

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
(назва навчального закладу)

Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ
Дисципліна «Технології розроблення програмного забезпечення»

Курс 3 Група ІА-12 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Симко Андрій Ігорович

(прізвище, ім'я, по батькові)

1. Тема роботи “Installer generator”
2. Строк здачі студентом закінченої роботи 30.12.2023
3. Вихідні дані до роботи: перелік функцій, які має виконувати Installer generator: генератор інсталяційних пакетів повинен мати спосіб налаштування файлів, що входять в установку, установки вікон з інтерактивними можливостями (галочка – створити ярлик на робочому столі, ввести в поле деякі дані, наприклад, ліцензійний ключ, вибір локалізації, вибір шляху створення інсталятора). Генератор повинен вивести один файл .msi.
4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці): Постановка задачі, огляд та аналіз існуючих рішень, визначення вимог до застосунку, опис сценаріїв використання, побудова концептуальної моделі системи, вибір мови програмування та платформи реалізації застосунку, вибір та розробка бази даних, вибір патернів програмування.

Додатки:

Додаток А-діаграма класів, Додаток Б-код проекту

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): Діаграма класів, діаграма прецедентів, діаграма розгортання системи, схема архітектури системи, скріншоти, що демонструють фрагменти коду, знімки екрану із зображенням графічного інтерфейсу застосунку.
6. Дата видачі завдання 05.10.2022

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Видача теми курсової роботи	22.09.2023	
2.	Загальний опис проекту	25.09.2023	
3.	Огляд існуючих рішень	2.10.2023	
4.	Визначення вимог до системи	9.10.2023	
5.	Визначення сценаріїв використання	16.10.2023	
6.	Концептуальна модель системи	30.10.2023	
7.	Вибір БД	7.11.2023	
8.	Вибір мови програмування та IDE	7.11.2023	
9.	Проектування розгортання системи	14.11.2023	
10.	Структура БД	17.11.2023	
11.	Визначення специфікації системи	20.11.2023	
12.	Вибір та обґрунтування патернів проектування	26.11.2023	
13.	Реалізація проекту	01.12.2023	
14.	Захист курсової		

Студент _____
(підпис)

Симко Андрій
(Ім'я ПРІЗВИЩЕ)

Керівник _____
(підпис)

Колеснік В.М
(Ім'я ПРІЗВИЩЕ)

«____» _____ 20__ р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема “Installer generator”

Курсова робота

З дисципліни «Технології розроблення програмного забезпечення»

Керівник

Викладач Колеснік В. М.

«Допущений до захисту»

(Особистий підпис керівника)

« » _____ 2024 р.

Захищений з оцінкою

(оцінка)

Члени комісії:

(особистий підпис)

(особистий підпис)

Виконавець

ст. Симко А. І.

залікова книжка № ІА – 1225

гр. ІА-12

(особистий підпис виконавця)

« » _____ 2024 р.

(розшифровка підпису)

(розшифровка підпису)

Зміст

ВСТУП.....	3
1. ПРОЄКТУВАННЯ СИСТЕМИ.....	4
1.1. Огляд існуючих рішень	4
1.2. Загальний опис проєкту.....	5
1.3. Вимоги до застосунків системи	6
1.4. Сценарії використання системи	9
1.5. Концептуальна модель системи	11
1.6. Вибір бази даних.....	12
1.7. Вибір мови програмування та середовища розробки	13
1.8. Проєктування розгортання системи	15
2. РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ.....	16
2.1. Структура бази даних.....	16
2.2. Архітектура системи.....	17
2.3. Інструкція користувача	24
ВИСНОВКИ.....	31
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	32
ДОДАТКИ.....	34
Додаток А.....	34
Додаток Б	35

В сучасному технологічному світі особливо важливою є потреба в ефективних інструментах для створення пакетів для встановлення програм, особливо в сфері інформаційних технологій. Ця потреба викликає необхідність розробки генератора пакетів для встановлення, який дозволить гнучко керувати вмістом та налаштуваннями процесу встановлення.

Мета полягає у розробці десктопного додатку для створення інсталяційних пакетів, що дозволить користувачам налаштовувати файли .msi або .exe та обирати параметри для їхньої установки, такі як створення ярликів на робочому столі та визначення шляху для встановлення програми. Цей підхід дає користувачам можливість ефективно управляти процесом установки програм, а також пристосовувати їх до різних операційних систем і вимог.

Таким чином, ця курсова робота спрямована на створення десктопного додатку, який дозволить користувачам легко і ефективно створювати налаштовані інсталяційні пакети, що відповідають їхнім індивідуальним вимогам та оптимізують процес установки програмного забезпечення.

1. ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

В контексті розробки інсталяційних пакетів, сучасний ринок програмного забезпечення пропонує різноманітні інструменти, проте кожен з них має свої особливості та обмеження. Найбільш поширеними інструментами для створення інсталяційних пакетів є такі програми, як Inno Setup, NSIS (Nullsoft Scriptable Install System), та Windows Installer XML (WiX) Toolset.

Inno Setup – це безкоштовний інсталятор для Windows програм. Він пропонує широкий спектр функцій, включаючи підтримку створення одного виконуваного файлу для установки, кастомізацію інтерфейсу установника та підтримку скриптів. Однак, Inno Setup не дозволяє створювати .msi файли та може виявитися обмеженим у більш складних сценаріях установки.

NSIS забезпечує гнучкість при створенні інсталяційних скриптів та дозволяє створювати компактні і швидкі установники. Однак, його скриптова мова може бути важкою для розуміння новачками, а також NSIS не створює .msi файли, що обмежує його використання в деяких корпоративних середовищах.

WiX Toolset – це більш потужний інструмент, який дозволяє створювати .msi файли та має багатий набір функцій для складних установок. Проте, використання WiX вимагає глибшого розуміння XML та процесів Windows Installer, що робить його менш доступним для початківців.

Існуючі рішення часто вимагають значних зусиль для налаштування та можуть бути недостатньо гнучкими або інтуїтивно зрозумілими для користувачів, які не мають досвіду у створенні інсталяційних пакетів. Тому, існує потреба в розробці більш простого у використанні, але при цьому потужного інструменту генерації інсталяційних пакетів, який зможе задовольнити потреби як початківців, так і досвідчених розробників. Такий

інструмент повинен включати інтерактивні можливості налаштування інсталяційних пакетів.

1.2. Загальний опис проєкту

Проект - це десктопний додаток, розроблений на Python з використанням бібліотеки tkinter, призначений для створення інсталяційних пакетів. За допомогою інтерфейсу користувача, реалізованого з використанням tkinter, користувачі можуть легко вводити необхідну інформацію для налаштування інсталятора. На основі введених даних створюється структура WiX файлу для створення .msi інсталяційних пакетів, і для додаткової функціональності також використовується Inno Setup для створення .exe файлів.

Процес створення .msi файла відбувається за допомогою Windows Installer XML (WiX) Toolset. Цей процес включає всі параметри, визначені користувачем, як-от вибір місця установки, налаштування файлів для майбутнього інсталятора, а також інші важливі аспекти, які необхідні для належної установки програмного продукту.

Після завершення процесу створення інсталятора, відомості про цей процес у вигляді логу автоматично записуються в Базу даних та Лог-файл. Для зберігання цих даних використовується SQLite, що забезпечує надійне та ефективне управління логами. Це дозволяє проводити моніторинг та аналіз процесу створення інсталяційних пакетів, а також збирати важливу інформацію для подальшого вдосконалення застосунку.

Таким чином, застосунок є ефективним інструментом для створення налаштованих інсталяційних пакетів, що надає гнучкість у використанні та високий рівень контролю над процесом створення інсталяторів.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

Діаграма прецедентів системи представлена на рис. 1.1.

Діаграма варіантів використання – це граф спеціального вигляду, який є графічною нотацією для представлення конкретних варіантів використання, акторів, можливо деяких інтерфейсів, і відносин між цими елементами.

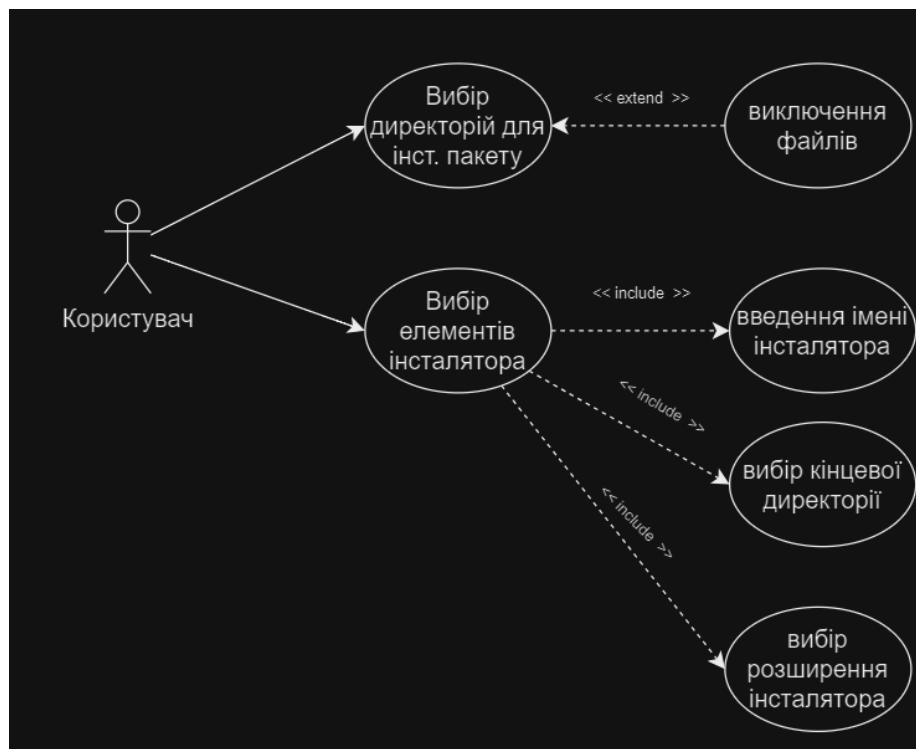


Рисунок 1.1 - Діаграма прецедентів

Генератор інсталяційних пакетів має бути здатний надати користувачам зручні засоби для створення індивідуально налаштованих інсталяційних пакетів (.msi або .exe файлів). Система забезпечує користувачів інструментами для вибору компонентів програми, які мають бути включені до інсталяційного пакету, визначення архітектури цільової системи, налаштування параметрів інсталяції, таких як визначення директорії для встановлення, а також можливість введення інформації необхідної для налаштування інсталятора, включаючи іменування та інші ключові опції.

Користувачі можуть створювати логічну структуру інстальатора, що включає перелік файлів, необхідних для установки, та інтерфейс інстальатора.

Система забезпечує наступні функціональні вимоги:

- Вибір директорії та файлів, які включатимуться до інсталяційного пакету.
- Вибір шляху кінцевої директорії.
- Налаштування опцій інстальатора, включаючи введення імені інстальатора.

Діаграма послідовностей представлена на рис. 1.2.

Діаграма послідовностей – це тип діаграми взаємодії, яка використовується для відображення порядку та умов взаємодії між об'єктами в певному сценарії.

