ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І. І. МЕЧНИКОВА
(повне найменування закладу вищої освіти)

Факультет математики, фізики та інформаційних технологій
(повне найменування факультету)

Кафедра інформаційних технологій
(повна назва кафедри)

Кваліфікаційна робота

на здобуття ступеня вищої освіти «Бакалавр»

«Розробка імітаційної моделі сценаріїв взаємодії користувача у діалогових процесах»

(тема кваліфікаційної роботи українською мовою)

«Development of a simulation model of user interaction scenarios in dialog processes»

(тема кваліфікаційної роботи англійською мовою)

Виконав: здобувач денної форми навчання
спеціальності 122 Комп'ютерні науки
(код, назва спеціальності)
Освітня програма Комп'ютерні науки
(назва)
ШУЛЬГА Данило Олегович
(прізвище, ім'я, по-батькові здобувача)

Керівник старш. викл. Штефан Н. З. _____
(науковий ступінь, вчене звання, прізвище, ініціали) (підпис)

Рецензент к.т.н., доцент кафедри інженерії ПЗ національного університету "Одеська політехніка", Тройніна А.С.
(науковий ступінь, вчене звання, прізвище, ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
Інформаційних технологій
№ ___ від ___ _____ 2025 р.

Завідувачка кафедри

_____ КАЗАКОВА Надія
(підпис) (прізвище,ім'я)

Захищено на засіданні ЕК № _____
протокол № ___ від _____ 2025 р.

Оцінка _____ / _____ / _____
(за національною шкалою/шкалою ECTS/ бали)

Голова ЕК

_____ КОПИЧЕНКО Іван
(підпис) (прізвище,ім'я)

Одеса 2025

АНОТАЦІЯ

Метою роботи є забезпечення користувачів надійним та гнучким програмним застосунком для імітаційної моделі сценаріїв взаємодії користувача у діалогових процесах.

Розроблений інструмент поєднує механізм запису й відтворення макросів із підтримкою точного фіксування натискань клавіш, рухів і кліків миші, глобальне налаштування гарячих клавіш у будь-якій частині системи та режими автокліку й мультикліку з контролем кількості повторів і інтервалів. Інтерфейс на базі Tkinter забезпечує інтуїтивну взаємодію, ядро Recorder реалізує багатопотокове записування і відтворення подій, та інший функціонал, а модулі на основі pynput, keyboard та ctypes/WinAPI гарантують точну емуляцію введення під Windows і Linux. Файлове сховище у форматі JSON відповідає за збереження макросів, налаштувань мультикліку та обраних фонів.

Результатом роботи є потужний автоклікер, який поєднує в собі весь спектр можливостей імітації дій користувача – від простих кліків і клавіатурних подій до складних послідовностей та циклічного сценарію, забезпечуючи значне підвищення продуктивності та зручності у виконанні повторюваних завдань.

Ключові слова: автоклікер, макроси, гарячі клавіші, автоклік, мультиклік, Python, Tkinter, pynput, ctypes, JSON.

ABSTRUCT

The goal of the work is to provide users with a reliable and flexible software application for simulating user interaction scenarios in dialog processes.

The developed tool combines a mechanism for recording and playing back macros with precise tracking of keystrokes, mouse movements and clicks, global hotkey configuration throughout the system, as well as auto-click and multi-click modes with control over repetition count and intervals. The Tkinter-based interface ensures intuitive interaction, the Recorder core implements multithreaded event recording and playback along with other functionalities, and modules built with pynput, keyboard, and ctypes/WinAPI guarantee accurate input emulation under both Windows and Linux. A JSON-based file storage system is responsible for saving macros, multi-click settings, and selected interface backgrounds.

The result of the work is a powerful auto-clicker that integrates the full range of user input simulation capabilities – from simple clicks and keyboard actions to complex sequences and cyclic scenarios – significantly increasing productivity and ease when performing repetitive tasks.

Keywords: auto-clicker, macros, hotkeys, auto-click, multi-click, Python, Tkinter, pynput, ctypes, JSON.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – інтерфейс програмування додатків

DLL – Dynamic-Link Library (динамічно підключувана бібліотека)

GUI – графічний інтерфейс користувача

HKL – ідентифікатор розкладки клавіатури (Handle to Keyboard Layout)

INPUT – структура WinAPI для емуляції подій введення

JSON – JavaScript Object Notation (формат серіалізації даних)

KLID – Keyboard Layout ID (ідентифікатор розкладки клавіатури у Windows)

.NET – платформа розробки від Microsoft

PIL – Python Imaging Library (бібліотека для роботи із зображеннями)

RGBA – колірний простір з каналом прозорості (Red Green Blue Alpha)

UAC – User Account Control (контроль облікових записів у Windows)

UI – інтерфейс користувача

UX – досвід користувача (User Experience)

WinAPI – Windows Application Programming Interface (WinAPI)

XTEST – X Test Extension (розширення X11 для емуляції подій)

ОС – операційна система

ID – Identifier (ідентифікатор, позначення)

PIL – Python Imaging Library (бібліотека для роботи із зображеннями в Python)

ВСТУП

У сучасних умовах розвитку інформаційних технологій та поширення цифрових систем обробки інформації все більшого значення набуває автоматизація рутинних операцій користувача. Автоклікери – спеціалізовані програмні засоби, призначені для автоматизації повторюваних дій на комп’ютері (наприклад, натискання клавіш чи кліки мишею) без участі користувача. Вони дозволяють суттєво спростити виконання однотипних завдань, значно скоротити час їх виконання і мінімізувати ризик помилок, пов’язаних з людським фактором.

Практичне застосування автоклікерів охоплює різні сфери діяльності. Так, в офісній роботі вони полегшують виконання рутинних операцій – наприклад, масового введення даних, заповнення електронних форм або обробки табличних документів – що дозволяє прискорити роботу працівників та знизити кількість помилок. У сфері розробки та тестування програмного забезпечення подібні програми використовують для автоматизованого тестування графічних інтерфейсів: програма може імітувати дії користувача при багаторазовому виконанні кліків і натискань, що дозволяє перевірити стійкість системи та виявити потенційні помилки. В ігровій індустрії автоклікери часто застосовують для виконання одноманітних завдань (наприклад, автоматичного «фарму» ресурсів або повторюваних квестів), що дає геймерам змогу зосередитися на більш цікавих аспектах гри.

Метою кваліфікаційної роботи є покращити ефективність виконання рутинних завдань користувача шляхом створення інструментарію автоклікера з розширеною функціональністю. Зокрема, передбачається реалізація гнучкої системи макросів із точним записом і відтворенням дій, підтримкою глобальних гарячих клавіш, а також циклічного режиму автокліку і мультикліку. Очікуваним результатом є інструмент, що значно скорочує час користувача на повторювані операції та підвищує загальну

продуктивність роботи в середовищах Windows і Linux.

Об’єктом дослідження є повторювані події введення користувача, такі як натискання клавіш, рухи курсора та клацання миші, у середовищах операційних систем Windows і Linux. Ці події формують базову взаємодію користувача з програмним забезпеченням і стають основою для побудови макросів та автоматизації.

Предметом розробки є методи й технічні засоби для фіксації, збереження та відтворення послідовностей дій користувача, а також підходи до глобального перехоплення гарячих клавіш і програмної емуляції кліків миші. Аналізуються алгоритми роботи в багатопоточному режимі, формати збереження конфігурацій, механізми циклічного повторення та взаємодія з файловим сховищем для реалізації кросплатформного автоклікера.

Для досягнення поставленої мети сформульовано такі завдання:

– проаналізувати існуючі інструменти автоматизації введення (автоклікери, скриптові утиліти), виявити їхні сильні й слабкі сторони;

– вибрати засоби та технології розробки кросплатформного автоклікера;

– розробити архітектуру системи автоматизації введення даних;

– реалізувати основний функціонал автоклікера (гарячі клавіші, запис/відтворення дій, фоновий режим, автоклік);

– розробити графічний інтерфейс користувача;

– забезпечити кросплатформеність розробленого програмного продукту;

– провести тестування та налагодження програмного забезпечення.

1 СПЕЦІФІКАЦІЯ ВИМОГ ДО ПРОГРАМНОГО ЗАСТОСУНКУ

1.1 Опис предметної області

Автоклікер – це програмний або апаратний інструмент, призначений для автоматизації процесу натискання кнопок миші чи клавіш клавіатури за попередньо заданими параметрами. Він може бути реалізований як окремий додаток під Windows, macOS чи Linux із графічним інтерфейсом або командним рядком, або як вбудована функція в периферійні пристрої (наприклад, миша з підтримкою автокліку). За принципом роботи автоклікери можуть виконувати кліки у фіксованих координатах, реагувати на події системи (зміну кольору пікселів, появу вікон або повідомлень) або додавати випадкові зміщення та інтервали між натисканнями з метою імітації людської поведінки.

Типовий набір функцій програмного автоклікера включає налаштування інтервалу між кліками, вибір типу кліку (ліве, праве, подвійне), встановлення кількості повторень або нескінченного циклу, а також можливість збереження профілів і запису макросів – послідовностей дій користувача з переміщеннями миші, комбінаціями клавіш і затримками. Деякі рішення надають планувальник для запуску автокліку в заданий час або через певний проміжок часу, а також API для інтеграції з іншими програмами.

Автоклікери застосовуються у тестуванні програмного забезпечення для перевірки інтерфейсу та стрес-тестів, в ігровій індустрії для автоматизації рутинних завдань, в офісній роботі для прискорення введення даних та обробки таблиць, у графічному дизайні для однотипних дій з обробки зображень, а також як допоміжні засоби для користувачів з обмеженими можливостями моторики.

Завдяки автоклікерам організації та окремі користувачі можуть значно підвищити продуктивність, зменшити ймовірність помилок і стандартизувати виконання повторюваних завдань. Разом з тим, використання автоклікерів

може зустрічати обмеження: антивірусне ПЗ або ігрові системи захисту можуть розглядати їх як небажане програмне забезпечення, інтерфейс цільового додатка може змінюватися, що порушить сценарій автоматизації, а у випадку ігор існує ризик блокування акаунтів через порушення правил користування.

1.2 Аналіз існуючих програмних рішень

Перед оглядом функціоналу варто відзначити, що практично всі відомі й якісні автоклікери мають інтерфейс англійською мовою, а деякі перекладені лише російською. Автоклікери з українським інтерфейсом зустрічаються вкрай рідко або взагалі відсутні на ринку. Значна частина користувачів таких програм – діти та підлітки, які застосовують їх для автоматизації рутинних дій в іграх, але часто не володіють іноземними мовами або мають обмежене знання англійської. Саме тому розробка повністю україномовного інтерфейсу є важливою перевагою, що робить його максимально доступним для української аудиторії.

«GS Auto Clicker»[1] – класичний інструмент для Windows (Vista/7/64-bit) з простим інтерфейсом англійською мовою. Головне вікно програми представлене на рис. 1.1.

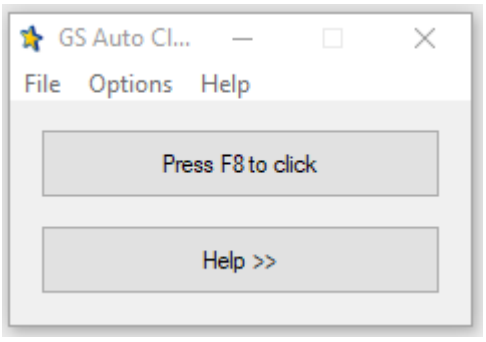

Рисунок 1.1 – Гловне вікно «GS Auto Clicker»

Програма запускається за натисканням заданої гарячої клавіші та циклічно клікає по вказаній кнопці миші. Користувач має можливість вибрати одиночне або подвійне натискання, задати інтервали між кліками в годинах, хвилинах, секундах або мілісекундах, а також обрати кількість повторень чи безкінечний режим. Є підтримка запису макросів, однак цей функціонал обмежується лише простою фіксацією кліків без реального аналізу або обробки подій клавіатури, координат миші чи часових затримок.

Програма не дозволяє редагування записаних дій, їх збереження у файл чи налаштування поведінки під час відтворення. Таким чином, GS Auto Clicker є прикладом найпростішого автоматизатора кліків без можливості розширення або налаштування складних сценаріїв. Серед головних недоліків – відсутність підтримки інших операційних систем (окрім Windows), вузький набір функцій і відсутність гнучкого програмного керування чи інтеграції з іншими інструментами.

«AutoHotkey»(AHK)[2] – потужна безкоштовна скриптова мова для Windows із інтерфейсом та документацією англійською мовою (частково доступні ресурси російською), призначена для автоматизації практично будь-яких завдань на робочому столі. Головне вікно програми представлене на рис. 1.2.

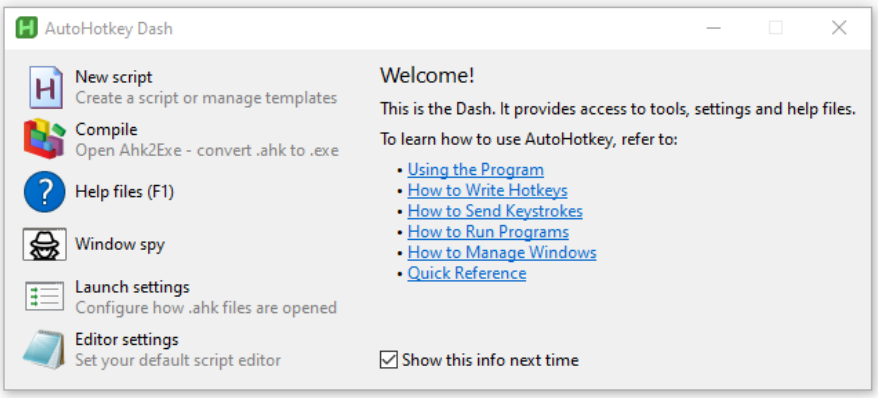

Рисунок 1.2 – Гловне вікно «AutoHotkey»

За допомогою AHK можна:

– створювати макроси та гарячі клавіші для миші й клавіатури (натискання, рух курсора, введення тексту);

– ремаппінг будь-яких клавіш та комбінацій (наприклад, переназначення CapsLock у Ctrl або створення складних мультимедійних комбінацій);

– автоматичне заповнення форм та діалогів (наприклад, вставка шаблонного тексту чи даних із буферу обміну);

– запуск зовнішніх програм із передачею параметрів, роботу з вікнами (переміщення, зміна розмірів, приховування) і маніпуляції з файлами.

AHK надає вбудований Recorder для фіксації базових макросів: він записує події миші і клавіатури, зберігаючи їх у вигляді скрипту AHK. Однак цей записувач має обмежений функціонал – він фіксує послідовності подій без жодних часових корекцій, не дозволяє інтерполяцію рухів курсора чи складні умови відтворення.

Для досвідчених користувачів «AutoHotkey» пропонує повноцінні мовні конструкції: цикли, умовні оператори, функції, обробка винятків, робота з Component Object Model та DLL-інтерфейсами. Це робить AHK дуже гнучкою: написавши кілька рядків коду, можна автоматизувати найскладніший сценарій, взаємодіячи напряму зі змістом вікон.

