — це текстовий інтерфейс користувача та комп'ютера, в якому комп'ютерні інструкції можна давати лише шляхом введення рядків тексту (команд) з клавіатури. Інтерфейс командного рядка можна порівняти з програмними системами управління або різними реалізаціями GUI (graphic user interface - з англ. графічний інтерфейс користувача). Формат вихідної інформації інтерфейсу командного рядка нестандартизований; зазвичай виводиться простий текст, а також графіка, аудіо виведення тощо.

Мета та завдання дослідження.

Метою даної дипломної роботи є покарщення на технічному та користувацькому рівнях управління процесами в ОС Windows

Для досягнення поставленої мети необхідно розв'язати такі завдання:

1. Проаналізувати фундаментальні принципи ООП;
2. Проаналізувати переваги та недоліки текстового інтерфейсу;
3. Проаналізувати існуючі бібліотеки для підв'язки програми до системи;
4. Виконати моделювання програми;
5. Виконати розробку CLI-програми за допомогою мови програмування C#;
6. Провести тестування програми та проаналізувати отримані результати.

2

ОБ'ЄКТ ДОСЛІДЖЕННЯ –

процес розробки удосконаленого CLI для адміністрування процесів

ПРЕДМЕТ ДОСЛІДЖЕННЯ –

удосконалена CLI-програма для адміністрування процесів та створення нових процесів

3

ПРАКТИЧНЕ ЗНАЧЕННЯ ОДЕРЖАНИХ РЕЗУЛЬТАТІВ

1. Полягає у можливості конфігурувати нові процеси для специфічних програм.
2. Управління увімкненими процесами та запуск вимкнених за допомогою файлів з розширенням .exe.
3. Перегляд повної інформації ввімкнених процесів.

4

ПОСТАНОВКА ЗАДАЧІ

Задача даної роботи полягає у розробці інструменту інтерфейсу командного рядку для вдосконаленого адміністрування процесів в ОС комп'ютера, а саме:

- Аналіз фундаментальних принципів ООП;
- Аналіз переваг та недоліків текстового інтерфейсу;
- Аналіз існуючих бібліотек для підв'язки програми до системи комп'ютера;
- Проектування архітектури програмного модуля;
- Програмна реалізація;
- Тестування програми;

КОМАНДИ

Дана програма використовує команди задля здійснення певних виведених алгоритмів, які потрібні користувачу. Тобто, у будь-якій консолі, що встановлена у системі комп'ютера, за допомогою ключового слова «supervisor» та назви команди з переданими даними (за необхідності), буде виклик саме необхідних для користувача дій.

ВИБІР МОВИ ПРОГРАМУВАННЯ

Мова програмування C# була розроблена для власних платформ Microsoft .NET Framework, .NET Core та ASP.NET. Приналежність до сімейства мов .NET дає доступ до великої сильної бібліотеки класів, адже це сімейство є одним з найпопулярніших у світі, тим самим, звичайні програмісти поповнюють базу бібліотек власними бібліотеками класів завдяки платформі .NET Core, що має відкритий вихідний код.

7

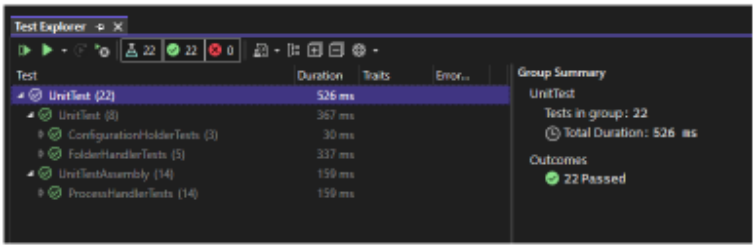
СХЕМА АЛГОРИТМУ CLI tool

8

Для тестування CLI tool були створені Unit-tests та проведені тести безпосередньо із самою програмою зі статусу користувача.

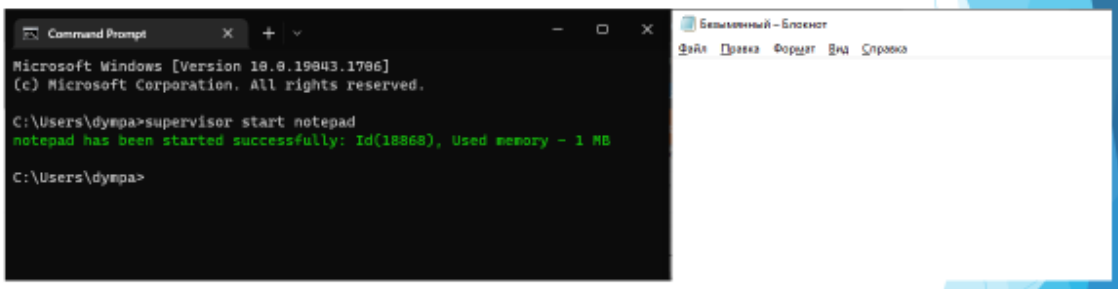
9

Результат Unit-tests



10

Результат тестування у консолі



11

АПРОБАЦІЯ НАУКОВИХ ДОСЛІДЖЕНЬ

За результатами бакалаврської дипломної роботи опубліковано тезу на конференції:

«Молодь в науці: дослідження, проблеми, перспективи (МН-2022)» — ІНСТРУМЕНТ ІНТЕРФЕЙСУ КОМАНДНОГО РЯДКА ДЛЯ ВДОСКОНОЛАНЕГОГО АДМІНІСТРУВАННЯ ПРОЦЕСІВ В ОС КОМП'ЮТЕРА

12

ВИСНОВКИ

У даній дипломній роботі був розроблений інструмент інтерфейсу для вдосконаленого адміністрування процесів у ОС комп'ютера. Для реалізації задачі була обрана мова програмування C#, оскільки вона найбільше підходила для вирішення задачі та вона виявилась найзручнішою серед інших мов, також вона має сильну бібліотеку класів.

Не менше уваги було приділено також бібліотеки яка буде поєднувати CLI tool з консоллю системи для можливості використовувати саме системну консоль для користування програмою, що в значній мірі спрощує це використання, адже нічого зайвого скачувати не потрібно. З цього також витікає, що не потрібно розробляти власний фреймворк, що значно спрощує розробку програми та зменшує використання оперативної пам'яті.

13

ДЯКУЮ ЗА УВАГУ!

14

ДОДАТОК Г

ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Інструмент інтерфейсу командного рядка для вдосконалення для вдосконалення адміністрування процесів в ОС комп'ютера.

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки

Показники звіту подібності Unicheck

Оригінальність 95,5% Схожість 4,5%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.
- ☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- ☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи _____ _____
(підпис) (прізвище, ініціали)

Керівник роботи _____ _____
(підпис) (прізвище, ініціали)