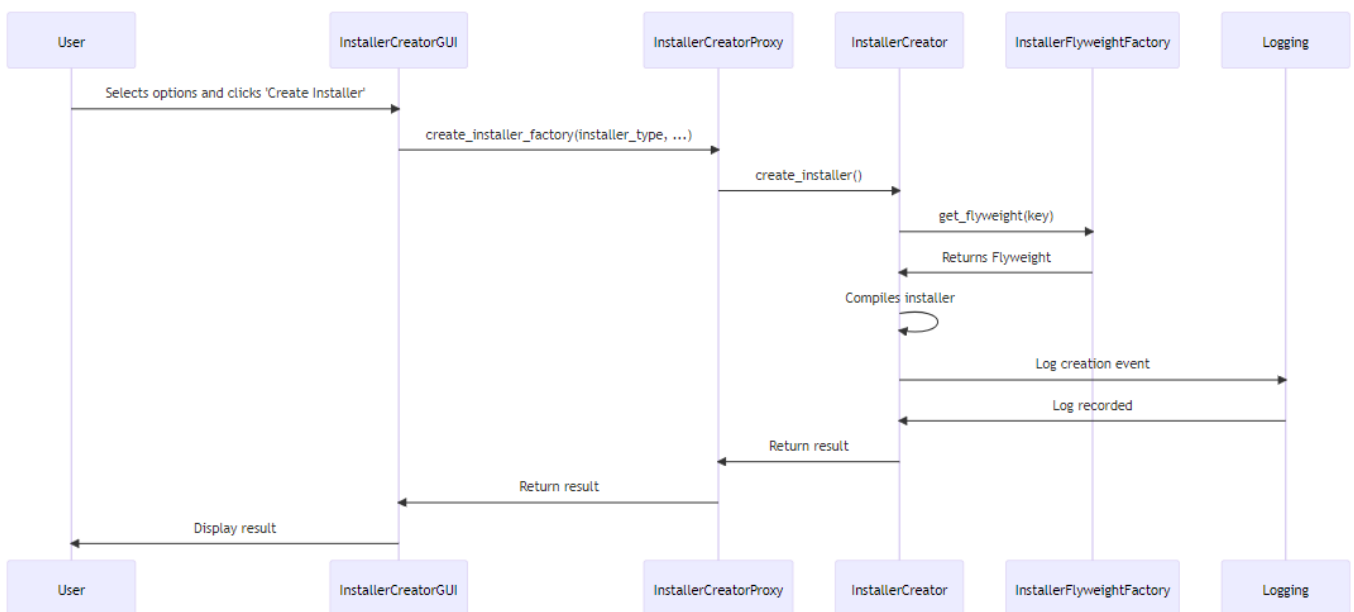


Рисунок 1.2 - Діаграма послідовностей

Діаграма станів представлена на рис. 1.3.

Діаграма станів – це тип діаграми, яка використовується для відображення всіх можливих станів, через які може проходити об'єкт або система, а також переходів між цими станами.

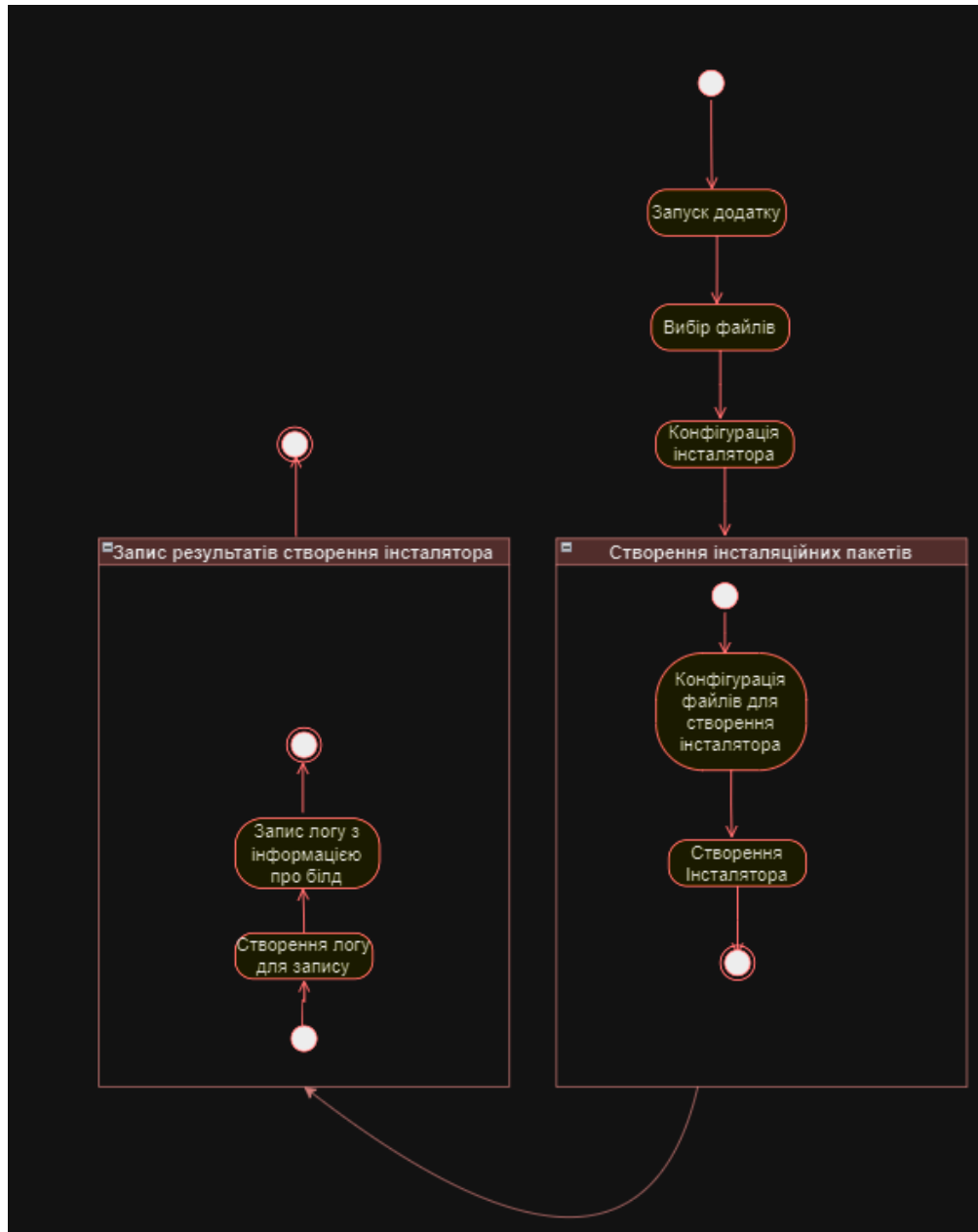


Рисунок 1.3 - Діаграма станів

1.3.2. Нефункціональні вимоги до системи

Система має бути простою і зручною у використанні, зі зрозумілим інтерфейсом, який дозволяє користувачам легко налаштовувати інсталяційні пакети без необхідності розширеного досвіду у сфері ІТ. Вона повинна бути надійною та ефективною, забезпечуючи швидке створення інсталяційних пакетів з мінімальними затримками.

Нефункціональні вимоги включають:

- Забезпечення зручності користувачів без зниження продуктивності.
- Надійність роботи системи та можливість її відновлення після помилок або збоїв.
- Висока швидкість обробки запитів та створення інсталяційних пакетів.
- Легкість оновлення та підтримки програмного забезпечення.

Ці нефункціональні вимоги повинні гарантувати, що система буде легкою у використанні та не буде вимагати від користувачів спеціальних технічних знань або додаткових ресурсів для виконання своїх основних функцій.

1.4. Сценарії використання системи

Таблиця 1.1 - Сценарій використання “Вибір файлів для інсталяції”

Назва	Вибір файлів для інсталяції.
Передумови	Відкриття застосунку.
Післяумови	Користувач отримає список обраних файлів.
Сторони, що взаємодіють	Користувач взаємодіє з інтерфейсом.
Опис	Користувач обирає папку, зі списку файлів у цій папці обирає потрібні файли.

Основний потік подій	<ol style="list-style-type: none"> 1. Користувач перебуває у вікні “Вибрати файл”. 2. Користувач обирає директорію. 3. Користувач обирає потрібні файли зі списку файлів в директорії.
Виняткові ситуації	Відсутні
Примітки	Відсутні

Таблиця 1.2- Сценарій використання “Вибір параметрів”

Назва	Вибір параметрів
Передумови	Користувач обрав потрібні файли для інсталятора.
Післяумови	Усі необхідні параметри були обрані.
Сторони, що взаємодіють	Користувач взаємодіє з інтерфейсом.
Опис	Користувач обирає параметри для інсталятора: вводить ім'я інсталятора, визначає можливість створення ярлика на робочому столі, обирає тип інсталятора та обирає кінцеву директорію.
Основний потік подій	<ol style="list-style-type: none"> 1) Користувач перебуває у налаштуванні параметрів інсталятора. 2) Користувач вводить ім'я інсталятора в текст-бокс. 3) Користувач обирає чек-бокс для можливості створення ярлика на робочому столі.

Проект складається з графічного інтерфейсу (реалізованого за допомогою Tkinter), за допомогою якого користувач обирає файли та параметри для інсталятора. Після цього створюється об'єкт інсталятора за допомогою якого білдери формують файл. З цього файлу далі створюється сам файл інсталятора .msi або .exe. Для створення інсталятора використовуються інструменти WiX Toolset та Inno Setup. По завершенню процесу створення, генерується лог з інформацією про успішне створення інсталятора, який записується в Базу даних та лог-файл.

1.6. Вибір бази даних

Для розробки та створення бази даних проєкту було використано SQLite, що є вбудованою СУБД для локального зберігання даних. SQLite дозволяє структурувати та зберігати дані у локальному файлі бази даних, не потребуючи окремого сервера для роботи. Він підтримує SQL для виконання операцій з базою даних, забезпечуючи точний контроль над схемою даних та можливість виконувати операції вставки, вибору, оновлення та видалення даних.

Основні переваги MongoDB:

- Простота використання: SQLite легко встановлюється і використовується. Це вбудована СУБД, тому вона не вимагає окремого сервера, і ви можете працювати з нею прямо на своєму пристрої без складностей у налаштуванні.
- Надійність: SQLite відомий своєю високою надійністю. Дані зазвичай зберігаються в одному файлі бази даних, що сприяє унікальній транзакційній системі і надійному зберіганню даних.
- Мала витратність ресурсів: SQLite має невеликі вимоги до ресурсів, що робить його ідеальним вибором для використання на мобільних пристроях та вбудованих системах.

- Кросплатформенність: SQLite підтримується на багатьох операційних системах, включаючи Windows, macOS, Linux і мобільні платформи, що робить його кросплатформеним і додатково зручним для розробників.
- Відкритий код: SQLite випущений під ліцензією Public Domain, що дозволяє використовувати його в комерційних і вільних проектах без обмежень.
- Наявність багатьох бібліотек: SQLite має різноманітні бібліотеки та розширення для різних мов програмування, що дозволяє легко інтегрувати його у ваші проекти.

Таким чином, вибір SQLite для проекту генерування інсталяційних пакетів обумовлений потребою у швидкій, масштабованій та гнучкій базі даних, яка здатна забезпечити ефективне управління логами та конфігураціями інсталяційних пакетів. Завдяки використанню SQLite проект може легко адаптуватися до зростаючих вимог та обсягів даних, забезпечуючи високу продуктивність та легкість у використанні.

1.7. Вибір мови програмування та середовища розробки

Для розробки проекту "Installer Generator," призначеного для створення інсталяційних пакетів, ми обрали мову програмування Python. Python - це високорівнева, інтерпретована мова програмування, яка відома своєю простотою використання та багатофункціональністю. Завдяки великій кількості бібліотек і фреймворків, Python дозволяє легко розробляти різні типи застосунків, включаючи десктопні, веб-додатки та сервіси. Ось декілька переваг використання Python:

- Велика стандартна бібліотека: Python поставляється з обширною стандартною бібліотекою, яка містить багато корисних модулів і функцій для різних завдань. Це спрощує розробку і дозволяє використовувати готові рішення.

- Кросплатформенність: Python підтримується на різних операційних системах, включаючи Windows, macOS і Linux, що дозволяє створювати кросплатформенні застосунки без зайвих зусиль.
- Розширюваність: Python дозволяє інтегрувати код на мовах C, C++, а також використовувати багато інших мов програмування, що розширює можливості і забезпечує широкий спектр застосувань.
- Велика спільнота та підтримка: Python має активну та велику спільноту розробників, що означає наявність безлічі ресурсів, форумів і бібліотек для вирішення будь-яких завдань.