Проте «AutoHotkey» має суттєві обмеження:

– працює виключно на Windows (Mac та Linux підтримуються лише експериментальними портами або за допомогою емуляції через Wine);

– відсутність інтегрованого GUI-редактора за замовчуванням – графічні діалоги потрібно створювати вручну через комбінації команд ;

– відсутність централізованого менеджера пакетів – додаткові скрипти та бібліотеки поширюються окремо на користувацьких ресурсах;

– обмежений записувач макросів – він не фіксує складні події (наприклад, рухи миші криволінійно), не дозволяє редагувати часові інтервали під час запису;

– крива навчання – хоча базові макроси прості, для повноцінного використання можливостей АНК потрібне знайомство зі скриптовою мовою та синтаксисом.

Python-утиліти для автоматизації[3] – скрипти на Python, які використовують бібліотеки автоматизації (наприклад, PyAutoGUI, pynput). Такі рішення можуть бути кросплатформенними, оскільки бібліотеки підтримують Windows, macOS та Linux. Вони дають велику гнучкість – руху миші, перевірки зображень на екрані тощо. Наприклад, PyAutoGUI надає керування курсором, кліки та введення з клавіатури; pynput дозволяє контролювати мишу і клавіатуру, а також прослуховувати події вводу. Головне обмеження – відсутність готового GUI за замовчуванням і потреби навичок програмування. Також слід зазначити інструмент xdotool (Linux/X11): це окрема утиліта, що емулює події миші/клавіатури через розширення XTEST. Вона працює в Linux (на X11, не на Wayland) і використовується через командний рядок.

Проведений аналіз аналогів свідчить.

Функціональні обмеження. Більшість популярних автоклікерів («GS Auto Clicker», «AutoHotkey») забезпечують лише базовий набір можливостей: регулювання інтервалів натискань, прості макроси, вибір кнопок миші та прив’язку гарячих клавіш.

Платформна прив’язка. «GS Auto Clicker» і «AutoHotkey» працюють виключно на Windows і не мають нативних портів для інших ОС. Python-утиліти на базі PyAutoGUI і pynput є кросплатформовими, проте не оснащені готовим GUI, їх застосування вимагає програмування.

Мовний бар’єр. Практично всі інтерфейси та документація існуючих рішень виконані англійською (деякі перекладені російською), що створює складнощі для української аудиторії, особливо дітей і підлітків.

Баланс гнучкості та зручності. Існуючі рішення або легко налаштовуються без коду (але мають вузький функціонал і лише Windows-підтримку), або дають велику гнучкість через програмування.

1.3 Функціональні вимоги до програмного застосунку

Діаграма прецедентів (Use Case) описує взаємодію «Користувача» з системою автоклікера. Користувач може налаштовувати гарячі клавіші, записувати і зберігати макроси, відтворювати макроси, вибирати фон та виконувати автоклік чи мультиклік. UML Use Case–діаграма фіксує набір таких дій (прецедентів) у контексті зовнішнього актора.

Нижче на рис. 1.3 наведено результат виконання PlantUML–коду діаграми прецедентів для розглядуваної системи.

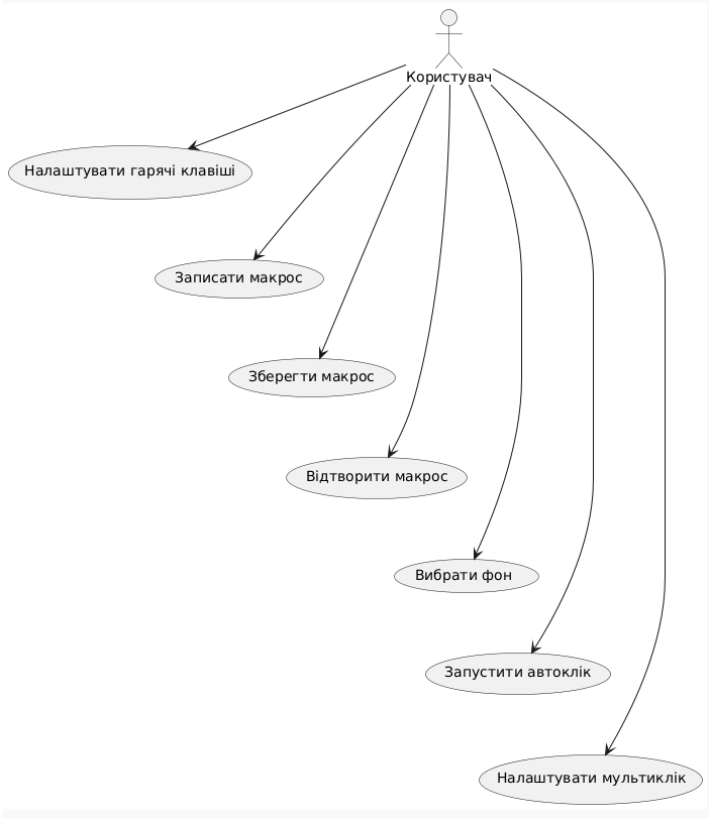

Рисунок 1.3 – Діаграма прецедентів для розглядуваної системи

Опис варіантів використання:

1) налаштувати гарячі клавіші – задається комбінація клавіш для запуску/зупинки запису макросу, автокліку або мультикліку тощо;

2) записати макрос – починається фіксація послідовності дій (натискання клавіш, клацання та рухи миші, затримки), які надалі будуть доступні для збереження та відтворення;

3) зберегти макрос – після завершення запису макросу користувач зберігає його до файлу;

4) відтворити макрос – користувач вибирає збережений макрос і запускає його виконання, система автоматично виконує зафіксовані дії у заданому порядку;

5) вибрати фон – відкривається вікно галереї фонів; користувач обирає одне з доступних зображень або завантажує своє, яке стає фоном інтерфейсу програми;

6) запустити автоклік – активується постійне автоматичне клацання у вибраному режимі (наприклад із заданим інтервалом);

7) налаштувати мультиклік – задається набір координат для багатоточкового автоматичного клацання, після завершення користувач може запустити або зберегти налаштування для майбутнього використання.

Кожен з прецедентів відображає завершену дію, що виконується системою на запит користувача, забезпечуючи очікуваний результат: записані макроси, обраний фон або автоматичне клацання за заданими параметрами.

1.4 Нефункціональні вимоги

До програмного застосунку встановлено такі нефункціональні вимоги:

1. Користувацька зручність (Usability).

Інтерфейс повинен бути інтуїтивно зрозумілим, з єдиною панеллю керування, через яку користувач має швидкий доступ до основних функцій. Час реакції інтерфейсу на будь-яку дію користувача не повинен перевищувати 200 мс.

2. Продуктивність (Performance).

Усі ключові модулі мають ініціалізуватися за час, що не перевищує 2 секунд із моменту запуску програми. Перехід від команди користувача до початку автокліку чи відтворення макросу має відбуватися за час, що не перевищує 500 мс.

3. Крос-платформеність і портативність.

Застосунок має без змін у вихідному коді працювати під операційними системами Windows і Linux. Упаковка та запуск повинні здійснюватися за допомогою одного інтерпретатора Python без потреби додаткових сторонніх сервісів чи модулів.

4. Масштабованість і розширюваність.

Архітектура програми спроєктована за модульним принципом. Додавання нових режимів автоматизації, форматів збереження даних або підтримки інших ОС має здійснюватися шляхом створення додаткових модулів, без необхідності змін у вже існуючих компонентах.

5. Надійність і стійкість.

Програма повинна коректно обробляти всі виняткові ситуації (відсутність потрібних каталогів чи файлів, пошкоджені JSON-записи, помилки емуляції введення) та інформувати користувача про помилки без аварійного завершення роботи. У разі збоїв має бути передбачена можливість безпечного припинення поточних операцій і відновлення нормальної роботи інтерфейсу.

2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ

2.1 Огляд середовища розробки

Для створення додатку було обрано середовище розробки Visual Studio Code – легковагове кросплатформенне середовище розробки (IDE) від Microsoft, яке забезпечує швидку роботу та високу гнучкість за рахунок розширень. У проекті використовувалася версія 1.100.2, яка має такі ключові можливості:

- інтегрований термінал для запуску скриптів і утиліт без виходу з редактора;
- розширення Python (Microsoft Python Extension) з підтримкою IntelliSense, автодоповненням та перевіркою типів через Pylance;
- вбудований відлагоджувач (debugger) для покрокового виконання коду, установки break-point'ів і перегляду значень змінних у реальному часі;
- підтримка Git/GitHub через вбудований Source Control: зручне створення комітів, перегляд змін і управління гілками прямо в інтерфейсі VS Code;
- live Share для спільного редагування коду в реальному часі;
- набір корисних утиліт: GitLens (додаткова інформація про історію змін), Prettier (форматування коду) та ESLint (статичний аналіз).

Завдяки цим інструментам Visual Studio Code забезпечує зручне середовище для розробки, налагодження та управління версіями проєкту.

2.2 Технології використані в розробці

Python[4] – високо-рівнева інтерпретована мова програмування із широкою екосистемою бібліотек та вбудованою підтримкою різних операційних систем. У проєкті Python забезпечує реалізацію всього функціоналу автоклікера, від обробки подій до формування графічного

інтерфейсу. Обрана версія Python 3.12.4 дозволяє використовувати сучасні можливості мови і гарантує кросплатформенність виконання.

Pynput[5] – спеціалізована бібліотека для управління та моніторингу пристроїв введення (миші та клавіатури). За допомогою pynput можна програмично переміщувати курсор, емулювати натискання кнопок миші та клавіатури, а також перехоплювати події введення в режимі реального часу. Наприклад, клас pynput.mouse.Controller використовується для керування курсором, а pynput.mouse.Listener дозволяє реєструвати обробники подій миші (натискання, рух). Аналогічні класи є й для клавіатури (pynput.keyboard.Controller і Listener). Такі можливості забезпечують запис користувацьких дій (макросів) та їх відтворення.

Keyboard[6] – легка бібліотека для перехоплення подій клавіатури та роботи з гарячими клавішами. Вона дозволяє налаштовувати глобальні гарячі клавіші, симулювати натискання і читати натиснуті комбінації незалежно від фокусу вікна. Наприклад, можна задати наступну комбінацію клавіш Ctrl+a для виклику функції при натисканні. Бібліотека keyboard підтримує Windows і Linux (для Linux може знадобитися запуск із sudo). Її простий API полегшує реалізацію функції швидкого запуску/зупинки макросів клавіатурними комбінаціями.

Tkinter[7] – пакет для створення графічного інтерфейсу користувача у Python. Як оболонка Tcl/Tk, tkinter надає вікна, кнопки, поля вводу та інші віджети для взаємодії з користувачем. Оскільки tkinter доступний на Windows і більшості UNIX-платформ, його використання забезпечує кросплатформенний інтерфейс без додаткових залежностей. У проєкті елементи tkinter застосовуються для формування меню, додавання, редагування та запуску макросів, вибору інтервалів тощо.

Threading[4] – модуль бібліотеки Python для роботи з потоками. У автоклікері застосовується багатопотокова архітектура: окремі потоки виконують прослуховування подій миші/клавіатури чи відтворення макросів у фоновому режимі, не блокуючи графічний інтерфейс. Модуль threading

надає клас Thread для створення та запуску потоків. Завдяки цьому можна безперервно відстежувати натискання клавіш чи рухи миші (які також запускаються як слушачі у окремих потоках) паралельно з роботою GUI.

Ctypes[4] – модуль бібліотеки Python для виклику функцій із системних бібліотек. Він надає C-сумісні типи даних і дозволяє викликати нативні API без необхідності писати розширення на C. У контексті автоклікера ctypes використовується для найглибшого рівня емуляції вводу: прямого формування структур MOUSEINPUT/INPUT і виклику функції WinAPI SendInput, що гарантує мінімальну затримку та максимальну точність а також обходить можливі захисні механізми різних додатків.

Pillow (PIL)[8] – потужна бібліотека для обробки растрових зображень у Python. У проекті вона використовується для:

– завантаження різних форматів (PNG, JPEG, BMP) та приведення їх до єдиного формату RGBA, що дозволяє працювати з прозорістю;

– конвертації зображень і застосування фільтрів (наприклад, LANCZOS) для якісного масштабування;

– створення ескізів (thumbnail) для швидкого відображення у вікні вибору фону без значних витрат пам’яті.

tkinter.filedialog + shutil[4] – поєднання стандартних модулів для організації завантаження зовнішніх файлів у інтерфейсі та копіювання їх до робочого каталогу програми.

Json[4] – стандартний модуль для серіалізації й десеріалізації даних у форматі JSON. У проєкті він використовується для збереження та завантаження.

platform та sys[4] – поєднання модулів для визначення поточної операційної системи й коректного знаходження ресурсів як у режимі скрипта, так і в зібраному виконуваному файлі (наприклад, через PyInstaller).

1) platform.system() дозволяє у коді розгортати різні гілки логіки залежно від ОС:

– на Windows в проекті використовуються WinAPI через ctypes;

– на Linux – subprocess з setxkbmap та pynput.

2) sys.MEIPASS – атрибут, який створює PyInstaller при упаковці програми в one-file mode. Він вказує на тимчасову директорію, куди розпаковуються ресурси (іконки, шаблони, фони).

3) Якщо sys.MEIPASS відсутній (програма запускається як звичайний скрипт), використовується шлях до поточного файлу _file_.

Таким чином, функція resource_path(...) коректно знаходить іконки, зображення фонів і інші зовнішні файли незалежно від способу запуску програми.

Subprocess[4] – модуль для запуску зовнішніх системних команд із Python. У проекті він застосовується, для перемикання розкладки клавіатури на Linux (за допомогою setxkbmap), а також може бути використаний для інших утиліт або скриптів. Головні моменти:

– надійне виконання: subprocess.run блокує виконання поточного потоку до завершення команди, що зручно, коли потрібно дочекатися результату (наприклад, успішного перемикання розкладки) перед продовженням;

– контроль коду завершення: параметр check=True автоматично піднімає CalledProcessError, якщо команда повернула ненульовий код помилки, що полегшує обробку невдалих викликів;

– гнучка передача аргументів: передавати список параметрів без необхідності збирання рядкового виклику через оболонку.

2.3 Критичний аналіз підходів до розробки

Кросплатформеність.

Обрана технологія з Python та бібліотеками pynput і tkinter гарантує роботу на Windows та Linux без суттєвих змін у коді. Наприклад, tkinter

входить до стандартної бібліотеки і працює на згаданих ОС, а pynput спільно з бібліотекою keyboard забезпечує доступ до подій введення незалежно від платформи. Альтернативи могли б включати розробку окремих програм під конкретні ОС (наприклад, .NET для Windows), що ускладнило б підтримку. Також існують інші Python-бібліотеки: наприклад, PyAutoGUI дозволяє клікати та управляти курсором на всіх платформах, але вона не призначена для фонової реєстрації подій. Обраний підхід має перевагу з точки зору універсальності та мінімального налаштування середовища. Обмеження полягає в тому, що деякі використовувані бібліотеки мають часткову підтримку (наприклад, keyboard потребує прав root на Linux).

Фоновий режим та багатозадачність.

Для забезпечення роботи автоклікера у фоновому режимі (без зависання графічного інтерфейсу) використано багатопоточність. Слухачі подій (pynput.Listener і keyboard) автоматично працюють в окремих потоках, що дозволяє одночасно приймати введення користувача і відтворювати макроси. Стандартний модуль threading надає простий механізм створення потоків. Цей підхід відрізняється від, наприклад, асинхронної моделі asyncio, яка у випадку з обробкою подій операційної системи менш очевидна у реалізації. Альтернатива – запуск окремого процесу чи сервісу для автоклікера – ускладнила б обмін даними з GUI. Обрана модульна багатопотокова архітектура забезпечує розділення завдань і підтримку фонової роботи.