Як середовище розробки було обрано PyCharm, одне з найпопулярніших середовищ для розробки на мові програмування Python.. PyCharm надає розробникам широкий набір функцій для зручної та ефективної роботи:

- Інтегроване рішення: PyCharm інтегрує в собі всі необхідні інструменти для розробки, від редактора коду з підказками та контекстною підтримкою до інструментів для налагодження і профілювання.
- Підтримка Wix Toolset: Для створення інсталяційних пакетів Pycharm може інтегруватися з Wix Toolset, що надає можливість створювати інсталятори безпосередньо з середовища розробки. Це використовувалось для тестування різних можливостей інструментів Wix.
- Інтеграція з віртуальними середовищами: PyCharm дозволяє легко створювати та керувати віртуальними середовищами Python для ізолювання проектів та залежностей.
- Інтеграція з SQLite: PyCharm дозволяє легко інтегрувати SQLite, надаючи потужні інструменти для роботи з базами даних безпосередньо з IDE.

Обрання Python та PyCharm для розробки проєкту "Installer Generator"

забезпечує високу продуктивність, безпеку та гнучкість, необхідні для створення надійного та ефективного програмного рішення.

1.8. Проєктування розгортання системи

Діаграма розгортання використовується для ілюстрації обчислювальних вузлів, що задіяні під час виконання програмного забезпечення, а також компонентів та об'єктів, які на них працюють. Компоненти на діаграмі символізують активні частини коду в момент виконання програми. Компоненти, які не мають активної репрезентації у процесі роботи програми, на такій діаграмі зазвичай не зображуються. В основі діаграми розгортання лежить відображення активних компонентів системи.

У контексті UML діаграма розгортання демонструє розподіл фізичних артефактів по вузлах, які зазвичай представлені у вигляді тривимірних блоків. Ці вузли можуть включати в себе підвузли, зображені як вкладені блоки, і можуть концептуально відображати множину фізичних вузлів, як от групу серверів баз даних, які працюють разом у кластері.

Діаграма розгортання системи буде представлена на рисунку 1.3:

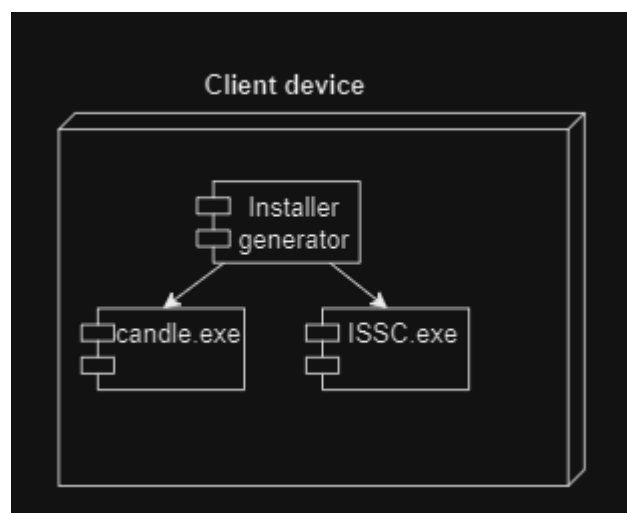


Рисунок 1.3 - Діаграма розгортання системи

2. РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура бази даних

Проект, є десктопним додатком "Installer Generator", що дозволяє користувачам створювати індивідуалізовані інсталяційні пакети. В рамках реалізації цього проекту було створено базу даних, яка веде записи про всі генеровані інсталяційні пакети.

Кожен запис логу містить дату та час створення (time), тип логування(level) тип інсталятора та стан білду(message). Ця структура є важливою для відстеження історії створення інсталяційних пакетів та можливих помилок, що виникли під час цього процесу.

Для зберігання даних використовується SQLite, яка ідеально підходить для збереження документоподібних записів і пропонує гнучкість у визначенні структури даних. SQLite забезпечує швидкий доступ до даних та їх ефективне зберігання, що є важливим для додатків, які працюють з великою кількістю логів та налаштувань.

Рисунок 2.1 демонструє приклад даних, що містяться в Базі даних:

	time	level	message
1	2024-01-06 05:16:21,659	INFO	2024-01-06 05:16:21,659 - INFO - Proxy: Starting to create MSI installer.
2	2024-01-06 05:16:21,659	INFO	2024-01-06 05:16:21,659 - INFO - Proxy: Starting to create MSI installer.
3	2024-01-06 05:16:27,151	INFO	2024-01-06 05:16:27,151 - INFO - Proxy: MSI installer created successfully.
4	2024-01-06 05:16:27,151	INFO	2024-01-06 05:16:27,151 - INFO - Proxy: MSI installer created successfully.

Рисунок 2.1- Приклад даних, що містяться в колекції
логів

2.2. Архітектура системи

2.2.1. Специфікація системи

Цей додаток для створення інсталяторів програмного забезпечення складається з кількох основних компонентів, які взаємодіють між собою:

Графічний інтерфейс користувача (GUI): Використовуючи бібліотеку Tkinter, додаток надає графічний інтерфейс для взаємодії з користувачем. Це дозволяє користувачам вибирати файли, директорії та вказувати налаштування для створення інсталяторів.

Створення інсталяторів: Додаток підтримує створення двох типів інсталяторів: EXE (за допомогою Inno Setup) та MSI (за допомогою WiX Toolset). Для кожного типу інсталятора реалізовані відповідні класи, які керують процесом їх генерації.

Логування: Додаток використовує систему логування для запису подій та помилок. Логи зберігаються у файлі та в базі даних SQLite, що дозволяє ефективно відстежувати діяльність додатку.

Загальний вигляд архітектури системи наведено на рис 2.2

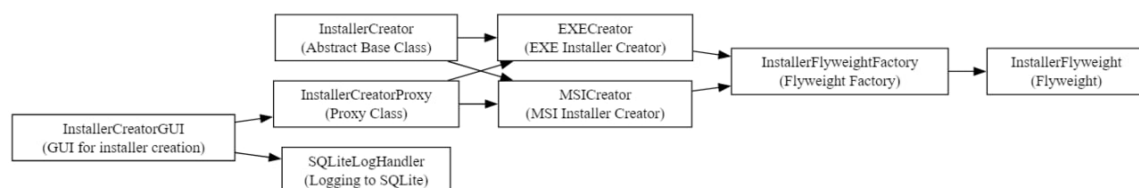


Рисунок 2.2 - Загальна архітектура системи

2.2.2. Вибір та обґрунтування патернів реалізації

Для реалізації застосунку Installer Generator було обрано такі патерни реалізації, як “Iterator”, “Abstract Factory”, “Proxy”, “Flyweight”, “Command” та “Strategy”.

- Патерн Ітератор (Iterator)

Паттерн Iterator у даному коді використовується через клас FileListIterator, який дозволяє послідовно перебирати файли у списку. Метод `__next__` повертає наступний файл із списку при кожному виклику, а при досягненні кінця списку видає виняток `StopIteration`. Цей паттерн забезпечує простий і ефективний спосіб ітерації по елементах колекції без необхідності знати її внутрішню структуру.:

```
class FileListIterator:
    """
    An iterator class to iterate over a list of files.

    Attributes:
        _file_list (List[Any]): The list of files to iterate over.
        _index (int): The current index in the file list.
    """
    def __init__(self, file_list: List[Any]):
        """
        Initializes the FileListIterator with a list of files.

        Args:
            file_list (List[Any]): The list of files to iterate over.
        """
        self._file_list: List[Any] = file_list
        self._index: int = 0

    def __next__(self) -> Any:
        """
        Returns the next file in the list.

        Raises:
            StopIteration: If the end of the file list is reached.
        """
        if self._index < len(self._file_list):
            result: Any = self._file_list[self._index]
            self._index += 1
            return result
        else:
            raise StopIteration
```

Рисунок 2.3 – Код класу FileListIterator

- Патерн “Abstract Factory”

Abstract Factory використовується через абстрактний клас `InstallerCreator`, який визначає інтерфейс для створення установників, і його конкретні реалізації `EXECreator` та `MSICreator`, які створюють різні типи установників (EXE та MSI відповідно). Цей паттерн дозволяє легко додавати нові типи установників, не змінюючи існуючий код, що використовує `InstallerCreator`

```
class InstallerCreator(ABC):
    """
    An abstract base class for creating installers.

    Attributes:
        source_directory (str): Source directory of files to include in the installer.
        output_directory (str): Output directory for the generated installer.
        file_list (list): List of files to include in the installer.
        installer_name (str): Name of the installer to be created.
    """

    def __init__(self, source_directory, output_directory, file_list, installer_name):
        """
        Initializes the InstallerCreator with necessary information.

        Args:
            source_directory (str): Source directory of files.
            output_directory (str): Output directory for the installer.
            file_list (list): List of files to include.
            installer_name (str): Name of the installer.
        """
        self.source_directory = source_directory
        self.output_directory = output_directory
        self.file_list = file_list
        self.installer_name = installer_name

    @abstractmethod
    def create_installer(self):
        """
        Abstract method to create an installer.

        This method must be implemented by subclasses.
        """
        pass
```

Рисунок 2.4 – Код інтерфейсу `InstallerCreator`

- Патерн Proxy

Proxy реалізований через клас `InstallerCreatorProxy`. Цей клас діє як проксі для об'єктів `InstallerCreator`, таких як `EXECreator` та `MSICreator`. `InstallerCreatorProxy` перехоплює виклики до цих об'єктів, дозволяючи додати додаткову функціональність, таку як логування, перш ніж передати виклик до справжнього об'єкта `InstallerCreator`. Це дозволяє логувати інформацію про процес створення установника, не змінюючи основної логіки створення установника.

```
class InstallerCreatorProxy(InstallerCreator):
    """
    A proxy class for creating installers, allowing for additional functionality.

    This class acts as a proxy for creating installers, enabling the logging of creation events and handling errors.
    """

    def __init__(self, real_creator: InstallerCreator, installer_type: str):
        """
        Initialize the InstallerCreatorProxy.

        Args:
            real_creator (InstallerCreator): The real installer creator to delegate the creation to.
            installer_type (str): The type of installer being created (e.g., 'MSI' or 'EXE').
        """
        self._real_creator = real_creator
        self._installer_type = installer_type

    def create_installer(self) -> Any:
        """
        Create the installer and log creation events.

        This method delegates the creation of the installer to the real creator while logging events.

        Returns:
            Any: The result of the installer creation process.

        Raises:
            Exception: If an error occurs during the installer creation process.
        """
        logging.info(f"Proxy: Starting to create {self._installer_type} installer.")

        try:
            result = self._real_creator.create_installer()
            logging.info(f"Proxy: {self._installer_type} installer created successfully.")
            return result
        except Exception as e:
            logging.error(f"Proxy: Error occurred while creating {self._installer_type} installer - {e}")
            raise
```

Рисунок 2.5 – Код класу `InstallerCreatorProxy`

- Патерн Flyweight

Паттерн Flyweight реалізований через класи `InstallerFlyweightFactory` та `InstallerFlyweight`. Цей паттерн використовується для ефективного управління пам'яттю при створенні багатьох схожих об'єктів. У даному випадку, `InstallerFlyweightFactory` діє як фабрика, яка створює та керує об'єктами `InstallerFlyweight`. Коли потрібно скопіювати скрипт установника, замість створення нового об'єкта для кожного випадку, використовується вже існуючий об'єкт `InstallerFlyweight`, якщо він доступний. Це зменшує витрати пам'яті та покращує продуктивність, особливо коли потрібно створити багато схожих об'єктів.

```
class InstallerFlyweight:
    """
    Represents a flyweight for compiling installer scripts.

    This class represents an installer flyweight used to compile installer scripts efficiently.
    Installer flyweights are shared objects that optimize memory usage when compiling similar scripts.
    """
    def __init__(self, script: str):
        """
        Initialize the InstallerFlyweight.

        Args:
            script (str): The script type identifier.
        """
        self._script: str = script

    6 usages (3 dynamic)
    def compile_script(self, compile_command: list) -> None:
        """
        Compile an installer script using the specified compile command.

        This method compiles an installer script using a given compile command and handles the output.

        Args:
            compile_command (list): The command used to compile the script.

        Raises:
            RuntimeError: If the compilation fails with a non-zero return code.
        """
        compile_result = subprocess.run(compile_command, capture_output=True, text=True)
        if compile_result.returncode != 0:
            print(f"Error output: {compile_result.stderr}")
            return
        print(f"Output: {compile_result.stdout}")
```