Перехоплення подій клавіатури та миші.

У контексті розробки утиліт та автоматизованих скриптів необхідно точне перехоплення та емуляція вводу користувача через клавіатуру та мишу. Існуючі рішення для цього в Python переважно базуються на бібліотеках високого рівня, таких як pynput та keyboard. Наприклад, pynput надає зручні класи для підписки на події натискання клавіш чи руху миші (через pynput.keyboard.Listener та pynput.mouse.Listener), а бібліотека keyboard також дозволяє перехоплювати введення та реагувати на натискання

клавіш. Однак застосування цих бібліотек може призводити до виникнення затримок та певних обмежень. Зокрема, pynput був спочатку розроблений з орієнтацією на Linux, і в середовищі Windows його емуляція введення часто сприймається системою як синтетична, що знижує реалістичність взаємодії.

Водночас у середовищі Windows можливе використання модуля ctypes, який дозволяє викликати функції Windows API[9] безпосередньо з Python. Зокрема, функція SendInput із user32.dll призначена для емуляції подій клавіатури та миші на найнижчому рівні системи. Виклик SendInput забезпечує мінімальну затримку та високу точність введення, оскільки команди формуються на рівні системного API і обробляються майже як апаратні події. Це дає суттєву перевагу порівняно з бібліотекою pynput: хоча остання ефективно працює в Linux, у Windows її події можуть бути сгенеровані з більшою затримкою і вважатися системою як менш «природні». Наприклад, деякі події, згенеровані через pynput, сприймаються ОС як штучні, що може призводити до додаткових фільтрацій чи затримок обробки. Використання ж ctypes/SendInput дозволяє обійти ці обмеження, оскільки виклики формуються на рівні WinAPI без проміжних абстракцій, створюючи події, які система сприймає майже як події від реальної клавіатури чи миші.

Графічний інтерфейс.

Використання tkinter забезпечує базовий, проте функціональний GUI з мінімальними залежностями. Було обрано «легкий» фреймворк без необхідності ліцензування (що могло б статись з PyQt у комерційних цілях). На відміну від сучасних графічних бібліотек (Qt, GTK, Kivy тощо), tkinter пропонує обмежений набір віджетів і примітивний стиль, але є доволі стабільним і простим реалізації. Альтернативою міг бути PyQt/PySide (більш гнучкий у дизайні, але тяжчий по розміру) або веб-інтерфейс на Electron (переносимість за рахунок ресурсів). Обрана архітектура GUI дозволила швидко реалізувати всі необхідні елементи: списки макросів, поля вводу, кнопки запуску тощо. Єдиним обмеженням є те, що стандартними засобами

tkinter складно реалізувати новітні UX-елементи.

Загальні висновки.

Обрана архітектурна стратегія – модульний підхід з чітким розподілом компонентів:

– підсистема GUI (tkinter);

– підсистема захоплення подій (pynput, keyboard, ctypes/WinAPI);

– механізм обробки макросів (логіка запису/відтворення);

– підтримка багатопоточності (threading).

Кросплатформеність забезпечується використанням бібліотек високого рівня (pynput, keyboard, tkinter), що працюють як під Windows, так і під Linux із мінімальними змінами. tkinter гарантує стабільний базовий GUI в обох ОС, а pynput разом з keyboard дозволяють перехоплювати події введення, хоча для Linux перевага саме в тому, що права доступу (наприклад, root) часто не потрібні, або вони простіше налаштовуються, ніж у Windows.

Фоновий режим та багатозадачність реалізовано через багатопоточність за допомогою модуля threading: слухачі подій працюють в окремих потоках, що убезпечує GUI від блокувань. При цьому обмеження Global Interpreter Lock може бути критичним лише для інтенсивних обчислень. У типовому сценарії вводу/виводу багатопоточність повністю задовольняє вимоги до паралельного прослуховування подій, водночас забезпечуючи фонову роботу без «зависання» інтерфейсу.

Перехоплення подій клавіатури та миші у Linux здійснюється через pynput, що в цілому працює досить стабільно для прослуховування і емуляції. Натомість у Windows pynput може згенерувати «синтетичні» події із затримками або обмеженнями (наприклад, втрата циклічного повторення клавіші і тд.), тому для емуляції введення було додано низькорівневий виклик SendInput через ctypes. Це дозволяє обходити фільтрацію ОС і досягти максимальної швидкодії та точності емуляції. Такий підхід поєднує кросплатформеність (через pynput/keyboard) із високою продуктивністю в Windows (через ctypes/WinAPI).

Графічний інтерфейс на базі tkinter забезпечує мінімальні залежності та достатній набір віджетів для створення функціонального GUI. Хоча tkinter не дозволяє втілити сучасні UX-елементи без додаткових модулів, його простота й стабільність виправдовують вибір. У разі потреби оновлення дизайну або перенесення інтерфейсу на сучасніший рушій (Qt, Electron) структура програми дозволяє замінити відповідний модуль GUI без кардинальної переробки інших компонентів.

У підсумку, комбінація вищезазначених технологій дозволила реалізувати гнучку, розширювану систему:

– модулі GUI, Recorder, гарячі клавіші/елементи введення та робота з файлами чітко розділені й взаємодіють через простий API;

– кросплатформені бібліотеки забезпечують базову функціональність як під Windows, так і під Linux;

– низькорівневі виклики через ctypes/WinAPI додають Windows-версії високої точності емуляції введення.

Переваги цього підходу полягають у швидкості розробки, портативності та можливості подальшого супроводу (заміна модулів без впливу на інші компоненти). Недоліки полягають у необхідності окремої реалізації низькорівневих викликів для Windows через ctypes.

2.4 Проектування архітектури

Архітектура програми побудована за модульним принципом з чітким розподілом обов’язків між компонентами. Основні компоненти системи включають: графічний інтерфейс (GUI), модуль запису/відтворення макросів та автоклікер з мультикліком (Recorder), модуль обробки гарячих клавіш і модуль роботи з файловою системою (збереження/завантаження даних). Графічний інтерфейс відповідальний за відображення елементів керування і прийом команд від користувача (кнопки, меню, поля введення). При натисканні користувачем кнопок у GUI генеруються події, які передаються

до модуля: наприклад, команда «Почати запис» передається до модуля Recorder.

Модуль Recorder реалізує логіку фіксації користувацьких дій – він відслідковує натискання клавіш і рухи/кліки миші під час запису макросу, зберігає їх у внутрішню структуру даних і потім забезпечує їх відтворення за командою. Гарячі клавіші перехоплюють глобальні комбінації клавіатури й активує відповідні функції програми незалежно від того, яке вікно активне в даний момент. Робота з файловою системою відповідає за збереження записаних макросів та конфігурацій користувача у файли та їхнє завантаження при старті програми.

Взаємодія між компонентами здійснюється через виклики методів і обмін подіями. Наприклад, коли користувач у графічному інтерфейсі обирає «Запис» макросу, відповідна функція GUI викликає метод start_recording() модуля Recorder. Гарячі клавіші працюють паралельно, перехоплюючи натискання клавіш і викликаючи методи програми (наприклад, зупинку відтворення макросу) незалежно від основного потоку роботи GUI.

Підтримка двох операційних систем забезпечується використанням кросплатформових бібліотек і умовних конструкцій у коді. Наприклад, для графічного інтерфейсу застосований кросплатформовий фреймворк (наприклад Tkinter), який однаково працює під Windows і Linux. Для емуляції клавіатурних та мишачих подій спочатку використовується бібліотека platform, що в коді визначає з якої ОС відкрито додаток й визиває відповідні методи для потрібної системи. Місця збереження файлів та формати шляхів узгоджуються з особливостями ОС: наприклад, застосунок автоматично обирає відповідний каталог для зберігання макросів на поточній системі.

Архітектуру програми можна поділити на такі підсистеми, кожні з яких відповідають за окремий аспект роботи автоклікера рис. 2.1. Виділено такі компоненти:

1. Графічний інтерфейс (UI).

Побудовано на Tkinter: головне вікно та додаткові діалоги (галерея

фонів, налаштування гарячих клавіш, список макросів, вибір точок мультикліку та діалог збереження макросу).

2. Recorder (Менеджер макросів, автоклік, мультиклік):

– макроси: запис, збереження і відтворення послідовностей дій;

– автоклік: циклічне клацання у вибраному режимі;

– мультиклік: клацання за набором точок, які користувач вибирає в GUI;

– налаштування мультикліку: збереження й завантаження конфігурацій цих точок у файл для подальшого використання.

3. HotkeyManager (Менеджер гарячих клавіш).

Перехоплює глобальні комбінації клавіш і ініціює у Recorder відповідні дії (запис/відтворення/автоклік/мультиклік).

4. BackgroundManager (Менеджер фонів).

Галерея зображень із папки background/, додавання/видалення файлів і запам’ятовування останнього вибраного фону.

5. Storage (Сховище макросів).

Папка recordings/ із JSON–файлами макросів. Recorder зчитує їх для відтворення і записує нові записи.

6. MultiConfigStorage (Конфігурації мультикліку).

Окремиа папка містить збережені набори точок і параметри мультикліку. Recorder звертається до цього сховища для завантаження та збереження конфігурацій мультикліку.

7. Settings (загальні налаштування).

– hotkeys.json – комбінації гарячих клавіш;

– repeat_settings.json – інтервали та кількість повторів для макросів;

– last_background.txt – ім’я останнього фону.

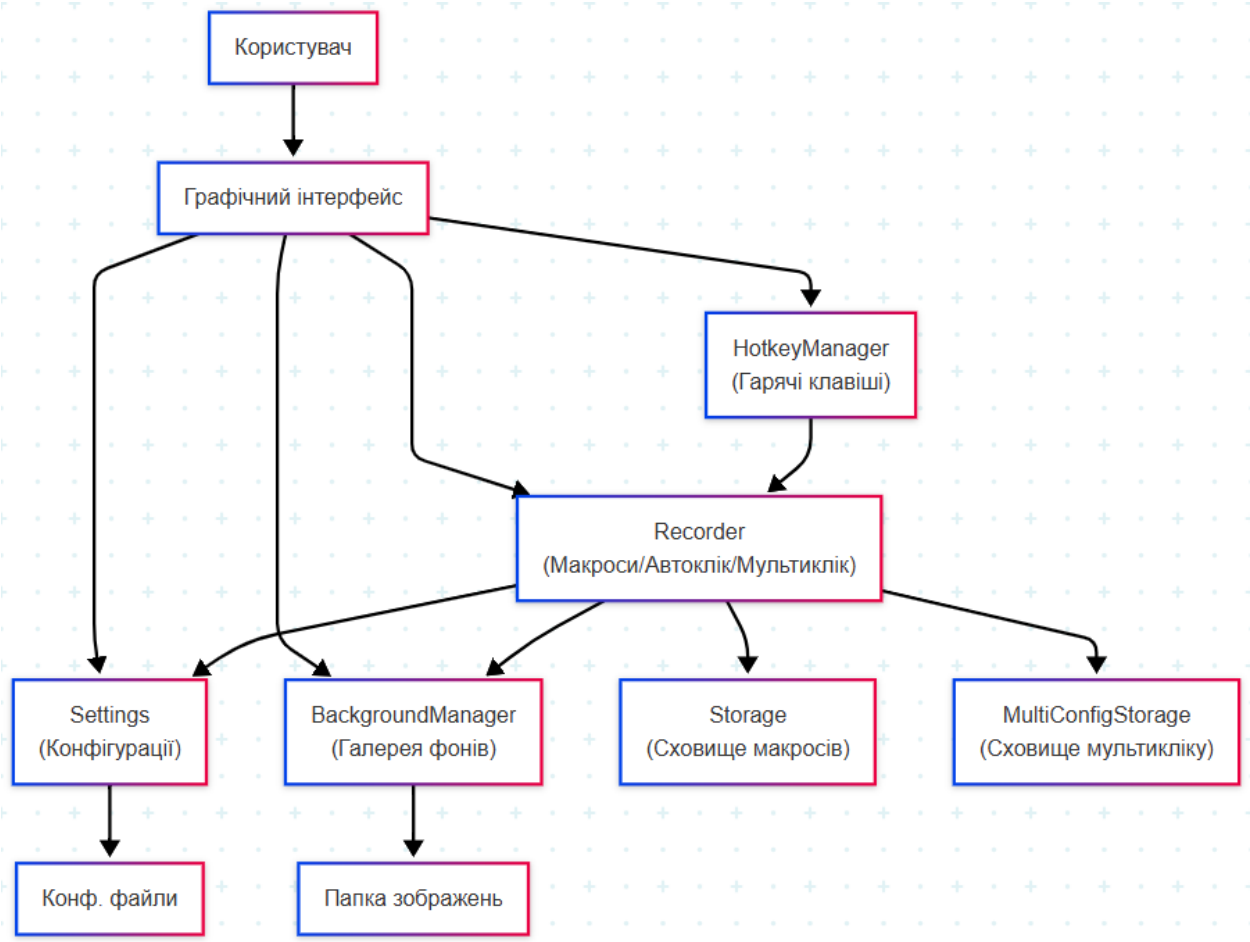

Рисунок 2.1 – Компонентна діаграма архітектури програми

Пояснення до компонентної діаграми: користувач звертається до UI. Через GUI обирає операції: запис/відтворення макросу, автоклік, мультиклік, вибір фону, налаштування гарячих клавіш і параметрів повтору.

UI звертається до Recorder. Інтерфейс делегує Recorder команди:

– почати/зупинити запис;

– відтворити макрос;

– активувати автоклік;

– активувати мультиклік.

UI звертається до HotkeyManager. Меню налаштувань гарячих клавіш надсилає нові комбінації, HotkeyManager починає обробку клавіатурних подій.

UI звертається до BackgroundManager. Галерея фонів відображає прев'ю, а після вибору оновлює фон UI та зберігає останній вибір.

UI звертається до SettingsUI. Вікно загальних налаштувань працює з файлами hotkeys.json, repeat_settings.json, last_background.txt.

Recorder звертається до Storage. Під час збереження макросів Recorder створює новий JSON–файл у recordings/; при ініціалізації завантажує список наявних макросів.

Recorder звертається до MultiConfigStorage. Зберігає й зчитує набори координат та параметри мультикліку, що дозволяє користувачу відтворювати мультиклік–послідовності в будь–який момент.

Recorder звертається до SettingsUI / BackgroundManager. Використовує загальні налаштування для інтервалів і повторів макросів, та звертається до BackgroundManager, якщо макрос чи мультиклік включає зміну фону.

HotkeyManager звертається до Recorder. При спрацюванні гарячих клавіш надходять запити для Recorder на початок/зупинку запису, відтворення або активацію автокліку/мультикліку.

BackgroundManager звертається до FS. Читає фізичні файли з каталогу background/.

SettingsUI звертається до FS_Cfg. Зчитує та оновлює конфігураційні файли (hotkeys.json, repeat_settings.json, last_background.txt).

2.4 Діаграма класів

На рис. 2.2 для ілюстрації внутрішньої структури підсистем наведено UML–діаграму класів, що відображає основні об'єкти програми та їхні взаємозв'язки.

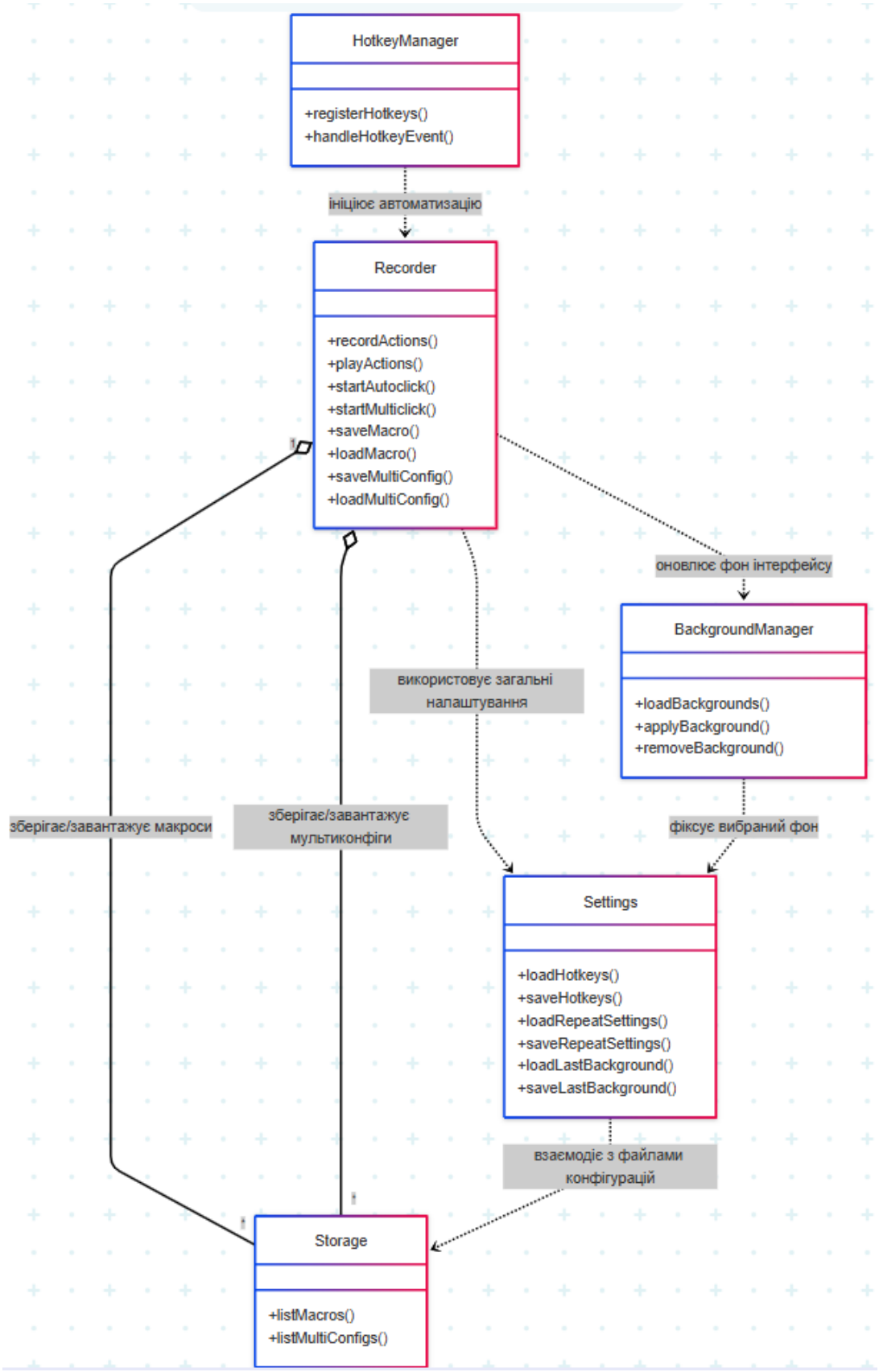

Рисунок 2.2 – Діаграма класів

Пояснення до зв’язків на діаграмі класів.

Композиція Recorder до Storage.

Об’єкт Recorder відповідає за збереження у файловому сховищі та завантаження з нього двох типів даних:

– макросів – після завершення запису викликається метод збереження, що створює JSON–файл у папці; при відтворенні макросів Recorder запитує Storage перелік доступних файлів і читає обраний JSON;

– конфігурацій мультикліку – набори точок і параметри мультикліку також серіалізуються у файл в окрему папку, а при запуску Recorder запитує Storage, щоб підвантажити ці налаштування.

Залежність Recorder до Settings.

Перед виконанням будь–якої дії автоматизації Recorder звертається до Settings, щоб отримати актуальні:

– комбінації гарячих клавіш;

– інтервали повтору макросів;

– ім’я останнього фону для застосування під час відтворення макросу або мультикліку.

Recorder не працює з цими значеннями безпосередньо з файлів, а використовує інтерфейс Settings для їх централізованого зчитування.

Залежність Recorder до BackgroundManager.

Під час виконання макросу чи мультикліку, якщо до сценарію входить зміна фону, Recorder викликає методи BackgroundManager для підвантаження та відображення нового зображення в інтерфейсі.

Залежність HotkeyManager до Recorder.

Коли HotkeyManager виявляє натискання однієї з налаштованих комбінацій (наприклад, для початку/зупинки запису або активації автокліку), він безпосередньо викликає у Recorder відповідний метод, ініціюючи потрібний режим автоматизації.

Залежність BackgroundManager до Settings.

Після вибору нового фону BackgroundManager звертається до Settings, щоб зберегти ім’я обраного зображення у файл. Це гарантує, що при наступному запуску програми саме цей фон буде застосовано автоматично.

Залежність Settings до Storage.

Settings використовує Storage для роботи з конфігураційними файлами:

– читання наявних файлів (hotkeys.json, repeat_settings.json) при старті програми (через запит Storage значень ключів та параметрів повтору);

– запис оновлених конфігурацій після того, як користувач змінив налаштування у відповідному діалоговому вікні.

Таким чином, кожен зв’язок показує конкретний сценарій звернення одного класу до іншого та розподіл відповідальностей між компонентами.

2.6 Діаграма послідовності дії

Цей сценарій ілюструє, як система відтворює раніше записану послідовність дій користувача (натискання клавіш, рухи й кліки миші, затримки). Діаграма на рис. 2.3 показує обмін повідомленнями між UI, Recorder і допоміжними підсистемами.

Пояснення до діаграми послідовності:

1. Користувач звертається до UI

Користувач через графічний інтерфейс ініціює відтворення макросу, натискаючи відповідну кнопку.

2. UI звертається до Recorder: інтерфейс викликає метод play(macroId) у підсистемі Recorder, передаючи ідентифікатор обраного макросу.

3. Recorder звертається до Storage

Recorder завантажує збережений макрос з файлового сховища, отримуючи список подій macroData.

4. Recorder звертається до Settings.

Перед безпосереднім виконанням Recorder звертається до Settings, щоб

отримати параметри циклічного відтворення: repeatCount (кількість повторів) і interval (час паузи між повтореннями).

5. Зовнішня петля (loop [повторити repeatCount ітерацій із паузою interval])

Відображає повторення всього сценарію відтворення макросу задану кількість разів з паузою interval між ітераціями.

6. Внутрішня петля (loop [для кожної події в macroData])

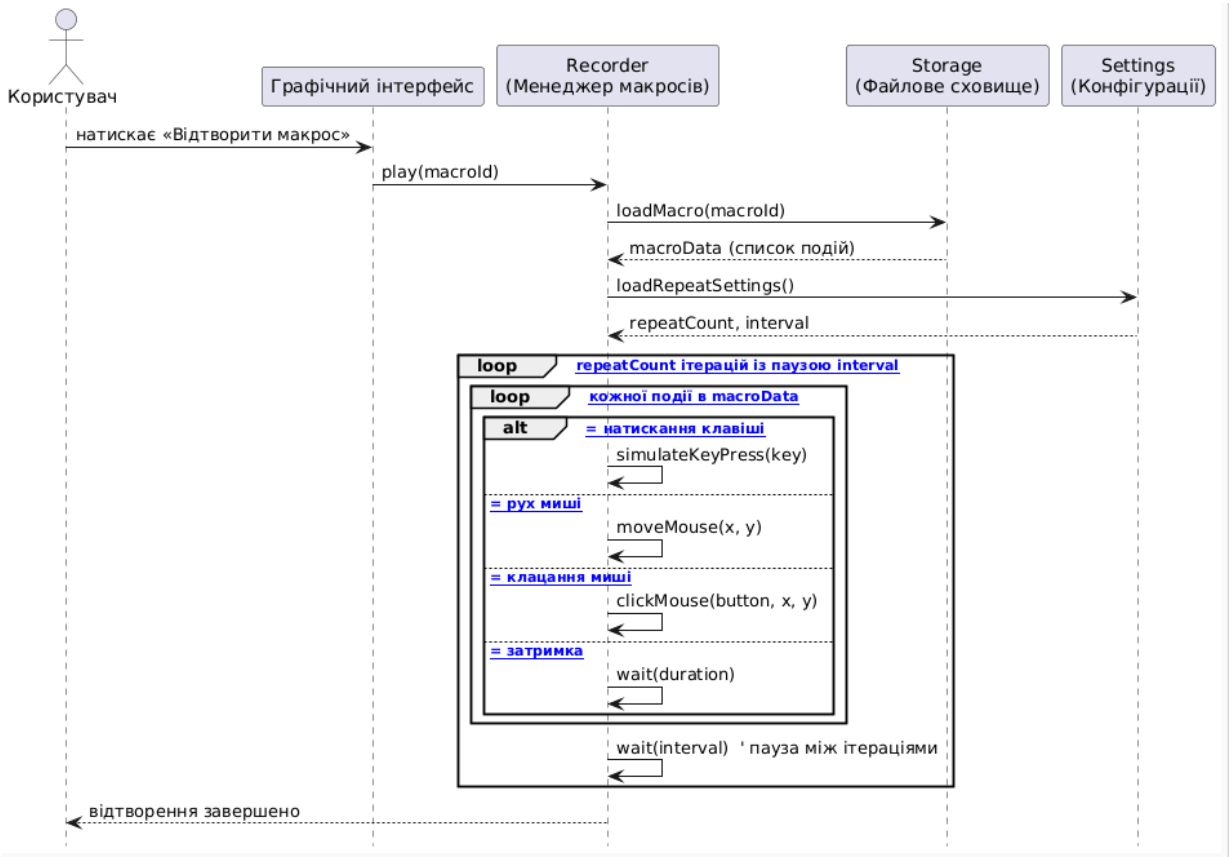

Рисунок 2.3 – Діаграма послідовності

Для кожної зареєстрованої події у макросі Recorder виконує одну з дій:

– натискання клавіші (simulateKeyPress);

– рух миші (moveMouse);

– клацання миші (clickMouse);

– затримка (wait).

7. Пауза між ітераціями

Після обробки всіх подій у макросі Recorder виконує затримку wait(interval) перед наступним циклом.

8. Recorder звертається до Користувач

Після завершення всіх ітерацій Recorder надсилає повідомлення користувачу про успішне завершення відтворення макросу.

Дана діаграма демонструє всю послідовність взаємодій між компонентами: UI, Recorder, Storage і Settings, а також ілюструє використання циклів для повторення та обробки подій.

3 ОПИС РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

3.1 Реалізація функціоналу гарячих клавіш

У розробленому автоклікері застосовуються глобальні гарячі клавіші (комбінації клавіш), які дозволяють користувачу оперативно керувати записом та відтворенням макросів і автокліку незалежно від фокусу вікна програми. Зокрема, гарячі клавіші призначено для запуску запису дій, зупинки запису, запуску відтворення останнього записаного макросу, увімкнення та вимкнення автоклікера тощо. Такий підхід забезпечує зручне керування без необхідності постійно взаємодіяти з графічним інтерфейсом, оскільки бібліотека keyboard перехоплює глобальні натискання клавіш навіть у фоновому режимі. Наприклад, у коді програми при реєстрації гарячих клавіш задаються функції–обробники, які викликають методи підсистеми запису/відтворення. Завдяки цьому користувач може миттєво стартувати чи зупиняти запис дій та автоклік без кліків у інтерфейсі.

За замовчуванням визначені гарячі клавіші у файлі конфігурації hotkeys.json (або в коді за відсутності цього файлу) задано стандартний набір комбінацій. Наприклад, за замовчуванням програма очікує наступні дії та поєднання клавіш:

- start – F2 (запуск запису макросу);
- stop – Ctrl+W (зупинка запису);
- force_stop – Ctrl+E (примусова зупинка відтворення);
- play – Ctrl+R (відтворити останній запис);
- autoclick – Ctrl+A (увімкнути автоклікер);
- force_stop_autoclick – Ctrl+S (зупинити автоклікер);
- multi_stop – Ctrl+D (зупинити мультивідтворення).

Такі значення відповідають словнику DEFAULT_HOTKEYS у коді додатку. Якщо файл hotkeys.json присутній у робочій папці, то при завантаженні параметрів (функція load_hotkey_config) спочатку зчитуються користувацькі налаштування, а потім для відсутніх ключів підставляються

значення за замовчуванням. Наведений фрагмент коду демонструє цю логіку:

```
# Завантаження конфігурації гарячих клавіш з файла або
використання стандартної
HOTKEYS_FILE = "hotkeys.json"
DEFAULT_HOTKEYS = {
    'start': 'F2',
    'stop': 'F3',
    'force_stop': 'ctrl+e',
    'play': 'ctrl+r',
    'autoclick': 'ctrl+a',
    'force_stop_autoclick': 'ctrl+s',
    'multi_stop': 'ctrl+d'
}

def load_hotkey_config():
    if os.path.exists(HOTKEYS_FILE):
        try:
            with open(HOTKEYS_FILE, 'r', encoding='utf-8') as f:
                cfg = json.load(f)
            # для кожної відсутньої клавіші підставляється
значення за замовчуванням
            for k, v in DEFAULT_HOTKEYS.items():
                cfg.setdefault(k, v)
            return cfg
        except:
            pass
    # якщо файл відсутній або пошкоджений, повернути копію
словника за замовчуванням
         return DEFAULT_HOTKEYS.copy()
```

На цьому етапі після завантаження конфігурації у змінну hotkey_config буде міститися актуальна комбінація клавіш. Вона використовується далі для реєстрації гарячих клавіш при запуску програми.