Рисунок 2.6 – Код класу `InstallerFlyweight`

```

class InstallerFlyweightFactory:
    """
    Factory for managing and retrieving installer flyweights.

    This class acts as a flyweight factory responsible for creating and managing installer flyweights.
    Installer flyweights are shared objects used to optimize memory usage when creating similar installers.
    """
    _flyweights: Dict[str, InstallerFlyweight] = {}

    2 usages
    @classmethod
    def get_flyweight(cls, key: str) -> InstallerFlyweight:
        """
        Retrieve an installer flyweight for the given key.

        If a flyweight with the specified key exists, it is returned; otherwise, a new one is created.

        Args:
            key (str): The key used to identify the installer flyweight.

        Returns:
            InstallerFlyweight: An installer flyweight instance.
        """
        if not cls._flyweights.get(key):
            cls._flyweights[key] = InstallerFlyweight(key)
        return cls._flyweights[key]

```

Рисунок 2.7 – Класи InstallerFlyweightFactory

- Патерн Command

Паттерн Command використовується в методах `compile_script` класів `EXECreator` та `MSICreator`. Паттерн Command дозволяє інкапсулювати всю інформацію, необхідну для виконання дії або запуску події, в окремий об'єкт. У даному випадку, команда для компіляції скрипта установника (наприклад, виклик компілятора Inno Setup або WiX для створення EXE або MSI установника відповідно) інкапсулюється у вигляді масиву `compile_command`.

Цей масив містить всі необхідні параметри та шляхи до виконуваних файлів, які потрібні для запуску процесу компіляції. Потім цей масив передається як аргумент у метод `subprocess.run`, який виконує команду. Такий підхід дозволяє легко змінювати та розширювати логіку виконання команд, не змінюючи основний код, який використовує ці команди.


```
def compile_exe_script(self, script_path: str) -> None:
    """
    Compiles the EXE script using Inno Setup Compiler.

    Args:
    | script_path (str): Path to the script file.
    """

    compile_command = [inno_setup_compiler, script_path]
    compile_result = subprocess.run(compile_command, capture_output=True, text=True)
    if compile_result.returncode != 0:
        print(f"Inno Setup error output: {compile_result.stderr}")
        return
    print(f"Inno Setup output: {compile_result.stdout}")
    print("EXE installer created successfully in the output directory.")
```

Рисунок 2.8 – код метода compile_exe_script

```
def compile_msi_script(self, msi_script_path: str) -> None:
    """
    Compiles the MSI script using WiX Toolset's candle and light applications.

    This method first uses 'candle' to compile the WiX script into an object file and
    then uses 'light' to link and bind the object file into an MSI package.

    Args:
    | msi_script_path (str): Path to the WiX script file.
    """

    candle_exe_path = r'C:\Program Files (x86)\WiX Toolset v3.14\bin\candle.exe'
    light_exe_path = r'C:\Program Files (x86)\WiX Toolset v3.14\bin\light.exe'

    wixobj_file_path = os.path.join(self.output_directory, 'installer.wixobj')

    candle_command = [candle_exe_path, msi_script_path, '-o', wixobj_file_path]
    candle_result = subprocess.run(candle_command, capture_output=True, text=True)
    if candle_result.returncode != 0:
        print(f"Candle error output: {candle_result.stderr}")
        return
    print(f"Candle output: {candle_result.stdout}")

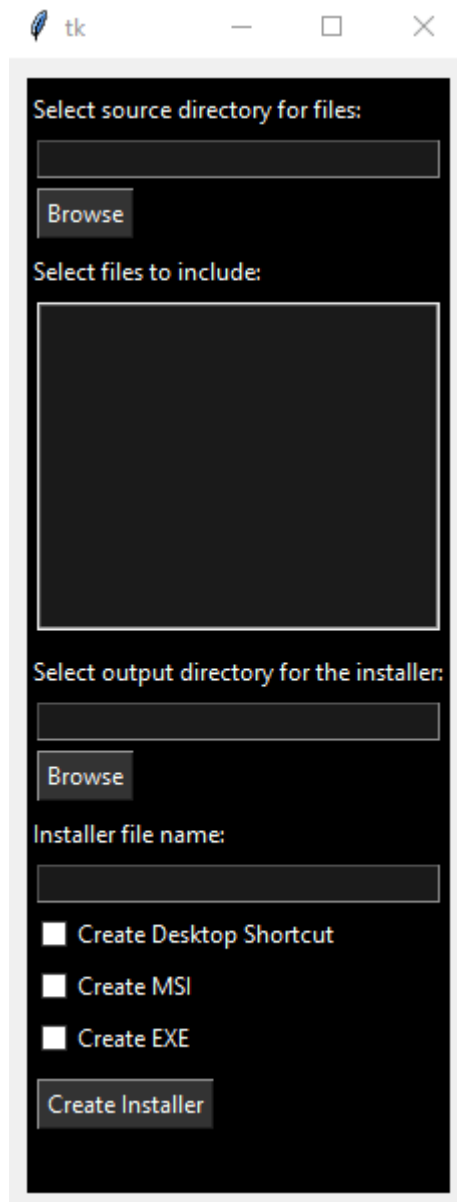
    if not os.path.exists(wixobj_file_path):
        print(f".wixobj file not created. Compilation may have failed.")
        return

    light_command = [light_exe_path, wixobj_file_path, '-o', os.path.join(self.output_directory, self.installer_name + '.msi')]
    light_result = subprocess.run(light_command, capture_output=True, text=True)
    if light_result.returncode != 0:
        print(f"Light error output: {light_result.stderr}")
        return
    print(f"Light output: {light_result.stdout}")
    print("MSI installer created successfully in the output directory.")
```

Рисунок 2.9 – код метода compile_msi_script

2.3. Інструкція користувача

При відкритті застосунку користувач побачить наступне інформаційне вікно:

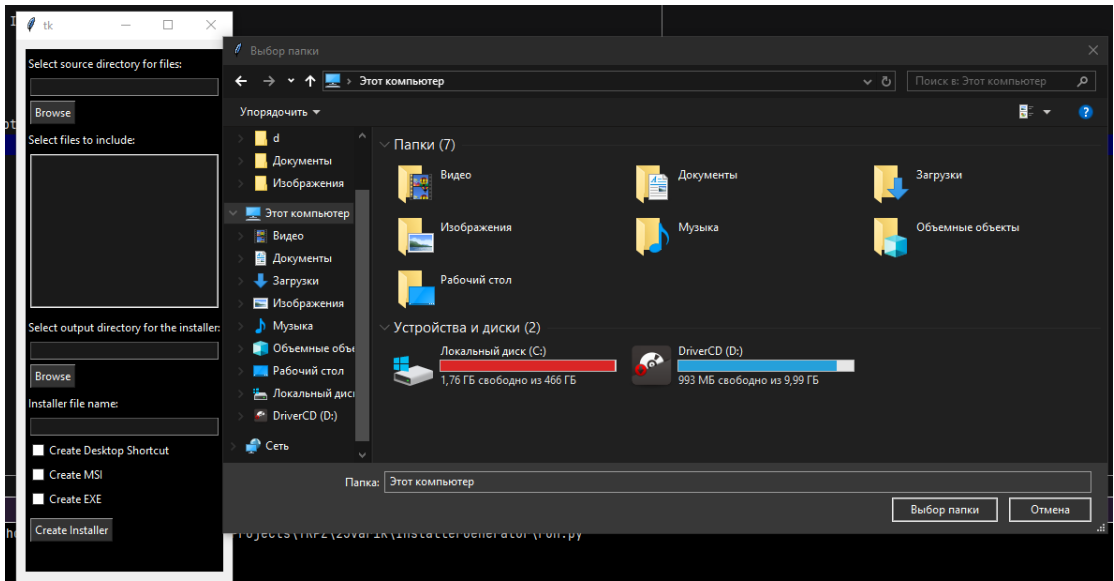


The screenshot shows a Tk window with a dark background. The window title bar includes a feather icon, the text 'tk', and standard window controls (minimize, maximize, close). The main content area is organized as follows:

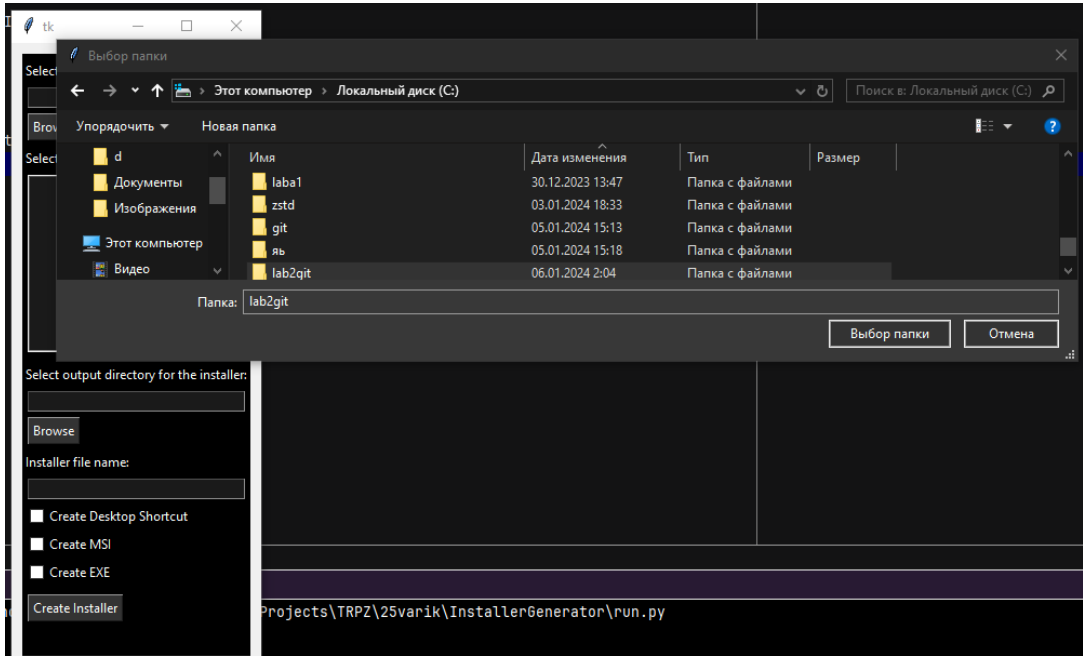
- Select source directory for files:** A text label above a single-line text entry field.
- Browse**: A button located below the source directory field.
- Select files to include:** A text label above a large, empty rectangular area for file selection.
- Select output directory for the installer:** A text label above another single-line text entry field.
- Browse**: A button located below the output directory field.
- Installer file name:** A text label above a single-line text entry field.
- Options:** Three checkboxes are listed vertically:
 - ☐ Create Desktop Shortcut
 - ☐ Create MSI
 - ☐ Create EXE
- Create Installer**: A button located at the bottom of the window.

Рисунок 2.10 – Вікно входу

При натисканні кнопки Browse, відкривається вікно вибору директорії з файлами:



Далі користувачу необхідно обрати директорію з необхідними файлами для інсталяції:



Далі, користувач бачитиме всі файли та директорії, які знаходяться у обраній директорії та матиме можливість обрати потрібні йому файли:

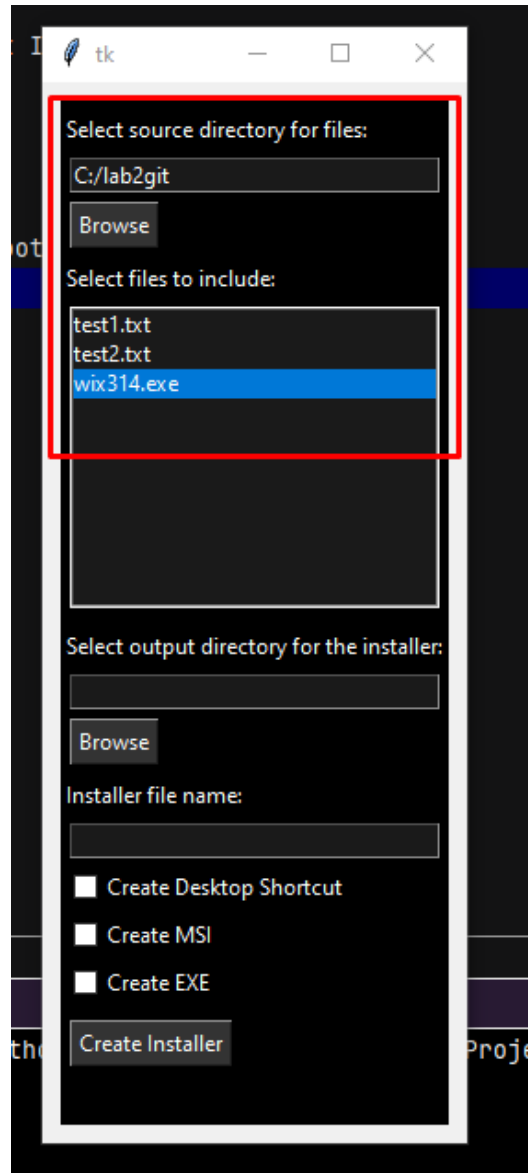


Рисунок 2.13 – Вибір необхідних файлів

Після вибору файлів, користувач повинен обрати шлях до директорії, де буде збережн інсталятор, вписати назву файла інсталятора, обрати тип інсталятора та вирішити чи потрібно створювати ярлик на робочому столі.

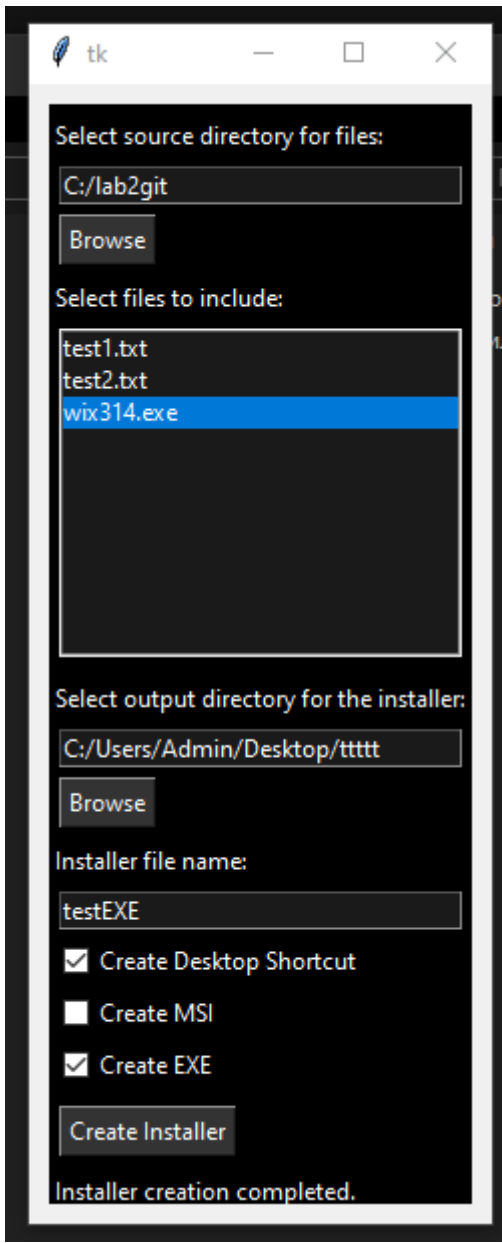


Рисунок 2.14 – Вікно вибору параметрів

Коли користувач натисне Create Installer, генератор створить файл істалятора у обраній директорії:

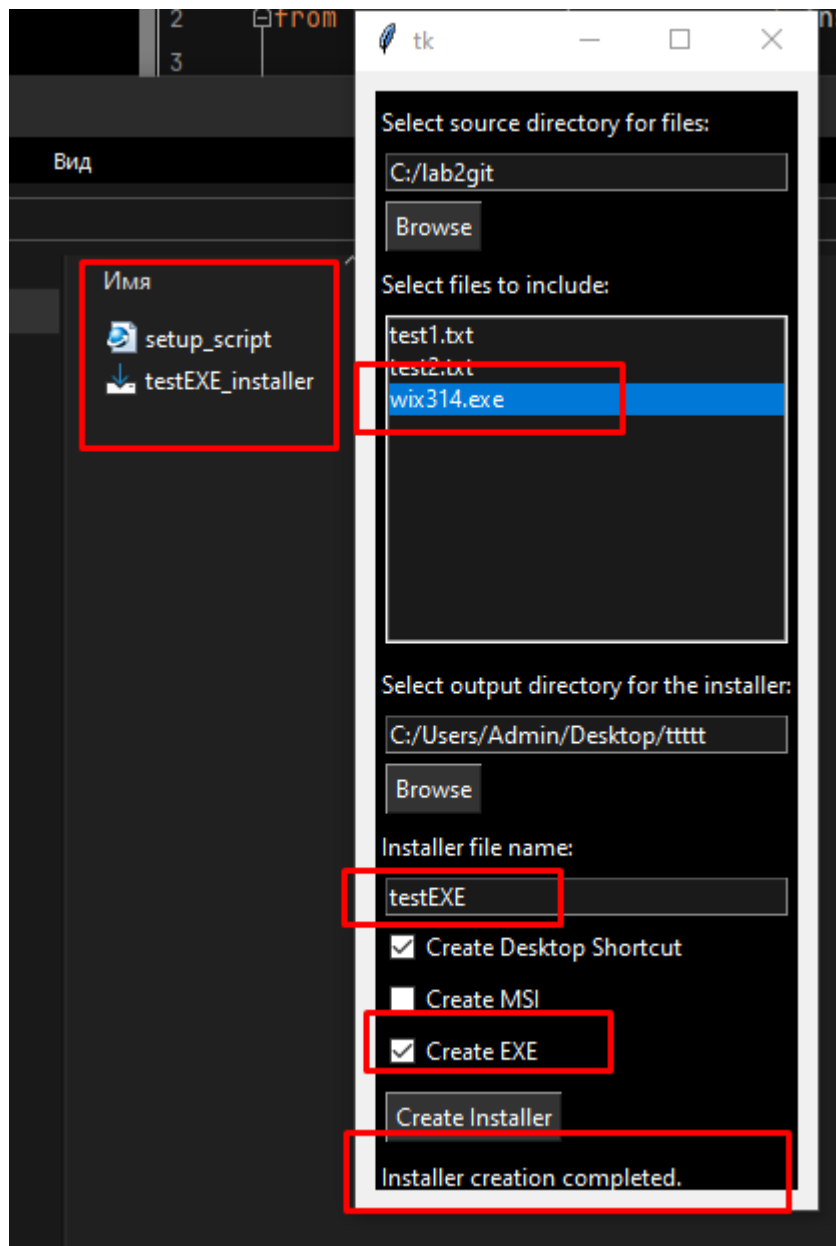


Рисунок 2.15 – Результат роботи застосунку

При відкритті файлу істалятора, користувач побачить стандартні вікна інсталятора Windows.