3.1.1 Реєстрація гарячих клавіш і обробка подій

При ініціалізації інтерфейсу (перед викликом root.mainloop()) викликається функція register_hotkeys(), яка реєструє глобальні комбінації через бібліотеку keyboard. Спочатку за необхідності видаляються попередні обробники гарячих клавіш (щоб уникнути дублювання). Потім для кожної дії додається відповідний обробник, наприклад:

```
def register_hotkeys():
    global hotkeys, hotkey_config
    # спочатку видалити старі прив’язки
    for hk_id in hotkeys.values():
        try:
            keyboard.remove_hotkey(hk_id)
        except KeyError:
            pass
    hotkeys.clear()
    # зареєструвати гарячі клавіші з обробниками
    hotkeys['start'] =
keyboard.add_hotkey(hotkey_config['start'], on_start,
suppress=True)
    hotkeys['stop']  =
keyboard.add_hotkey(hotkey_config['stop'], on_stop,
suppress=True)
    hotkeys['force_stop'] =
keyboard.add_hotkey(hotkey_config['force_stop'], on_force_stop,
suppress=True)
    hotkeys['play'] = keyboard.add_hotkey(hotkey_config['play'],
on_play, suppress=True)
    hotkeys['autoclick'] =
keyboard.add_hotkey(hotkey_config['autoclick'], on_autoclick,
suppress=True)
    hotkeys['force_stop_autoclick'] = keyboard.add_hotkey(
        hotkey_config['force_stop_autoclick'],
on_force_stop_autoclick, suppress=True)
    hotkeys['multi_stop'] =
keyboard.add_hotkey(hotkey_config['multi_stop'], on_multi_stop,
suppress=True)
    print("Гарячі клавіші зареєстровані:", hotkey_config)
```

Ключовим тут є виклик keyboard.add_hotkey(комбінація, функція, suppress=True): при натисканні заданої комбінації спрацьовує відповідна функція–обробник (наприклад, on_start, on_stop тощо), а параметр suppress=True дозволяє притиснути стандартну обробку цих клавіш у системі. Обробники on_start, on_stop тощо у коді реалізують логіку запуску та зупинки запису/відтворення, використовуючи внутрішні методи об’єкта recorder:

```
def on_start():
# перевірка, що в даний момент нічого не відтворюється, не
зациклюється і не записується
    if not recorder.is_playing and not recorder.is_looping and
not recorder.is_recording:
```

```
        recorder.start_recording()

def on_stop():
    # перевірка, чи зараз йде запис
    if recorder.is_recording:
        recorder.stop_recording()

def on_autoclick():
    recorder.start_autoclick()
```

Коли програма запускається, після створення всіх елементів GUI викликається register_hotkeys(), тому всі комбінації стають активними навіть до початку основного циклу mainloop(). Таким чином, обробка подій клавіатури відбувається у фоновому режимі, незважаючи на стан вікна додатку. На рис. 3.1 показано реєстрацію гарячих клавіш після запуску програми й головне вікно.

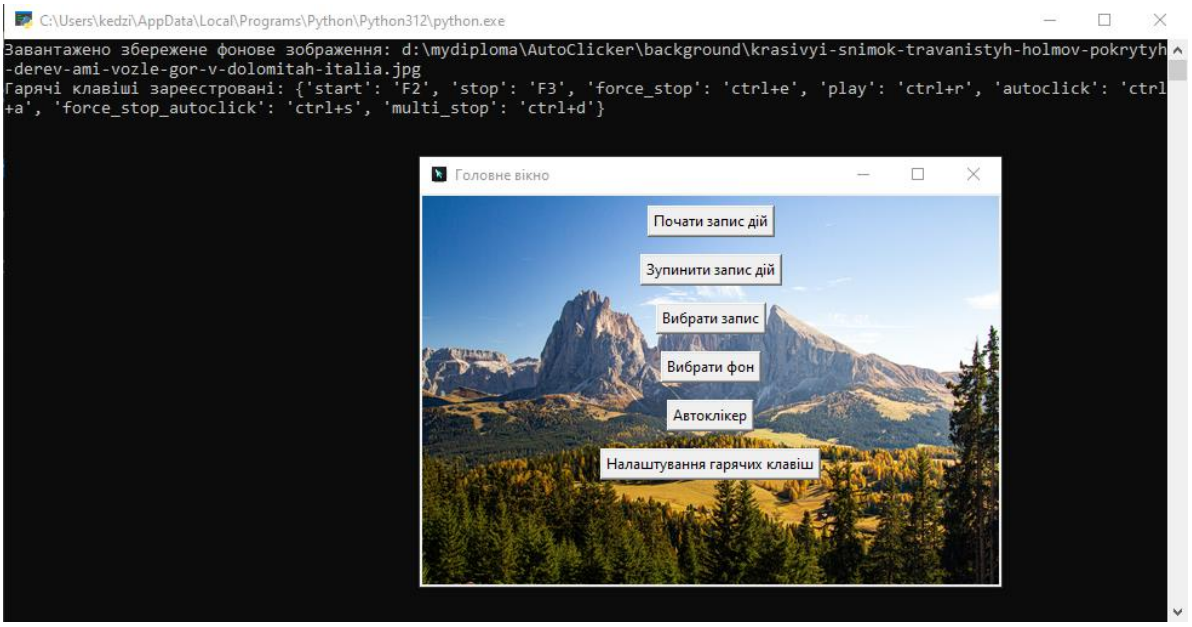

Рисунок 3.1 – Відображення консолі Windows з повідомленням про реєстрацію гарячих клавіш

3.1.2 Використання бібліотеки keyboard

Для прослуховування подій клавіатури використано модуль keyboard, який надає простий API для глобальних гарячих клавіш. Ця бібліотека підтримує операційні системи Windows і Linux (на Linux для глобального захоплення клавіш потрібні підвищені права sudo). Її можливості включають реєстрацію складних комбінацій та емуляцію натискань. Завдяки keyboard.add_hotkey програма відстежує поєднання незалежно від фокусу вікна.

Оскільки keyboard запускає обробку подій у власних потоках, GUI програми не блокується. Зокрема, у записувачі дій використовується багатопоточність: окремі потоки паралельно обробляють події миші та клавіатури та виконують відтворення. Це дозволяє одночасно інтерпретувати натискання гарячих клавіш у фоновому режимі, не перериваючи роботу інтерфейсу. Також бібліотека keyboard забезпечує інтерпретацію різних модифікаторів (Ctrl, Alt тощо), тому визначення комбінацій (наприклад, ctrl+a) однозначно запускає прив’язану функцію незалежно від розкладки клавіатури. На рис. 3.2 показано старт та зупинку запису в операційній системі Windows.

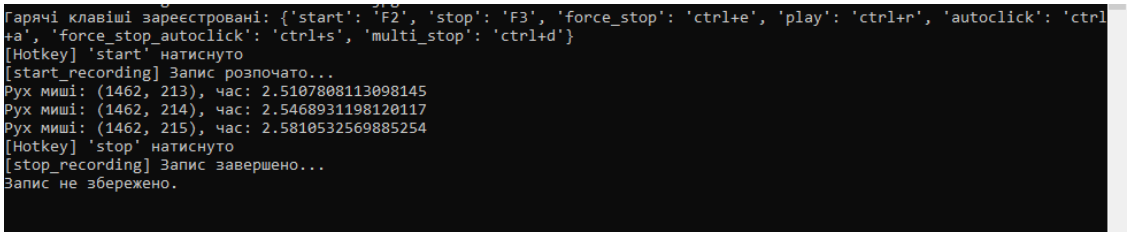

Рисунок 3.2 – Відображення натискання горячої клавіші для старту та зупинки запису в консолі Windows

3.1.3 Зміна та збереження гарячих клавіш користувачем

У програмі реалізовано можливість зміни гарячих клавіш через окреме діалогове вікно. При натисканні кнопки “Налаштування гарячих клавіш” відкривається вікно Tkinter з переліком дій і полями вводу для кожної комбінації. Значення полів ініціалізуються поточними налаштуваннями з hotkey_config. Користувач може ввести нові комбінації, після чого натиснути “Зберегти”. Обробник кнопки Зберегти у коді записує введені значення у словник конфігурації, викликає save_hotkey_config(cfg) (запис у файл JSON) і повторно виконує register_hotkeys(), оновлюючи прив’язки. Фрагмент коду ілюструє цей процес:

```
def on_save():
    # перебираються всі поля вводу налаштувань гарячих клавіш
    for key, var in entries.items():
        value = var.get().strip() # береться текст і обрізаються
пробіли

        # якщо поле пусте – повертається до значення за
замовчуванням
        if not value:
            value = DEFAULT_HOTKEYS.get(key, "")

        # оновлення конфігурації для цього ключа
        hotkey_config[key] = value

    save_hotkey_config(hotkey_config) # запис оновлених
налаштувань в файл
    register_hotkeys() # перереєстрація гарячих клавіш з новими
комбінаціями
    win.destroy() # закриття вікна налаштувань
```

Таким чином, після збереження нової конфігурації клавіші миттєво переналаштовуються у програмі. Якщо користувач залишить деякі поля порожніми, функція load_hotkey_config наступного разу підставить стандартні значення для відсутніх параметрів. Якщо ж дві дії отримують одну й ту ж комбінацію, то остання прив’язка фактично перезапише попередню. У коді реалізовано очищення старих прив’язок (remove_hotkey)

перед додаванням нових, що дозволяє мінімізувати дублювання. На рис. 3.3–3.6 показано переналаштування горячих клавіш та ситуацію коли поле буде залишене пустим.

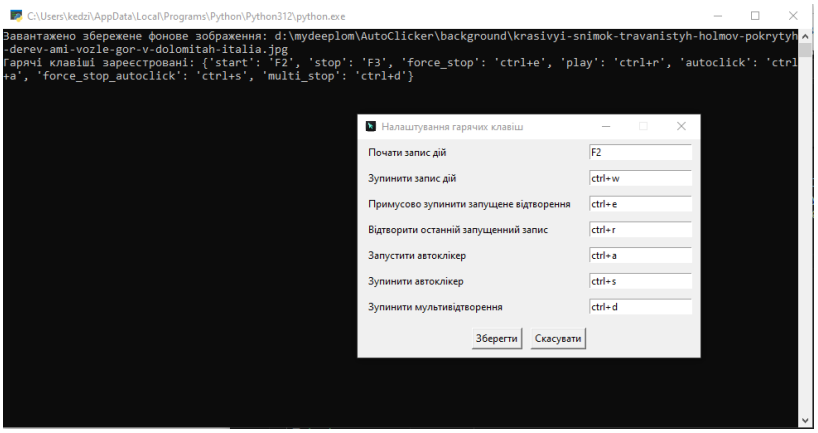

Рисунок 3.3 – Зміна значення в полі «Зупинити запис дій» з F3 на ctrl+w

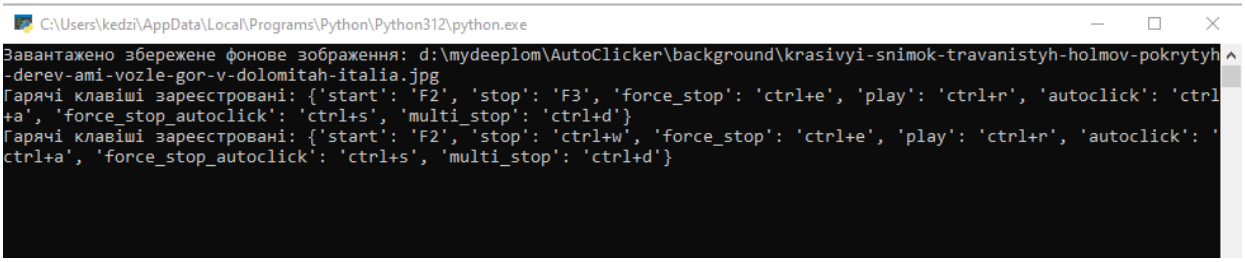

Рисунок 3.4 – Перегляд консолі після натискання кнопки «Зберегти»

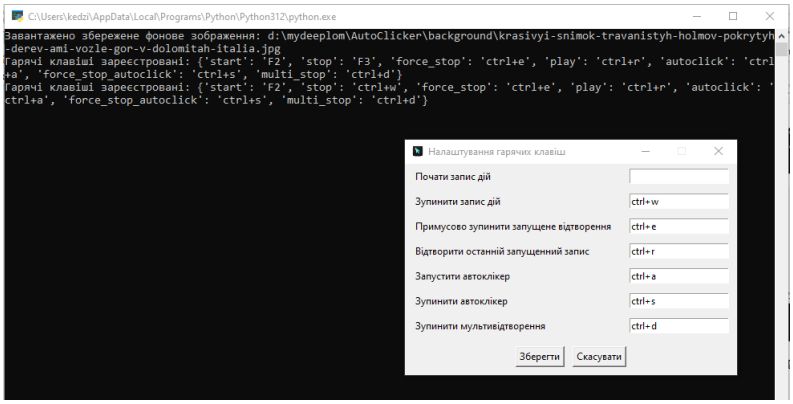

Рисунок 3.5 – Тестування ситуації коли поле буде порожнім

Рисунок 3.6 – Перегляд консолі після натискання кнопки «Зберегти» при порожньому полі

3.1.4 Кросплатформеність і фоновий режим

Для забезбечення кросплатформенності, було обрано рішення, яке працює однаково на Windows і Linux. Бібліотека keyboard підтримує обидві ОС (на Linux зазвичай потрібний запуск програми з sudo). У коді перевірка ОС використовується для надання прав (функція require_admin_privileges(), яка за необхідності перезапускає програму з піднятими правами через sudo на Linux або UAC на Windows). Після цього реєстрація гарячих клавіш виконується згідно з конфігурацією – і на Windows, і на Linux клавіші поводяться однаково. Отже, користувач, змінивши гарячі клавіші у Windows, може очікувати таку ж поведінку і на Linux без додаткових налаштувань. Це підкреслює кросплатформеність рішення та уніфіковану обробку вводу. На рис. 3.7–3.8 показано старт і зупинку запису гарячими клавішами, а також переналаштування гарячих клавіш.

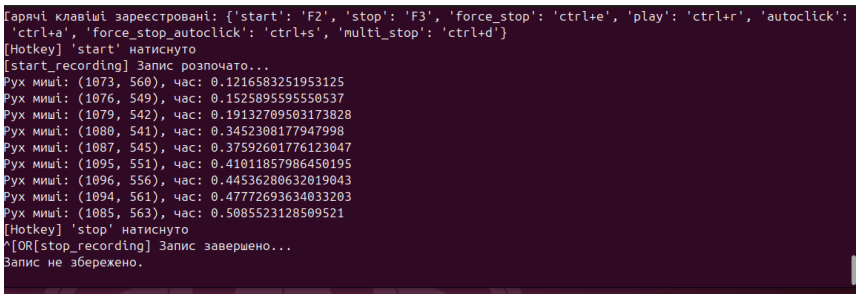

Рисунок 3.7 – Відображення натискання горячої клавіші для старту та зупинки запису в консолі Ubuntu

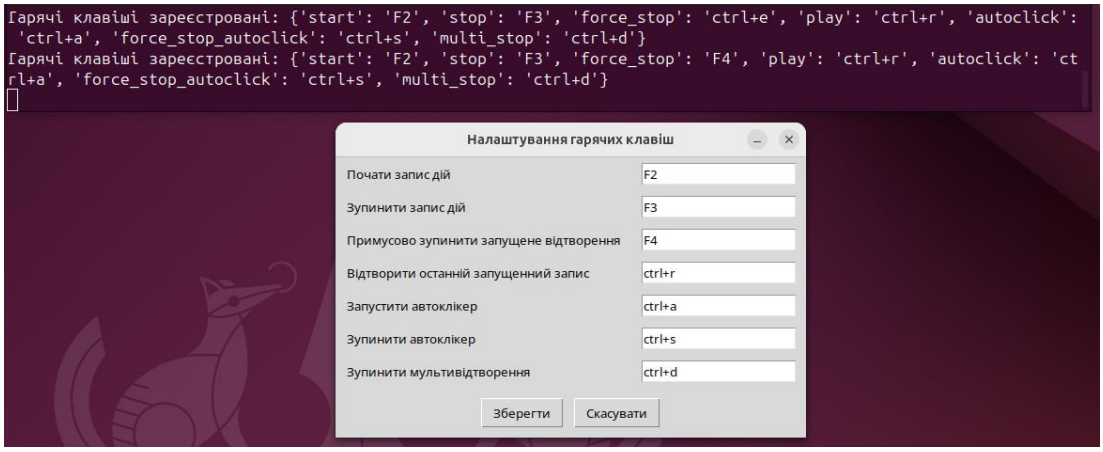

Рисунок 3.8 – Відображення переналаштування горячої клавіші для примусової зупинки запису в вікні додотку та перегляд виконання в консолі Ubuntu

3.2 Запис та відтворення макросів

3.2.1 Призначення функцій запису/відтворення макросів

Функціонал запису та відтворення макросів реалізовано для автоматизації повторюваних дій користувача. За допомогою запису макросу програма фіксує послідовність подій (натискання клавіш, переміщення та натискання кнопок миші) разом із часовими інтервалами між ними, а функція відтворення дозволяє відтворити цю послідовність з оригінальною затримкою, імітуючи дії користувача. Такий підхід економить час при виконанні рутинних задач, оскільки одну і ту ж послідовність дій можна зберегти як «макрос» і запускати її в будь–який момент.