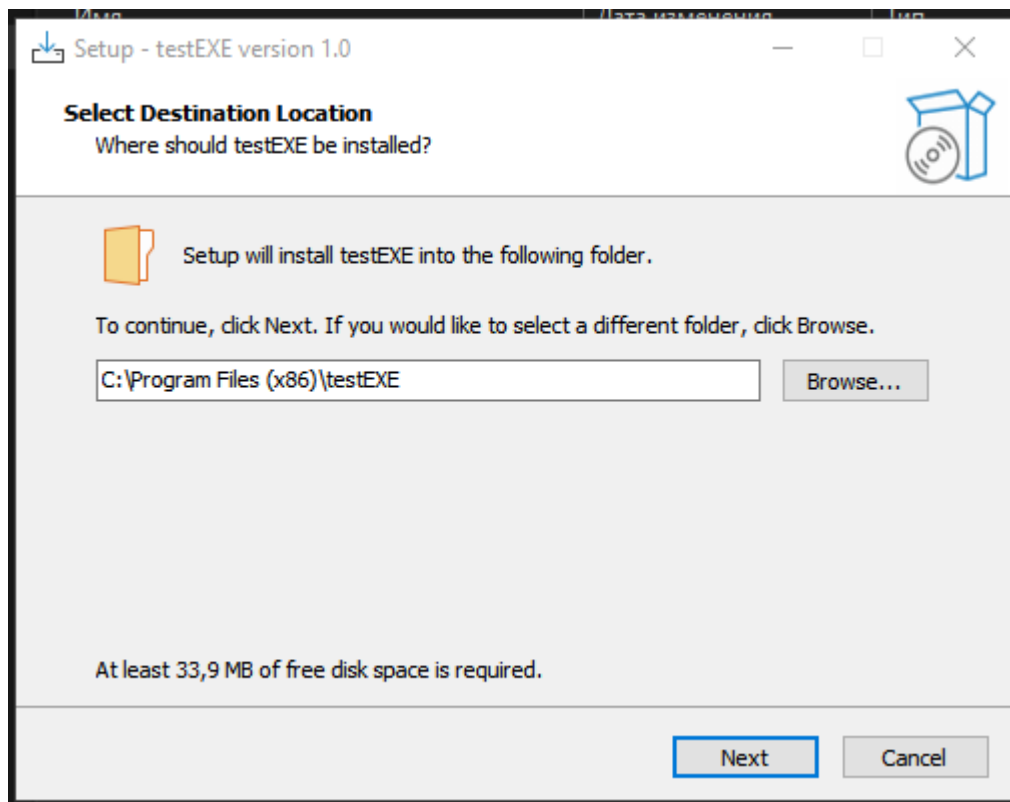


Рисунок 2.16 – Вікно входу

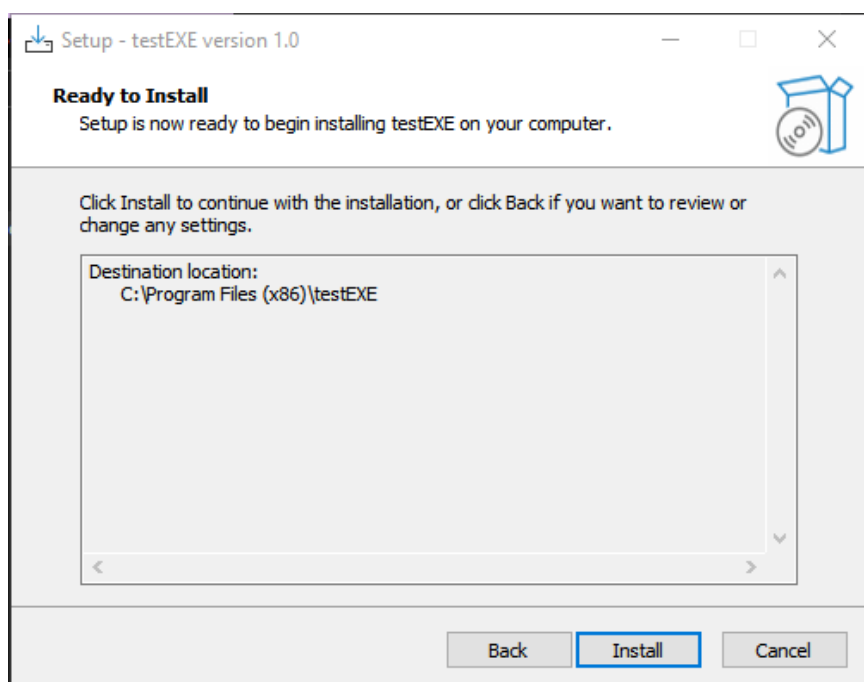


Рисунок 2.17 – Вікно ліцензійної угоди

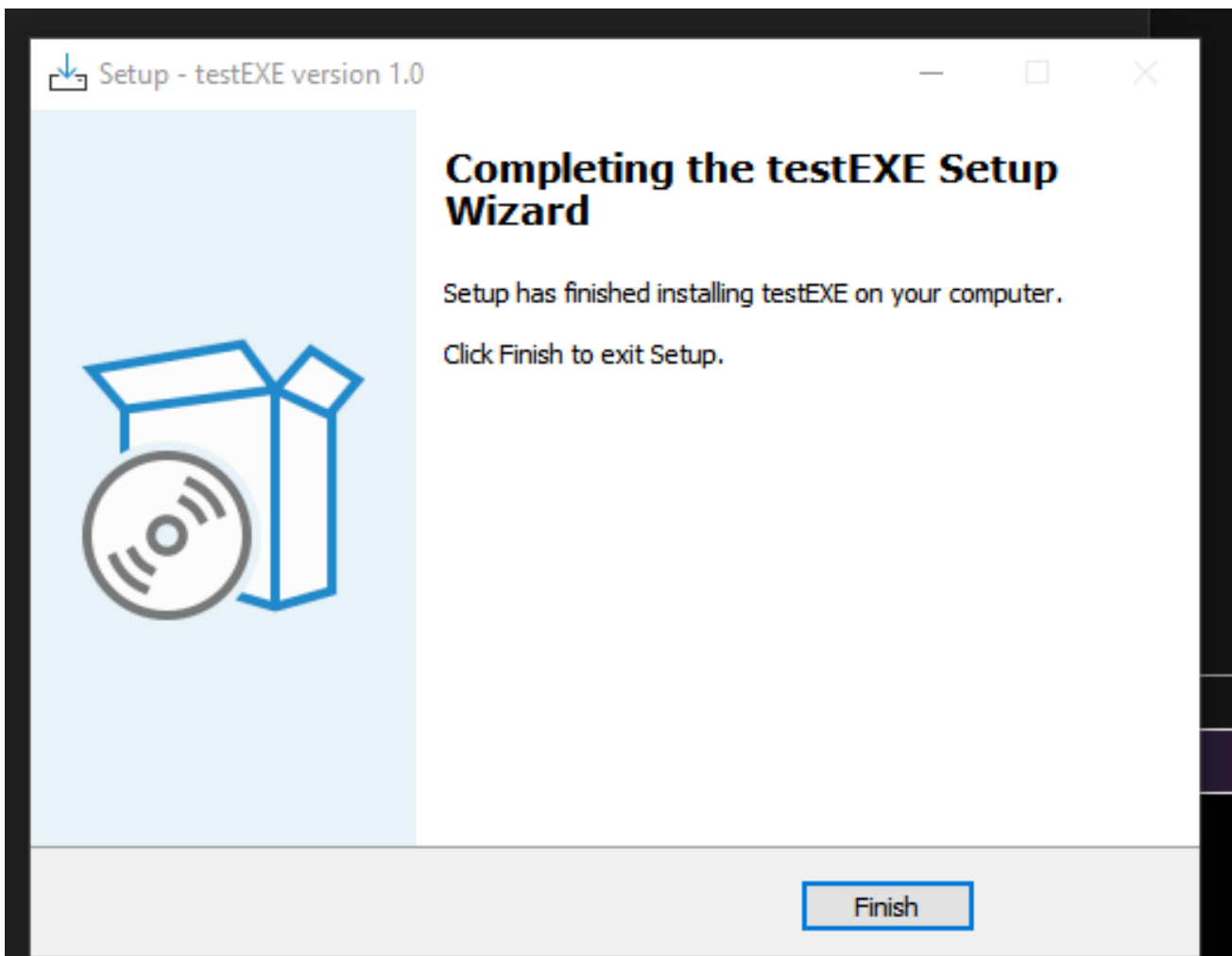


Рисунок 2.18 – Завершення істаляції

У цій курсовій роботі було досліджено тему "Генератор інсталяційних пакетів" як інструмент, що дозволяє користувачам налаштовувати інсталяційні процеси. Основна увага приділялася можливості налаштування файлів, що входять до складу інсталяційного пакету, та інтерактивним елементам установки, таким як створення ярликів на робочому столі та введення ліцензійних ключів.

Генератор забезпечує створення одного виконуваного файлу (.exe або .msi), що спрощує процес розгортання програмного забезпечення. Важливою особливістю є можливість детального налаштування компонентів інсталяції, що дозволяє користувачам адаптувати процес установки під свої конкретні потреби.

У ході дослідження було встановлено, що такий підхід значно підвищує гнучкість інсталяційних процесів та забезпечує кращий користувацький досвід. Результати розробки підтверджують, що генератор інсталяційних пакетів є ефективним рішенням для створення налаштованих інсталяцій, що відповідають сучасним вимогам до програмного забезпечення.

Цей інструмент може бути корисним для розробників програмного забезпечення, які прагнуть надати кінцевим користувачам гнучкі та зручні опції установки, забезпечуючи при цьому високий рівень налаштування і контролю над процесом інсталяції.

1. "WiX Toolset Documentation," WiX Toolset, [Електронний ресурс].

Режим доступу до ресурсу: <https://wixtoolset.org/docs/intro/>

2. "Python Getting Started Guide," Python.org, [Електронний ресурс].

Режим доступу до ресурсу: <https://www.python.org/about/gettingstarted/>

3. "Рефакторинг і Патерни проектування", [Електронний ресурс].

Режим доступу до ресурсу: <https://refactoring.guru/uk>

4. "A Comprehensive Guide to Python Design Patterns," Full Stack Python, [Електронний ресурс].

Режим доступу до ресурсу: <https://www.fullstackpython.com/design-patterns.html>

5. "Inno Setup Documentation" [Електронний ресурс].

Режим доступу до ресурсу: <https://jrsoftware.org/ishelp.php>

6. "SQLite Python Documentation" [Електронний ресурс].

Режим доступу до ресурсу: <https://docs.python.org/3/library/sqlite3.html>

7. "Abstract Factory in Python" [Електронний ресурс].

Режим доступу до ресурсу: <https://refactoring.guru/design-patterns/abstract-factory/python/example>

8. "Proxy in Python" [Електронний ресурс].

Режим доступу до ресурсу: <https://refactoring.guru/design-patterns/proxy/python/example#:~:text=Proxy%20is%20a%20structural%20design,request%20to%20a%20service%20object>

9. "Iterator in Python" [Електронний ресурс].

Режим доступу до ресурсу: <https://refactoring.guru/design-patterns/iterator/python/example>

10. "Subprocess management" [Электронный ресурс].

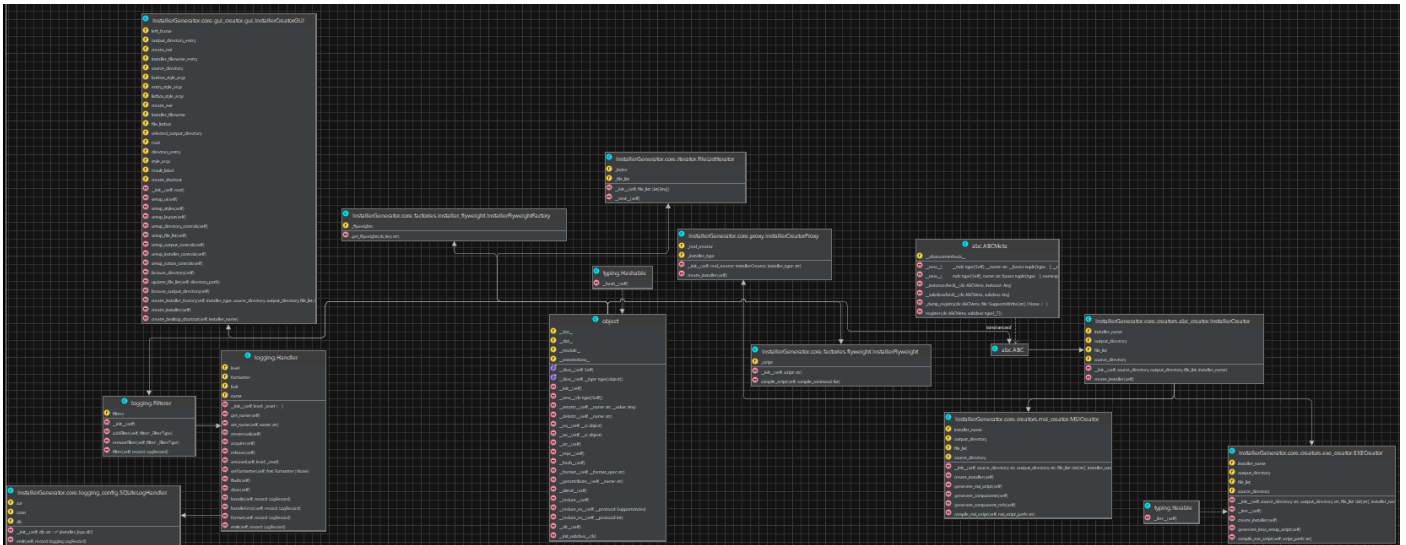
Режим доступа до ресурсу:

<https://docs.python.org/3.2/library/subprocess.html>

ДОДАТКИ

Додаток А

Діаграма класів



Додаток Б

Код проекту

run.py

```
import tkinter as tk
from core.gui_creator.gui import InstallerCreatorGUI

if __name__ == '__main__':
    root = tk.Tk()
    app = InstallerCreatorGUI(root)
    root.mainloop()
```

proxy.py

```
import logging
from typing import Any
from .creators.abc_creator import InstallerCreator
from .logging_config import setup_logging

setup_logging()
logger = logging.getLogger(__name__)

class InstallerCreatorProxy(InstallerCreator):
    """
    A proxy class for creating installers, allowing for additional functionality.

    This class acts as a proxy for creating installers, enabling the logging of
    creation events and handling errors.
    """

    def __init__(self, real_creator: InstallerCreator, installer_type: str):
        """
        Initialize the InstallerCreatorProxy.

        Args:
            real_creator (InstallerCreator): The real installer creator to delegate the
            creation to.
            installer_type (str): The type of installer being created (e.g., 'MSI' or
            'EXE').
        """
        self._real_creator = real_creator
        self._installer_type = installer_type

    def create_installer(self) -> Any:
        """
        Create the installer and log creation events.

        This method delegates the creation of the installer to the real creator while
        logging events.

        Returns:
            Any: The result of the installer creation process.

        Raises:
            Exception: If an error occurs during the installer creation process.
        """
        logging.info(f"Proxy: Starting to create {self._installer_type} installer.")
```

```

    try:
        result = self._real_creator.create_installer()
        logging.info(f"Proxy: {self._installer_type} installer created
successfully.")
        return result
    except Exception as e:
        logging.error(f"Proxy: Error occurred while creating {self._installer_type}
installer - {e}")
        raise

```

logging_config.py

```

import logging
import sqlite3

class SQLiteLogHandler(logging.Handler):
    """
    A custom logging handler that stores log records in a SQLite database.

    :param db: Path to the SQLite database file (default: '.\\installer_logs.db')
    :type db: str
    """
    def __init__(self, db: str = r'.\\installer_logs.db'):
        """
        Initialize the SQLiteLogHandler.

        :param db: Path to the SQLite database file (default: '.\\installer_logs.db')
        :type db: str
        """
        super().__init__()
        self.db: str = db
        self.conn: sqlite3.Connection = sqlite3.connect(self.db)
        self.cur: sqlite3.Cursor = self.conn.cursor()
        self.cur.execute("""
            CREATE TABLE IF NOT EXISTS logs (
                time TEXT,
                level TEXT,
                message TEXT
            )
        """)
        self.conn.commit()

    def emit(self, record: logging.LogRecord) -> None:
        """
        Emit a log record by inserting it into the SQLite database.

        :param record: The log record to be emitted.
        :type record: logging.LogRecord
        """
        log_entry: str = self.format(record)
        self.cur.execute("INSERT INTO logs (time, level, message) VALUES (?, ?, ?)",
            (record.asctime, record.levelname, log_entry))
        self.conn.commit()

    def setup_logging() -> None:
        """
        Configure the logging system to use a SQLiteLogHandler and save logs to a file.

        This function sets up the logging system to store logs in a file named
        'installer_creator.log' and

```

also adds the `SQLiteLogHandler` to store logs in an `SQLite` database.

Usage:

```
'''
setup_logging()
logger = logging.getLogger(__name__)
logger.info("This is an example log message.")
'''

:return: None
"""
logging.basicConfig(filename='.\\installer_creator.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
sqlite_handler: SQLiteLogHandler = SQLiteLogHandler()
sqlite_handler.setLevel(logging.INFO)
formatter: logging.Formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
sqlite_handler.setFormatter(formatter)
logging.getLogger().addHandler(sqlite_handler)
```

iterator.py

```
from typing import List, Any

class FileListIterator:
    """
    An iterator class to iterate over a list of files.

    Attributes:
        _file_list (List[Any]): The list of files to iterate over.
        _index (int): The current index in the file list.
    """
    def __init__(self, file_list: List[Any]):
        """
        Initializes the FileListIterator with a list of files.

        Args:
            file_list (List[Any]): The list of files to iterate over.
        """
        self._file_list: List[Any] = file_list
        self._index: int = 0

    def __next__(self) -> Any:
        """
        Returns the next file in the list.

        Raises:
            StopIteration: If the end of the file list is reached.
        """
        if self._index < len(self._file_list):
            result: Any = self._file_list[self._index]
            self._index += 1
            return result
        else:
            raise StopIteration
```

base_config.py

```
inno_setup_compiler = r"C:\Program Files (x86)\Inno Setup 6\ISCC.exe"
candle_exe_path = r"C:\Program Files (x86)\WiX Toolset v3.14\bin\candle.exe'
light_exe_path = r'C:\Program Files (x86)\WiX Toolset v3.14\bin\light.exe'
```

gui.py

```

import os
import tkinter as tk
from tkinter import filedialog, ttk
import winshell
from ..creators.exe_creator import EXECreator
from ..creators.msi_creator import MSICreator
from ..proxy import InstallerCreatorProxy
class InstallerCreatorGUI:
    """
    Graphical User Interface for creating installers.

    This class provides a user-friendly interface for configuring and generating MSI
    and EXE installers.
    """
    def __init__(self, root):
        """
        Initialize the InstallerCreatorGUI instance.

        Args:
            root (tk.Tk): The root window of the GUI.
        """
        self.root = root
        self.create_msi = tk.BooleanVar()
        self.create_exe = tk.BooleanVar()
        self.create_shortcut = tk.BooleanVar()
        self.source_directory = tk.StringVar()
        self.selected_output_directory = tk.StringVar()
        self.installer_filename = tk.StringVar()
        self.setup_ui()

    def setup_ui(self):
        """
        Set up the user interface components.

        This method calls various setup methods to configure the GUI's styles, layout,
        controls, and actions.
        """
        self.setup_styles()
        self.setup_layout()
        self.setup_directory_controls()
        self.setup_file_list()
        self.setup_output_controls()
        self.setup_installer_controls()
        self.setup_action_controls()

    def setup_styles(self):
        """
        Set up styles for GUI elements.

        This method configures styles for background color, foreground color, and focus
        color.
        """
        self.style_args = {'bg': 'black', 'fg': 'white'}
        self.entry_style_args = {'bg': '#1a1a1a', 'fg': 'white', 'insertbackground':
'white'}
        self.listbox_style_args = {'bg': '#1a1a1a', 'fg': 'white'}
        self.button_style_args = {'bg': '#333333', 'fg': 'white', 'activebackground':
'#4d4d4d', 'activeforeground': 'white'}

        style = ttk.Style()
        style.configure("TCheckbutton", background='black', foreground='white',
focuscolor=style.configure(".")["background"])

    def setup_layout(self):
        """
        Set up the GUI layout.

```



```

    This method creates a left frame with a black background.
    """
    self.left_frame = tk.Frame(self.root, bg='black')
    self.left_frame.pack(side=tk.LEFT, padx=10, pady=10)

    def setup_directory_controls(self):
        """
        Set up controls for selecting the source directory.

        This method includes a label, an entry field for the directory path, and a
        browse button.
        """
        directory_label = tk.Label(self.left_frame, text="Select source directory for
files:", **self.style_args)
        directory_label.pack(anchor='w', pady=(5, 0))

        self.directory_entry = tk.Entry(self.left_frame,
textvariable=self.source_directory, **self.entry_style_args)
        self.directory_entry.pack(fill='x', padx=5, pady=5)

        browse_directory_button = tk.Button(self.left_frame, text="Browse",
command=self.browse_directory, **self.button_style_args)
        browse_directory_button.pack(anchor='w', padx=5)

    def setup_file_list(self):
        """
        Set up the file list control.

        This method creates a Listbox with multiple selection mode for selecting files
        to include in the installer.
        """
        file_list_label = tk.Label(self.left_frame, text="Select files to include:",
**self.style_args)
        file_list_label.pack(anchor='w', pady=(5, 0))

        self.file_listbox = tk.Listbox(self.left_frame, selectmode=tk.MULTIPLE,
**self.listbox_style_args)
        self.file_listbox.pack(fill='both', expand=True, padx=5, pady=5)

    def setup_output_controls(self):
        """
        Set up controls for selecting the output directory.

        This method includes a label, an entry field for the directory path, and a
        browse button.
        """
        output_directory_label = tk.Label(self.left_frame, text="Select output
directory for the installer:", **self.style_args)
        output_directory_label.pack(anchor='w', pady=(5, 0))

        self.output_directory_entry = tk.Entry(self.left_frame,
textvariable=self.selected_output_directory, **self.entry_style_args)
        self.output_directory_entry.pack(fill='x', padx=5, pady=5)

        browse_output_button = tk.Button(self.left_frame, text="Browse",
command=self.browse_output_directory, **self.button_style_args)
        browse_output_button.pack(anchor='w', padx=5)

    def setup_installer_controls(self):
        """
        Set up controls for configuring the installer.

        This method includes input fields for the installer name and checkboxes for
        various installer options.
        """

```

```

        installer_filename_label = tk.Label(self.left_frame, text="Installer file
name:", **self.style_args)
        installer_filename_label.pack(anchor='w', pady=(5, 0))

        self.installer_filename_entry = tk.Entry(self.left_frame,
textvariable=self.installer_filename, **self.entry_style_args)
        self.installer_filename_entry.pack(fill='x', padx=5, pady=5)

        shortcut_checkbox = ttk.Checkbutton(self.left_frame, text="Create Desktop
Shortcut", style="TCheckbutton", variable=self.create_shortcut)
        shortcut_checkbox.pack(anchor='w', padx=5, pady=(0, 5))

        msi_checkbox = ttk.Checkbutton(self.left_frame, text="Create MSI",
style="TCheckbutton", variable=self.create_msi)
        msi_checkbox.pack(anchor='w', padx=5, pady=(0, 5))

        exe_checkbox = ttk.Checkbutton(self.left_frame, text="Create EXE",
style="TCheckbutton", variable=self.create_exe)
        exe_checkbox.pack(anchor='w', padx=5, pady=(0, 5))

    def setup_action_controls(self):
        """
        Set up controls for actions like creating the installer.

        This method includes a button to create the installer and a label for
        displaying the result.
        """
        create_installer_button = tk.Button(self.left_frame, text="Create Installer",
command=self.create_installer, **self.button_style_args)
        create_installer_button.pack(anchor='w', padx=5, pady=5)

        self.result_label = tk.Label(self.left_frame, text="", **self.style_args)
        self.result_label.pack(anchor='w', pady=(5, 0))

    def browse_directory(self):
        """
        Open a directory dialog to select the source directory.

        Updates the source directory entry field and updates the file list.
        """
        directory_path = filedialog.askdirectory()
        if directory_path:
            self.source_directory.set(directory_path)
            self.directory_entry.delete(0, tk.END)
            self.directory_entry.insert(0, directory_path)
            self.update_file_list(directory_path)

    def update_file_list(self, directory_path):
        """
        Update the file list based on the selected source directory.

        Args:
            directory_path (str): The path of the selected source directory.
        """
        self.file_listbox.delete(0, tk.END)
        for filename in os.listdir(directory_path):
            self.file_listbox.insert(tk.END, filename)

    def browse_output_directory(self):
        """
        Open a directory dialog to select the output directory for the installer.

        Updates the output directory entry field.
        """
        directory_path = filedialog.askdirectory()
        if directory_path:
            self.selected_output_directory.set(directory_path)

```

```

        self.output_directory_entry.delete(0, tk.END)
        self.output_directory_entry.insert(0, directory_path)

    def create_installer_factory(self, installer_type, source_directory,
                                output_directory, file_list, installer_name):
        """
        Create an installer factory for MSI or EXE.

        Args:
            installer_type (str): The type of installer to create (MSI or EXE).
            source_directory (str): The source directory of files.
            output_directory (str): The output directory for the installer.
            file_list (list): List of files to include in the installer.
            installer_name (str): Name of the installer.

        Returns:
            InstallerCreatorProxy: An instance of an installer creator factory.
        """
        if installer_type == 'MSI':
            real_creator = MSICreator(source_directory, output_directory, file_list,
                                      installer_name)
        elif installer_type == 'EXE':
            real_creator = EXECreator(source_directory, output_directory, file_list,
                                      installer_name)
        else:
            raise ValueError("Unknown installer type")

        return InstallerCreatorProxy(real_creator, installer_type)

    def create_installer(self):
        """
        Create the installer based on user inputs.

        Calls the appropriate installer creator based on selected checkboxes (MSI or
        EXE).

        Creates a desktop shortcut if the corresponding checkbox is selected.
        Displays the result in the GUI.
        """
        installer_name = self.installer_filename.get()
        source_directory = self.source_directory.get()
        output_directory = self.selected_output_directory.get()

        if not installer_name:
            self.result_label.config(text="Please enter a name for the installer.")
            return

        if not source_directory or not output_directory:
            self.result_label.config(text="Please select both source and output
            directories.")
            return

        file_list = [self.file_listbox.get(idx) for idx in
                     self.file_listbox.curselection()]

        if self.create_msi.get():
            msi_creator = self.create_installer_factory('MSI', source_directory,
                                                         output_directory, file_list, installer_name)
            msi_creator.create_installer()

        if self.create_exe.get():
            exe_creator = self.create_installer_factory('EXE', source_directory,
                                                         output_directory, file_list, installer_name)
            exe_creator.create_installer()

        if self.create_shortcut.get():
            self.create_desktop_shortcut(installer_name)

```

```

self.result_label.config(text="Installer creation completed.")

def create_desktop_shortcut(self, installer_name):
    """
    Create a desktop shortcut for the installer.

    Args:
        installer_name (str): Name of the installer.
    """
    output_directory = self.selected_output_directory.get()

    shortcut_path = os.path.join(winshell.desktop(), installer_name + '.lnk')

    target = os.path.join(output_directory, installer_name + '.exe')

    with winshell.shortcut(shortcut_path) as shortcut:
        shortcut.path = target
        shortcut.description = "Shortcut to " + installer_name
        shortcut.working_directory = output_directory

```

installer_flyweight.py

```

from .flyweight import InstallerFlyweight
from typing import Dict

class InstallerFlyweightFactory:
    """
    Factory for managing and retrieving installer flyweights.

    This class acts as a flyweight factory responsible for creating and managing
    installer flyweights.
    Installer flyweights are shared objects used to optimize memory usage when creating
    similar installers.
    """
    _flyweights: Dict[str, InstallerFlyweight] = {}

    @classmethod
    def get_flyweight(cls, key: str) -> InstallerFlyweight:
        """
        Retrieve an installer flyweight for the given key.

        If a flyweight with the specified key exists, it is returned; otherwise, a new
        one is created.

        Args:
            key (str): The key used to identify the installer flyweight.

        Returns:
            InstallerFlyweight: An installer flyweight instance.
        """
        if not cls._flyweights.get(key):
            cls._flyweights[key] = InstallerFlyweight(key)
        return cls._flyweights[key]

```

flyweight.py

```

import subprocess

class InstallerFlyweight:
    """

```

Represents a flyweight for compiling installer scripts.