3.2.2 Реєстрація подій миші/клавіатури та запуск/зупинка макросу

Початок запису макросу виконується методом start_recording(), який активує низькорівневі перехоплення подій миші та клавіатури. Зокрема, під час запису створюються обробники self.record_mouse_click,

self.record_mouse_move (через pynput.mouse.Listener) для миші та self.record_key_event (через keyboard.hook) для клавіатури. Ці методи викликаються на кожну подію і додають відповідні записи до списку self.actions. Наприклад, метод обробки натискання миші фіксує координати і кнопку:

```
if pressed:
    self.actions.append(('mouse_down', current_time, x, y,
button.name))
else:
    self.actions.append(('mouse_up',   current_time, x, y,
button.name))
```

Тут current_time – затримка від початку запису. Аналогічно, рух миші фіксується як подія виду ('move', час, x, y) при переміщеннях з інтервалом не менше 30 мс між точками. Запис подій клавіатури здійснюється методом record_key_event; для події натискання клавіші додається запис ('key_down', ts, scan_code, layout, key_name), а для відпускання – ('key_up', ts, scan_code, layout, key_name).

Окрім цього, при кожній події клавіатури перевіряється поточна розкладка клавіатури: якщо вона відрізняється від попередньої, то оновлюється self.current_layout і фіксується подія зміни розкладки (виводиться в лог). На рис. 3.9 показано запис невеличкого макросу з рухами/кліками миші та написанням тексту в Windows, а на рис. 3.10 для Linux.

Запис макросу можна ініціювати як через графічний інтерфейс, так і через гарячі клавіші. У головному вікні програми є кнопки «Почати запис дій» та «Зупинити запис дій» (рис.3.11–3.12), які пов’язані із методами recorder.start_recording() та recorder.stop_recording() відповідно.

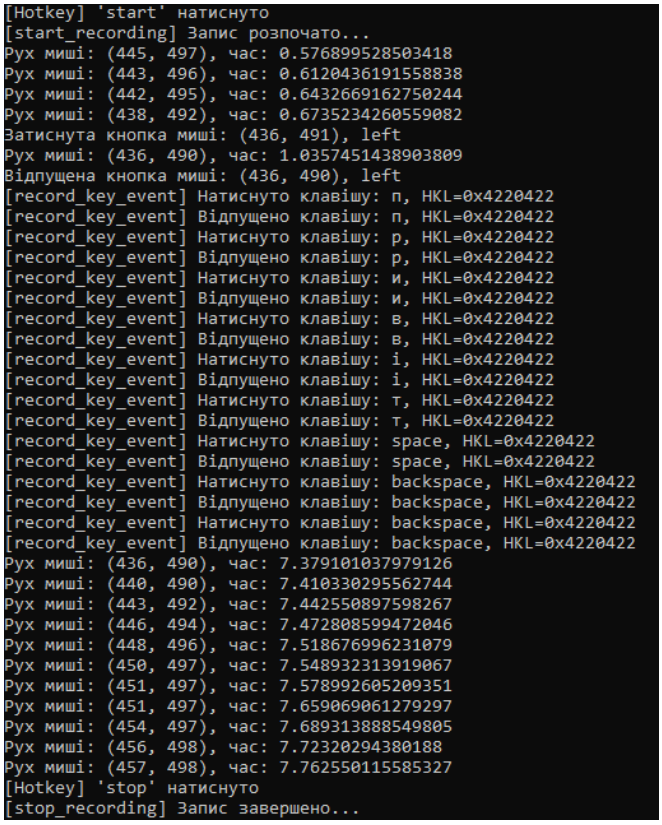

Рисунок 3.9 – Відображення запису подій у консолі Windows

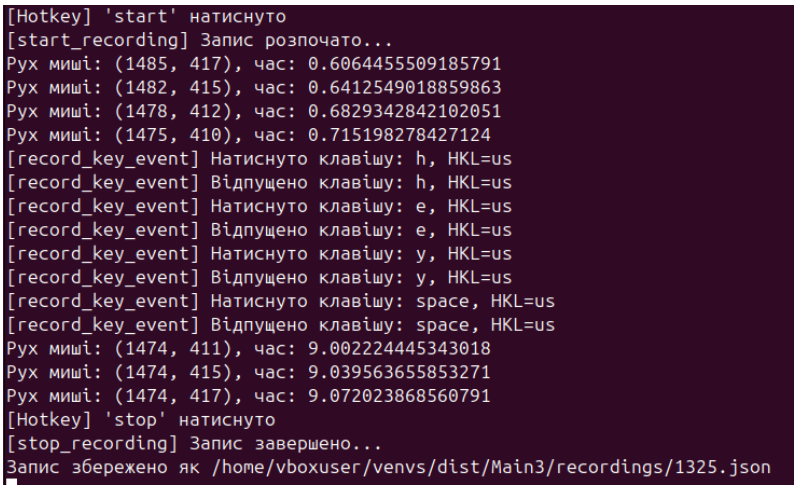

Рисунок 3.10 – Відображення запису подій у консолі Linux

Використовується бібліотека keyboard для призначення гарячих клавіш: наприклад, при натисканні спеціальної комбінації викликається функція on_start(), що викликає recorder.start_recording(), а комбінація stop –

викликає stop_recording(). Таким чином, користувач може почати чи завершити запис як кнопками на екрані, так і зручними комбінаціями клавіш (див. функціонал гарячих клавіш).

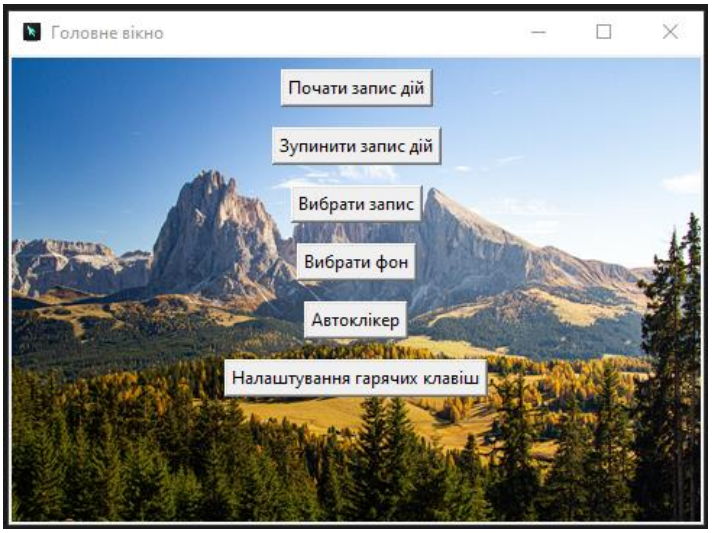

Рисунок 3.11 – Головне вікно додатку (ОС–Windows)

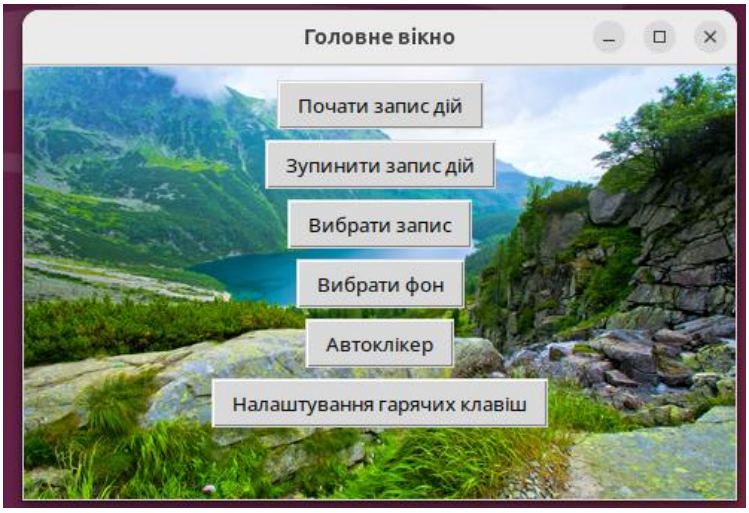

Рисунок 3.12 – Головне вікно додатку (ОС–Linux)

3.2.3 Формат збереження дій

У процесі запису всі події накопичуються у списку self.actions. Кожна дія

зберігається як кортеж з відповідним форматом:

події миші:

– ('mouse_down', ts, x, y, button) – натискання (button = 'left' або 'right') в точці (x,y) у відрізку часу ts секунд від початку запису;

– ('mouse_up', ts, x, y, button) – відпускання кнопки миші.

– ('move', ts, x, y) – переміщення миші до координат (x,y) у момент часу ts.

події клавіатури:

– ('key_down', ts, sc, hkl, key_name) – натискання клавіші з іменем key_name, сканкодом sc та ідентифікатором розкладки hkl у час ts;

– ('key_up', ts, sc, hkl, key_name) – відпускання цієї клавіші.

Після завершення запису користувача просять ввести назву макросу і дані self.actions зберігаються у JSON–файл (див. рис. 3.13–3.14).

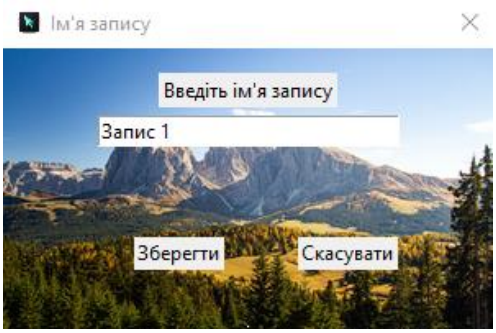

Рисунок 3.13 – Відображення вікна збереження запису (ОС–Windows)

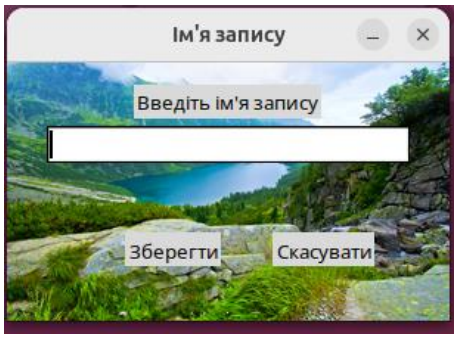

Рисунок 3.14 – Відображення вікна збереження запису (ОС– Linux)

Перед збереженням на початок списку автоматично вставляється особлива подія, що фіксує початкову розкладку клавіатури:

```
if SYSTEM == "Windows":
    hkl = self.initial_layout & 0xFFFFFFFF
else:
    hkl = self.initial_layout
self.actions.insert(0, ('initial_layout', 0.0, hkl))  #
початкова розкладка
```

Це забезпечує збереження інформації про те, якою була розкладка під час запису макросу. Вміст файлу виглядає як масив подій з поля initial_layout, наприклад:

```
[
    [
        "initial_layout",
        0.0,
        69338146
    ],
    [
        "move",
        0.576899528503418,
        445,
        497
    ],
…
```

3.2.4 Відтворення макросу та синхронізація подій

Для відтворення макросу програма читає збережений JSON у списоку self.actions. Якщо перший запис – ('initial_layout',0.0, hkl), він вилучається зі списку, а значення hkl зберігається в self.initial_layout. Далі викликається метод play_actions(), що проходить по кожній дії у списку. Перед початком відтворення спочатку встановлюється початкова розкладка:

```
print(f"[play_actions] Встановлюється initial_layout:
{self.initial_layout}")
```

```
# встановлюється початкова розкладка перед початком відтворення
self.force_set_keyboard_layout(self.initial_layout)

# скидаються всі натиснуті модифікатори (Ctrl, Shift, Alt тощо)
reset_modifiers()

# очищається поточний HKL, щоб при першій операції перевірити і
встановити його за необхідності
self.current_play_hkl = None
```

Після цього програма послідовно імітує кожну подію у правильний момент часу. Для синхронізації використовується затримка: програма чекає, поки з моменту початку відтворення не пройде target_ts секунд (через time.sleep()).

Коли настає час події, вона відтворюється – наприклад, рух миші виконується функцією move_mouse(x,y), натискання/відпускання кнопки миші – викликом відповідних функцій mouse_down_at(x,y) чи mouse_up_at(x,y). Для клавіш перевіряється розкладка: якщо в записі ця розкладка (target_hkl) відрізняється від поточної self.current_play_hkl, перед виконанням імітується переключення:

```
# якщо поточний HKL відрізняється від потрібного
if self.current_play_hkl != target_hkl:
    print(f"[play_actions] Перемикається на HKL: {target_hkl}")
    # примусово встановлюється потрібна розкладка
    self.force_set_keyboard_layout(target_hkl)
    # оновлюється поточний HKL
    self.current_play_hkl = target_hkl
```

Тільки після цього симулюється натискання (press_key(sc)) або відпускання (release_key(sc)) клавіші. Такий підхід гарантує, що між подіями зберігаються реальні часові інтервали, а кожна дія виконується у потрібній послідовності та у відповідний момент.

3.2.5 Детальніше про перемикання розкладки клавіатури

У кросплатформенному автоклікері реалізовано механізм фіксації і відновлення розкладки клавіатури, що критично для коректного введення символів. Оскільки клавіатурна розкладка визначає, як сканкоди мапляться на символи, при відтворенні макросу необхідно встановити ту саму розкладку, що була під час запису. У методі start_recording() зберігається поточна розкладка у змінній self.initial_layout (виклик get_current_keyboard_layout()). Після завершення запису в діалозі збереження ця початкова розкладка кодується у записі через подію ('initial_layout',0.0,hkl).

Метод force_set_keyboard_layout забезпечує примусове перемикання розкладки: він має дві реалізації – для Windows і Linux. На Windows використовується виклик API. У Linux викликається системна команда setxkbmap з потрібним кодом розкладки. Обидва варіанти враховують, що ідентифікатор HKL (або його скорочений формат) може відрізнятися за довжиною. Наприклад:

```
# Windows: отримується 16-бітний KLID та завантажується
розкладка через API
klid     = hkl & 0xFFFF  # виділяється 16 біт з HKL
klid_str = f"{klid:08x}" # форматується як 8-значний
шістнадцятковий рядок
# завантажується KLID у процес
loaded_hkl = LoadKeyboardLayoutW(klid_str, KLF_SETFORPROCESS)
if not loaded_hkl:
    print(f"[force] Не вдалося
LoadKeyboardLayoutW('{klid_str}')")
# повідомлення в активне вікно про зміну розкладки
PostMessageW(hwnd, WM_INPUTLANGCHANGEREQUEST, 0, loaded_hkl)


# Linux: отримується LANGID і змінюється розкладка через
setxkbmap
langid = hkl & 0xFFFF # виділяються низькі 16 біт з HKL
layout = HKL_TO_LAYOUT.get(langid # пошук відповідного коду
розкладки
if not layout:
```

```
    print(f"[force] Невідома розкладка LANGID=0x{langid:04X}")
else:
    # виконується системна команда, щоб одразу перемкнути
розкладку
subprocess.check_call(['setxkbmap', layout],
stderr=subprocess.DEVNULL)
```

При відтворенні макросу, перед обробкою кожної клавіші код перевіряє: якщо поточна розкладка відрізняється від тієї, що фіксувалася у записі події (target_hkl), викликається force_set_keyboard_layout(target_hkl). Це гарантує, що навіть якщо користувач змінив розкладку між записом і запуском макросу, введені символи залишаться правильними.