This class represents an installer flyweight used to compile installer scripts efficiently.

Installer flyweights are shared objects that optimize memory usage when compiling similar scripts.

```

"""
def __init__(self, script: str):
    """
    Initialize the InstallerFlyweight.

    Args:
        script (str): The script type identifier.
    """
    self._script: str = script

def compile_script(self, compile_command: list) -> None:
    """
    Compile an installer script using the specified compile command.

    This method compiles an installer script using a given compile command and
    handles the output.

    Args:
        compile_command (list): The command used to compile the script.

    Raises:
        RuntimeError: If the compilation fails with a non-zero return code.
    """
    compile_result = subprocess.run(compile_command, capture_output=True,
text=True)
    if compile_result.returncode != 0:
        print(f"Error output: {compile_result.stderr}")
        return
    print(f"Output: {compile_result.stdout}")

```

msi_creator.py

```

import os
import subprocess
import uuid
from typing import List

from ..base_config import candle_exe_path, light_exe_path
from ..factories.installer_flyweight import InstallerFlyweightFactory
from .abc_creator import InstallerCreator

class MSICreator(InstallerCreator):
    """
    Class for creating an MSI installer.

    Attributes:
        source_directory (str): Directory where source files are located.
        output_directory (str): Directory where the MSI installer will be created.
        file_list (List[str]): List of files to be included in the installer.
        installer_name (str): Name of the installer.
    """
    def __init__(self, source_directory: str, output_directory: str, file_list:
List[str], installer_name: str):
        """
        Initialize the MSICreator.

```

```

Args:
    source_directory (str): Directory containing the source files.
    output_directory (str): Output directory for the installer.
    file_list (List[str]): List of files to include in the installer.
    installer_name (str): Name for the MSI file.
"""
self.source_directory: str = source_directory
self.output_directory: str = output_directory
self.file_list: List[str] = file_list
self.installer_name: str = installer_name

def create_installer(self) -> None:
    """
    Creates an MSI installer.

    This method handles the logic for generating an MSI installer. It checks
    for the presence of required files and the installer name, and then proceeds
    to compile the MSI installer using WiX Toolset.
    """
    if not self.file_list:
        print("No files selected. Please select files to include in the
installer.")
        return

    if not self.installer_name:
        print("Please enter a name for the MSI file.")
        return

    msi_script_path = os.path.join(self.output_directory, "installer.wxs")
    with open(msi_script_path, "w") as msi_script_file:
        msi_script_file.write(self.generate_msi_script())

    wixobj_file_path = os.path.join(self.output_directory, 'installer.wixobj')

    candle_command = [candle_exe_path, msi_script_path, '-o', wixobj_file_path]
    flyweight = InstallerFlyweightFactory.get_flyweight("MSI")
    flyweight.compile_script(candle_command)

    if not os.path.exists(wixobj_file_path):
        print(".wixobj file not created. Compilation may have failed.")
        return

    light_command = [light_exe_path, wixobj_file_path, '-o',
os.path.join(self.output_directory, self.installer_name + '.msi')]
    flyweight.compile_script(light_command)

    print("MSI installer created successfully in the output directory.")

def generate_msi_script(self) -> str:
    """
    Generates an XML script for MSI installation using WiX Toolset.

    Returns:
        str: A string containing the WiX XML script necessary to create the MSI
installer.
    """
    new_guid = str(uuid.uuid4()).upper()
    components_xml = self.generate_components()
    component_refs_xml = self.generate_component_refs()

    msi_script = f"""<?xml version="1.0" encoding="UTF-8"?>
        <Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
            <Product Id="{self.installer_name}"
Language="1033" Version="1.0.0.0" Manufacturer
            ="MyCompany" UpgradeCode="{new_guid}">
                <Package InstallerVersion="200" Compressed="yes"
InstallScope="perMachine" />
    """

```

```

        <Media Id="1" Cabinet="medial.cab" EmbedCab="yes" />
        <Directory Id="TARGETDIR" Name="SourceDir">
            <Directory Id="ProgramFilesFolder">
                <Directory Id="INSTALLFOLDER"
Name="{self.installer_name}">
                    {components_xml}
                </Directory>
            </Directory>
        </Directory>
        <Feature Id="ProductFeature"
Title="{self.installer_name}" Level="1">
            {component_refs_xml}
        </Feature>
    </Product>
</Wix>
"""
    return msi_script

def generate_components(self) -> str:
    """
    Generates XML components for each file to be included in the MSI installer.

    This method iterates over the file list and creates an XML component for each
file,
    which is then used in the WiX script.

    Returns:
        str: A string containing XML components for the installer script.
    """
    components_xml = ""
    for file in self.file_list:
        file_id = os.path.basename(file)
        source_path = os.path.join(self.source_directory, file)
        component_xml = f"""
        <Component Id="{file_id}" Guid="{str(uuid.uuid4())}">
            <File Id="{file_id}" Source="{source_path}" KeyPath="yes" />
        </Component>
        """
        components_xml += component_xml
    return components_xml

def generate_component_refs(self) -> str:
    """
    Generates XML component references for the WiX script.

    This method creates a reference for each component in the installer script,
ensuring that each file is correctly included in the MSI package.

    Returns:
        str: A string containing XML component references.
    """
    component_refs_xml = ""
    for file in self.file_list:
        file_id = os.path.basename(file)
        component_ref_xml = f"<ComponentRef Id='{file_id}' />"
        component_refs_xml += component_ref_xml
    return component_refs_xml

def compile_msi_script(self, msi_script_path: str) -> None:
    """
    Compiles the MSI script using WiX Toolset's candle and light applications.

    This method first uses 'candle' to compile the WiX script into an object file
and
    then uses 'light' to link and bind the object file into an MSI package.

    Args:

```

```

        msi_script_path (str): Path to the WiX script file.
    """
    candle_exe_path = r'C:\Program Files (x86)\WiX Toolset v3.14\bin\candle.exe'
    light_exe_path = r'C:\Program Files (x86)\WiX Toolset v3.14\bin\light.exe'

    wixobj_file_path = os.path.join(self.output_directory, 'installer.wixobj')

    candle_command = [candle_exe_path, msi_script_path, '-o', wixobj_file_path]
    candle_result = subprocess.run(candle_command, capture_output=True, text=True)
    if candle_result.returncode != 0:
        print(f"Candle error output: {candle_result.stderr}")
        return
    print(f"Candle output: {candle_result.stdout}")

    if not os.path.exists(wixobj_file_path):
        print(".wixobj file not created. Compilation may have failed.")
        return

    light_command = [light_exe_path, wixobj_file_path, '-o',
os.path.join(self.output_directory, self.installer_name + '.msi')]
    light_result = subprocess.run(light_command, capture_output=True, text=True)
    if light_result.returncode != 0:
        print(f"Light error output: {light_result.stderr}")
        return
    print(f"Light output: {light_result.stdout}")
    print("MSI installer created successfully in the output directory.")

```

exe_creator.py

```

import os
import subprocess
from typing import List, Iterator

from .abc_creator import InstallerCreator
from ..base_config import inno_setup_compiler
from ..iterator import FileListIterator
from ..factories.installer_flyweight import InstallerFlyweightFactory

class EXECreator(InstallerCreator):
    """
    Class for creating an EXE installer.

    Attributes:
        source_directory (str): Directory where source files are located.
        output_directory (str): Directory where the EXE installer will be created.
        file_list (List[str]): List of files to be included in the installer.
        installer_name (str): Name of the installer.
    """

    def __init__(self, source_directory: str, output_directory: str, file_list:
List[str], installer_name: str):
        """
        Initialize the EXECreator.

        Args:
            source_directory (str): Directory containing the source files.
            output_directory (str): Output directory for the installer.
            file_list (List[str]): List of files to include in the installer.
            installer_name (str): Name for the EXE file.
        """
        self.source_directory: str = source_directory
        self.output_directory: str = output_directory

```



```

self.file_list: List[str] = file_list
self.installer_name: str = installer_name

def __iter__(self) -> Iterator[str]:
    """
    Returns an iterator for the file list.

    Returns:
        FileListIterator: An iterator for the file list.
    """
    return FileListIterator(self.file_list)

def create_installer(self) -> None:
    """
    Creates an EXE installer.

    This method handles the logic for generating an EXE installer.
    """
    if not self.file_list:
        print("No files selected. Please select files to include in the
installer.")
        return

    if not self.installer_name:
        print("Please enter a name for the EXE file.")
        return

    script_path = os.path.join(self.output_directory, "setup_script.iss")
    with open(script_path, "w") as script_file:
        script_file.write(self.generate_inno_setup_script())

    inno_setup_compiler = r"C:\Program Files (x86)\Inno Setup 6\ISCC.exe"
    compile_command = [inno_setup_compiler, script_path]

    flyweight = InstallerFlyweightFactory.get_flyweight("EXE")
    flyweight.compile_script(compile_command)

    print("EXE installer created successfully in the output directory.")

def generate_inno_setup_script(self) -> str:
    """
    Generates an Inno Setup script for the installer.

    Returns:
        str: A string containing the Inno Setup script.
    """
    script = f"""
        [Setup]
        AppName={self.installer_name}
        AppVersion=1.0
        DefaultDirName={{autopf}}\\{self.installer_name}
        OutputDir={self.output_directory}
        OutputBaseFilename={self.installer_name}_installer
        Compression=lzma
        SolidCompression=yes
        [Files]
        """

    for file in self:
        script += f"Source: \"{os.path.join(self.source_directory, file)}\"";
DestDir: \"{self.app}\\\"\\n"
    return script

def compile_exe_script(self, script_path: str) -> None:
    """
    Compiles the EXE script using Inno Setup Compiler.

```

```

    Args:
        script_path (str): Path to the script file.
    """

    compile_command = [inno_setup_compiler, script_path]
    compile_result = subprocess.run(compile_command, capture_output=True,
text=True)
    if compile_result.returncode != 0:
        print(f"Inno Setup error output: {compile_result.stderr}")
        return
    print(f"Inno Setup output: {compile_result.stdout}")
    print("EXE installer created successfully in the output directory.")

```

abc_creator.py

```

from abc import ABC, abstractmethod

class InstallerCreator(ABC):
    """
    An abstract base class for creating installers.

    Attributes:
        source_directory (str): Source directory of files to include in the installer.
        output_directory (str): Output directory for the generated installer.
        file_list (list): List of files to include in the installer.
        installer_name (str): Name of the installer to be created.
    """

    def __init__(self, source_directory, output_directory, file_list, installer_name):
        """
        Initializes the InstallerCreator with necessary information.

        Args:
            source_directory (str): Source directory of files.
            output_directory (str): Output directory for the installer.
            file_list (list): List of files to include.
            installer_name (str): Name of the installer.
        """
        self.source_directory = source_directory
        self.output_directory = output_directory
        self.file_list = file_list
        self.installer_name = installer_name

    @abstractmethod
    def create_installer(self):
        """
        Abstract method to create an installer.

        This method must be implemented by subclasses.
        """
        pass

```