Деякі з блоків коду реалізації. Наприклад, вставка початкової розкладки до подій запису (функція ask_to_save) виглядає так:

```
if SYSTEM == "Windows":
    # для Windows 32-бітний формат HKL
    hkl = self.initial_layout & 0xFFFFFFFF
else:
    # для Linux використовується значення initial_layout без
змін
    hkl = self.initial_layout

# вставляється на початок списку дій подію встановлення
розкладки
# формат: (тип_дії, час_від_початку, HKL)
self.actions.insert(0, ('initial_layout', 0.0, hkl))
```

А у методі відтворення (play_actions) реалізовано перевірку і перемикання розкладки для кожної клавіші:

```
# якщо поточна HKL відрізняється від потрібної – перемикається
розкладка
if self.current_play_hkl != target_hkl:
    self.force_set_keyboard_layout(target_hkl) # примусово
встановлюється розкладка
    self.current_play_hkl = target_hkl # оновлюється поточний
HKL

# виконується натискання або відпускання клавіші в залежності
від типу події
if typ == 'key_down':
```

```
    press_key(sc)    # натиснути клавішу з scancode sc
else:
    release_key(sc)  # відпустити клавішу з scancode sc
```

Таким чином, описаний механізм забезпечує правильну фіксацію і відтворення послідовності дій з урахуванням часових затримок і необхідної розкладки клавіатури.

3.2.6 Циклічне повторення макросів та їх зупинка

Для організації багаторазового повторення макросу в коді використовуються флаги і лічильники циклів. Наприклад, у функції start_playback_loop() зчитуються налаштування запису (record_settings), зокрема параметр repeats – кількість повторів макросу (значення 0 відповідає нескінченному повторенню). Фактичну кількість уже виконаних ітерацій відстежує змінна self.playback_count. Перед запуском нового циклу перевіряється, чи вже не йде відтворення (self.is_playing). Якщо попередній цикл ще виконується, він зупиняється через встановлення події зупинки:

```
if self.playback_thread and self.playback_thread.is_alive():
    self.stop_event.set()  # сигналізує циклу про зупинку
    self.playback_thread.join() # чекає завершення фонового
потоку
```

Після цього ініціалізуються прапорці is_playing = True і is_looping = True, а також створюється або очищується об’єкт self.stop_event = threading.Event().

Усередині вкладеної функції _loop() організовується цикл повторення макросу: лічильник count збільшується на кожній ітерації, викликається self.play_actions(), і доки self.is_looping == True і не досягнуто потрібної кількості повторів, цикл продовжується. Таким чином, параметри total_repeats (максимум повторів із конфігурації) та self.playback_count керують кількістю циклів відтворення макросу.

Між ітераціями виконується пауза, що задається через параметр interval (наприклад, interval = cfg.get('interval', 1) – затримка в секундах). Після завершення кожного циклу виконується time.sleep(interval), що задає проміжок між послідовностями дій. Таким чином, щоб зробити затримку між повторами, використовується стандартний засіб time.sleep() з потрібним часом очікування.

Запуск циклічного відтворення макросу здійснюється у фоновому потоці, аби не блокувати інтерфейс програми. У коді це реалізовано методом start_playback_loop(), який створює новий потік з цільовою функцією _loop та запускає його. Наприклад:

```
def start_playback_loop(self):
    # ... підготовка параметрів (is_playing, параметри
repeats/interval, stop_event тощо) ...
    def _loop():
        count = 0
        while self.is_looping and (total_repeats == 0 or count <
total_repeats):
            count += 1
            ok = self.play_actions()  # виконується записані дії
            if not ok:
                break
            if total_repeats and count >= total_repeats:
                break
            # завершилась одна ітерація – зупинка перед повтором
            self.force_stop_playback()
            time.sleep(interval)
            self.simulate_play_hotkey()
            return
        self.finalize_playback()
    # створення і запуск фонового потіку для _loop
    self.playback_thread = threading.Thread(target=_loop,
daemon=True)
    self.playback_thread.start()   # запуск виконання run() у
окремому потокі
```

Як видно, виклик self.playback_thread.start() організовує виклик функції _loop() в окремому (фоному) потоці управління, що дозволяє головному потоку обробляти події UI.

Зупинка відтворення макросу може відбуватися як автоматично після

досягнення заданої кількості повторів, так і вручну – за допомогою флагу–події self.stop_event, гарячих клавіш або кнопок інтерфейсу. threading.Event застосовується тут для безпечної (thread–safe) передачі сигналу зупинки між потоками. Наприклад, якщо користувач натискає гарячу клавішу зупинки (у коді використовується комбінування клавіш, які викликають метод force_stop_playback()), то в цьому методі виконується:

```
self.is_playing = False
self.is_looping = False
if self.stop_event:
    self.stop_event.set()  # сигнал на зупинку фону
```

Таким чином, встановлюється подія stop_event, на яку регулярно перевіряють цикл відтворення і сама функція play_actions(). У ній у кожному кроці ітерування списку дій перед виконанням команди перевіряється:

```
for idx, act in enumerate(self.actions):
    # перевірка, чи було запитано зупинку перед виконанням цієї
дії
    if self.stop_event.is_set():
        print(f"[play_actions] Перервано на події #{idx}")
        return False

    target_ts = act[1]  # час запуску цієї дії від початку
відтворення

    while True:
        elapsed = time.time() - start_time  # пройшло часу з
початку відтворення
        if elapsed >= target_ts:
            # час виконати дію
            break

        # перевірка запиту на зупинку перед самою подією
        if self.stop_event.is_set():
            print(f"[play_actions] Перервано перед подією
#{idx}")
            return False

        time.sleep(min_sleep)  # невелика пауза
```

Якщо stop_event встановлено, цикл одразу переривається і

play_actions() повертає False, що призводить до виходу з ітераційного повтору. Така перевірка в циклі гарантує негайне завершення макросу при натисненні «Стоп».

Важливо розуміти, що в Python потік не може бути примусово вбитий іншим потоком (немає Java–подібних функцій stop тощо). Тому саме механізм сигналізування через threading.Event дозволяє безпечно зупиняти виконання циклу. Після завершення циклу або виклику зупинки (force_stop_playback чи кнопки) виконується finalize_playback(), що також встановлює stop_event і звільняє ресурси (звільнення натиснутих клавіш, з'єднання з GUI тощо).

Підсумок: код циклічного відтворення макросу реалізовано через фоновий цикл _loop(), що викликає play_actions() повторно у межах заданих налаштувань repeats і interval. Затримка між циклами задається через time.sleep(interval). Фоновий потік створюється за допомогою threading.Thread(...) і запускається викликом start(). Примусова зупинка робиться через сигнал self.stop_event, який перевіряється в кожному кроці виконання дій. Натискання гарячих клавіш або кнопок інтерфейсу викликає метод force_stop_playback(), який встановлює цей прапорець. На рис. 3.15–3.16 показано відтворення запису з проходженням двух повторів й примусовою зупинкою для двух ОС.

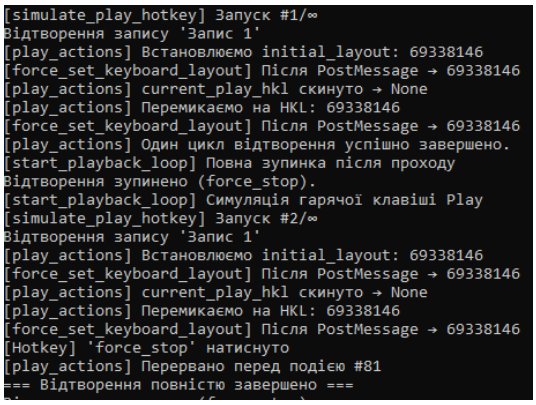

Рисунок 3.15 – Відображення відтворення запису в консолі з показом примусової зупинки (ОС– Windows)

Рисунок 3.16 – Відображення відтворення запису в консолі з показом примусової зупинки (ОС– Linux)

3.3 Збереження та керування макросами

Після завершення запису макросів важливо зберегти їх у файл для подальшого використання. Збереження макросів дозволяє користувачу повторно запускати відпрацьовані послідовності дій без потреби запису знов. Крім того, керування макросами забезпечує можливість вибору потрібного макросу зі списку, видалення застарілих або небажаних записів та налаштування параметрів повтору (інтервалу та кількості циклів). Таке рішення економить час і полегшує організацію великих обсягів макросів:

– повторне використання: збережені макроси можна запускати щоразу без повторного запису.

– управління списком: користувач може вибирати потрібний макрос із наявних, видаляти непотрібні.

– налаштування параметрів: кожному макросу можна задати кількість повторів і час між ними.

Підсистема збереження реалізована за допомогою формату JSON. Після завершення запису викликається функція ask_to_save(), яка відкриває вікно введення імені запису. Користувач вводить ім’я, і програма створює файл із цим ім’ям у спеціальній папці recordings. При цьому використовується коректне формування шляху через os.path.join, що гарантує роботу на різних ОС.

У список self.actions на початок вставляється кортеж ('initial_layout', 0.0, hkl), щоб зберегти початкову розкладку під час запису макросу. Потім весь список подій записується у JSON–файл через json.dump, що робить формат збереження зрозумілим і кросплатформеним (з урахуванням кодування UTF–8 та відступів для читабельності). Таким чином, кожний макрос зберігається як окремий файл *.json у каталозі recordings, який створюється за допомогою os.makedirs(self.recordings_dir, exist_ok=True) при запуску.

Після збереження програма оновлює внутрішній список збережених макросів – структура self.saved_recordings містить кортежі (ім’я, шлях). Її наповнення відбувається у методі load_saved_recordings(), який перебирає файли в папці recordings і додає кожен JSON–файл до списку:

```
def load_saved_recordings(self):
    self.saved_recordings.clear() # очищення попереднього списку
записів
    for fname in os.listdir(self.recordings_dir):
        # вибір лише файлів з розширенням .json
        if fname.lower().endswith(".json"):
            name = os.path.splitext(fname)[0] # ім’я без
розширення
            path = os.path.join(self.recordings_dir, fname)  #
повний шлях до файлу
            self.saved_recordings.append((name, path)) #
додавання до списку кортеж (ім’я, шлях)
```

При відтворенні обраного макросу відповідний файл читається за допомогою json.load. Наприклад, у функції on_play() після вибору імені та шляху виконується:

```
# завантаження списку дій з файлу JSON у self.actions
with open(filepath, "r", encoding="utf-8") as f:
    self.actions = json.load(f)

# якщо перша дія – це установка початкової розкладки, зберегти
її і видалити з черги
if self.actions and self.actions[0][0] == 'initial_layout':
    self.initial_layout = self.actions[0][2] # зберегти значення
розкладки
    self.actions.pop(0) # видалити цю дію з масиву

# запуск циклу відтворення подій
self.start_playback_loop()
```

Після завантаження списку подій (self.actions) програма встановлює збережену початкову розкладку клавіатури (якщо вона була збережена) і запускає цикл відтворення макросу. У такий спосіб реалізовано завантаження та програвання макросів з файлів.

Інтерфейс керування макросами представлено у вигляді окремого вікна з переліком записів та кнопками для операцій над ними. Кнопка “Вибрати запис” у головному вікні викликає метод show_saved_recordings(), який створює новий Toplevel із заголовком “Вибрати запис”. У цьому вікні міститься Listbox зі списком імен доступних макросів і три кнопки: “Видалити запис”, “Відтворити запис” та “Налаштування повторів” (рис. 3.17).

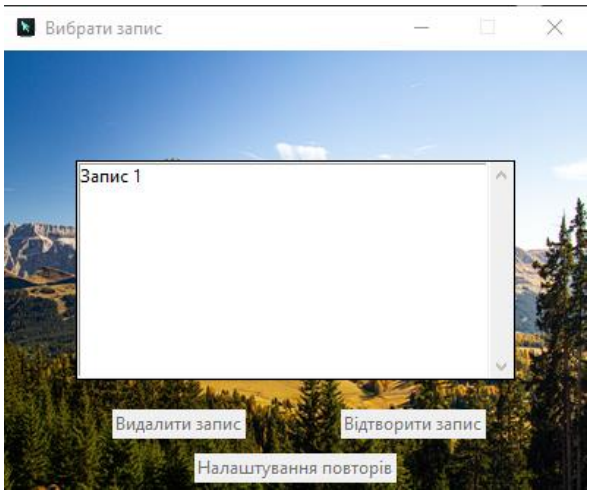

Рисунок 3.17 – Відображення вікна вибора запису (ОС– Windows)

При першому відкритті кнопки неактивні; натискання на елемент списку (<<ListboxSelect>>) вмикає відповідні кнопки (див. код нижче), на рис. 3.18–3.19 показано вікно при виборі запису.

```
# створення Listbox без активного елемента
lb = tk.Listbox(inner, activestyle="none")

# заповнення Listbox іменами збережених записів
for name, _ in self.saved_recordings:
    lb.insert("end", name)

# Функція-обробник вибору в Listbox
def on_list_select(event=None):
    sel = lb.curselection() # отримання індексу обраних
елементів
    state = "normal" if sel else "disabled" # активація
кнопки, якщо щось вибрано
    delete_btn.config(state=state) # перемикання стану
кнопки «Видалити»
    play_btn.config(state=state) # перемикання стану кнопки
«Відтворити»
    settings_btn.config(state=state) # перемикання стану
кнопки «Налаштування»

# додавання обробника до події вибору в Listbox
lb.bind("<<ListboxSelect>>", on_list_select)
```

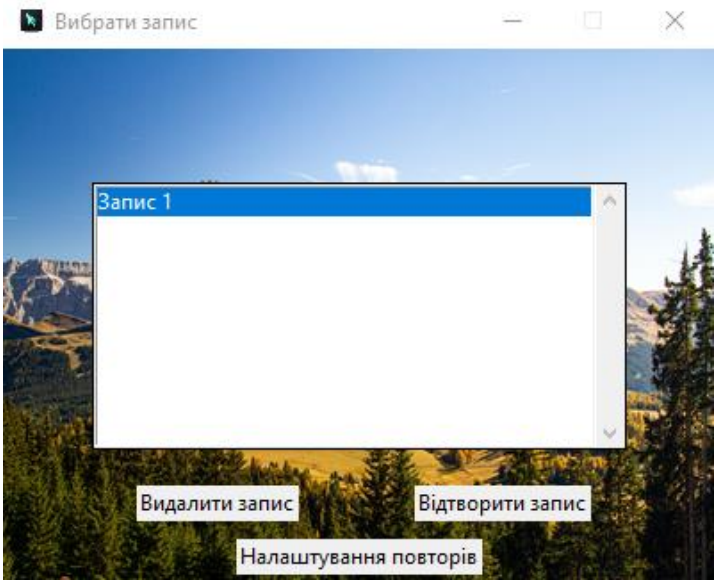

Рисунок 3.18 – Відображення вікна з обраним записом (ОС– Windows)

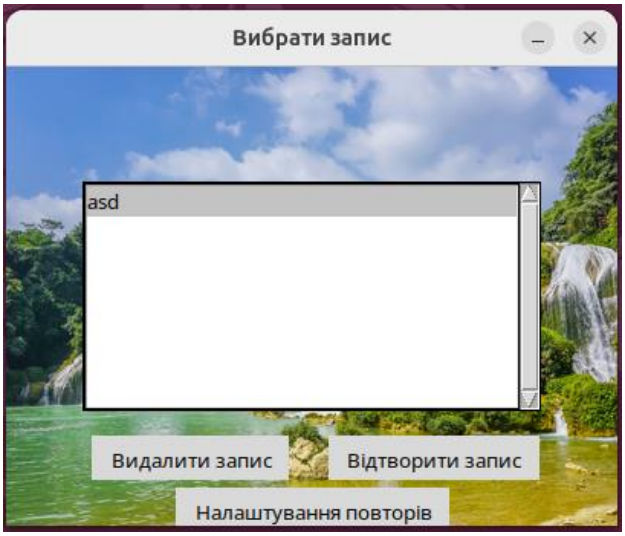

Рисунок 3.19 – Відображення вікна з обраним записом (ОС– Linux)

Натискання “Видалити запис” виконує функцію on_delete(), яка забирає виділений макрос зі списку та видаляє файл з диску:

```
def on_delete():
    idx = lb.curselection()[0] # отримання індексу обраного
запису
    name, filepath = self.saved_recordings.pop(idx) # видаляє
запис з внутрішнього списку
    os.remove(filepath) # видаляє файл з диску
    lb.delete(idx) # видаляє запис із Listbox
```

Після видалення пункт у списку зникає і клавіші знову вимикаються. Кнопка “Відтворити запис” викликає функцію on_play() і починає відтворення макросу, а “Налаштування повторів” відкриває ще один діалог для введення інтервалу та кількості повторів. У цьому діалозі Toplevel користувач задає параметри, які зберігаються у файлі repeat_settings.json методом save_record_settings():

```
dlg = tk.Toplevel(win)  # створення вікна налаштувань

# змінні для інтервалу й кількості повторів із початковими
значеннями
interval_var = tk.DoubleVar(value=interval)
```

```
repeats_var  = tk.IntVar(value=repeats)
def save_and_close():
    # забезпечення невід'ємних значень
    iv = max(0.0, float(interval_var.get()))
    rv = max(0, int(repeats_var.get()))

# зберігання налаштування для поточного файлу
    self.record_settings[filepath] = {'interval': iv, 'repeats':
rv}
    self.save_record_settings() # записує налаштування на диск
    dlg.destroy() # закриває діалогове вікно
```

Діалог налаштувань повторів дозволяє зберегти індивідуальні параметри циклів виконання кожного макросу. Після збереження налаштування зберігаються у структурі self.record_settings і записуються у JSON–файл repeat_settings.json у папці програми. На рис. 3.20 показано вікно налаштування повторів, а на рис. 3.21 – вивід в консоль повідомлення про збереження змін для цього запису.

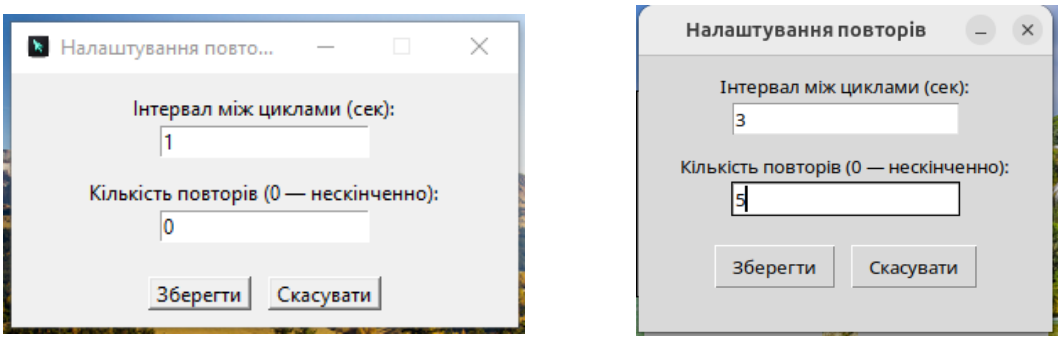

Рисунок 3.20 – Відображення вікна налаштування повторів для Windows та Linux

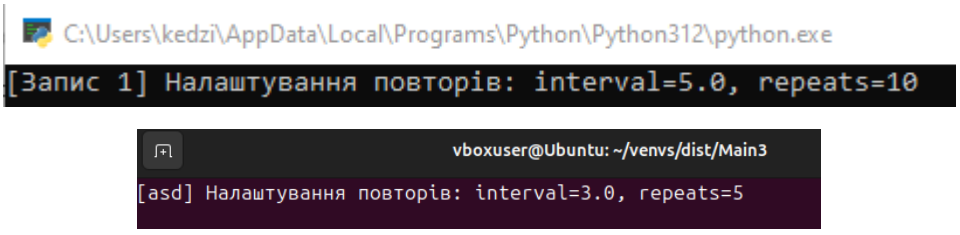

Рисунок 3.21 – Відображення збереження налаштування в консолі Windows та Linux

Функція get_app_dir() визначає кореневу директорію програми як для Windows, так і для Linux/Mac, після чого до неї додаються підкаталоги за допомогою os.path.join. Це забезпечує правильне формування шляхів у будь–якій ОС. Крім того, при збереженні початкової розкладки перевіряється ОС (if SYSTEM == "Windows") й в інших частинах коду, для реалізації різноманітних доповнень під кожну ОС. У діалогах GUI, наприклад при налаштуванні повторів, розміри вікон задаватися по–різному для Windows і інших систем. Завдяки таким рішенням збереження та завантаження макросів стабільно працюють незалежно від платформи.

3.4 Робота з зображеннями

Підтримка фонових зображень реалізована для покращення зручності використання програми (UX) та персоналізації інтерфейсу. Користувач може обирати улюблені зображення як фон, що робить програму привабливішою та допомагає виділити її серед інших подібних за однаковою функціональністю.

Вікно вибору фону (open_background_window) – відкриває нове підвікно із Canvas та смугою прокрутки, у якому завантажуються ескізи всіх зображень із папки background. Функція load_images() перебирає файли у цій папці і для зображень з розширеннями PNG, JPG, GIF, BMP тощо створює об’єкти PIL.Image. Кожне зображення зменшується до ескізу розміром 200×200 пікселів через метод resize(..., Image.LANCZOS) у функції get_thumbnail(). Ескізи відображаються на Canvas із встановленими координатами за допомогою canvas.create_image.

Користувач може клацнути по одному ескізі або створити рамку виділення перетягуванням миші (drag), щоб обрати один чи декілька варіантів (див. рис. 3.22). При виборі окремого зображення воно підсвічується синьою рамкою і викликається діалог налаштування фону.

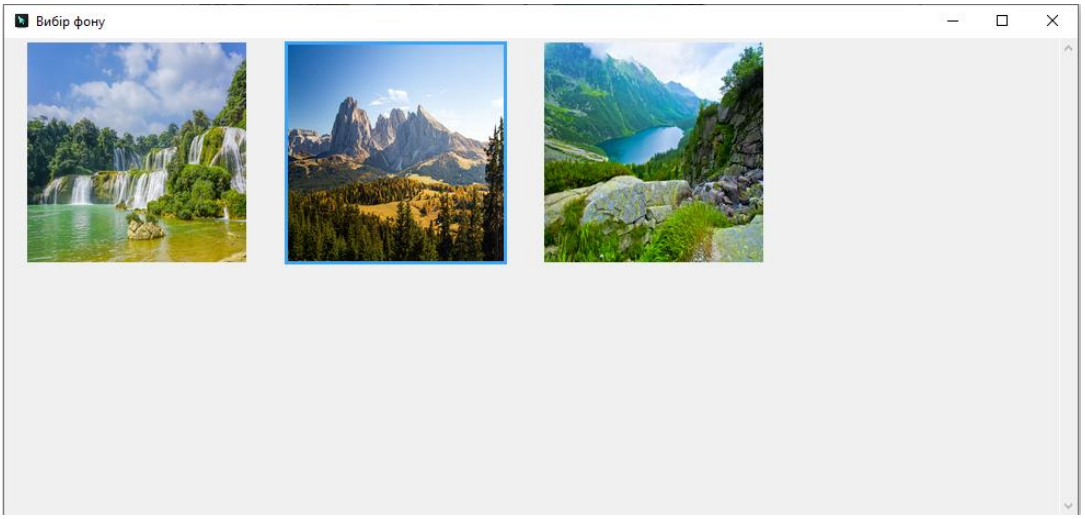

Рисунок 3.22 – Відображення вікна вибору фону та виділення зображеня

```
# завантаження зображень з папки background
def load_images():
    # отримує шлях до папки з фоновими зображеннями
    image_folder = resource_path("background")
    images = []  # список для завантажених зображень

    for file in os.listdir(image_folder):
        # відбір файліф з потрібними розширеннями
        if file.lower().endswith(('.png', '.jpg', '.jpeg',
'.gif', '.bmp')):
            img_path = os.path.join(image_folder, file) # повний
шлях до файлу
            img = Image.open(img_path).convert("RGBA") #
відкриває й конвертуємо в RGBA
            img.filename = img_path # зберігає шлях у атрибуті
filename
            images.append(img) # додає об’єкт до списку

    return images  # Повертаємо список завантажених зображень

# створення ескізу за допомогою Pillow
def get_thumbnail(img, size=(200, 200)):
    thumbnail = img.copy().resize(size, Image.LANCZOS) #
копиювання оригіналу та зміна розміру з високою якістю
    tk_img = ImageTk.PhotoImage(thumbnail) # перетворення
Pillow-об’єкт у формат, придатний для Tkinter
    return tk_img # повернення об’єкту PhotoImage для
відображення у віджеті

# додавання зображення на Canvas з прив’язкою верхнього лівого
кута
image_id = canvas.create_image(x_pos, y_pos, anchor="nw",
```

```
image=tk_img)

# зберігання посилання на PhotoImage, щоб Tkinter не звільнив
його з пам’яті
canvas.image_refs.append(tk_img)

# додавання ідентифікатору зображення до списку для подальшої
обробки
canvas_image_ids.append(image_id)

# зв’язує id на Canvas з оригінальним Pillow-об’єктом
image_gallery[image_id] = img
```

Діалог налаштування фону (create_select_background_window) – це друге підвікно, яке відкривається після вибору зображення. У ньому можна підтвердити застосування фону або видалити його (див. рис. 3.23). Якщо передати у функцію обране зображення, то воно записується як current_background_image, викликається save_last_background() для збереження імені останнього фону у файл, а також виконується update_background() для встановлення фону в цьому діалозі та update_main_window_background() для основного вікна.

Крім того, у діалозі є кнопки “Видалити фон” (викликає delete_background()) і “Завантажити з пристрою”, яка відкриває стандартний filedialog. При виборі файлів через діалог вони копіюються в папку background функцією shutil.copy(), після чого викликається refresh_background_window() для оновлення списку ескізів.

```
# завантаження нових зображень з диску у папку background
file_paths = filedialog.askopenfilenames(
    title="Виберіть зображення",
    filetypes=[("Зображення", "*.png *.jpg *.jpeg *.gif *.bmp
*.tiff"), ("Усі файли", "*.*")]
)
if not file_paths:
    return  # немає вибраних файлів – вихід

for fp in file_paths:
    shutil.copy(fp, backgrounds_dir)  # копіює кожен вибраний
файл у папку з фонами
refresh_background_window() #оновлює відображення вікна фонів
після додавання нових зображень
```

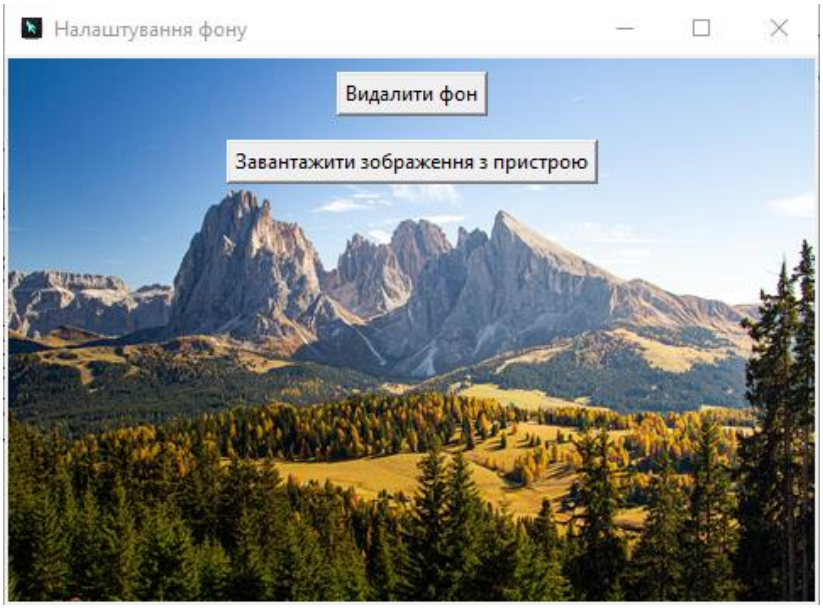

Рисунок 3.23 – Діалогове вікно налаштування фону (з опціями видалення/додавання зображень)

Збереження останнього обраного фону – реалізується через змінні LAST_BG_FILE та функції save_last_background(image) і load_last_background(). Після вибору фону save_last_background() записує ім’я файлу фону у текстовий файл (використовуючи os.path.basename), щоб при наступному запуску програми load_last_background() заздалегідь завантажив цей фон. Це дозволяє зберегти вибір користувача між сесіями (див. рис. 3.24). Наприклад:

```
def save_last_background(image):
    # отримання ім’я файлу без шляху
    base = os.path.basename(image.filename)
    # запис ім’я останнього фону у файл для наступного запуску
    with open(LAST_BG_FILE, "w", encoding="utf-8") as f:
        f.write(base)
```

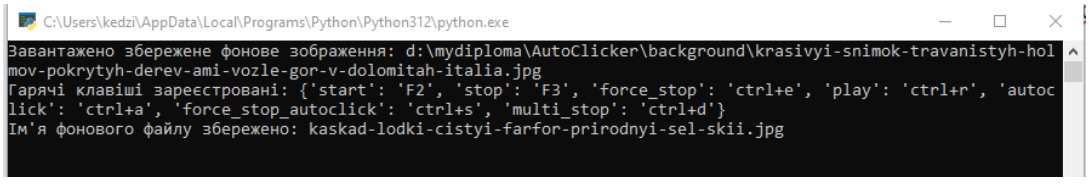

Рисунок 3.24 – Відображення в консолі повідомлення про зміну фону

Обрані зображення на Canvas (виділення) – при перетягуванні курсора мишею формується рамка (створюється тимчасове зображення напівпрозорого прямокутника і синя рамка). Кожен ескіз, який повністю або частково потрапив у цю рамку, відзначається (див. рис. 3.25). Це реалізовано в обробнику on_canvas_mouse_drag(): для кожного image_id перевіряється, чи перетинаються області, і якщо так – малюється прямокутник з тегом SELECTED_TAG.

```
# перевірка, чи прямокутник виділення (ix1,iy1)-(ix2,iy2)
перетинається з областю зображення (x1,y1)-(x2,y2)
if not (ix2 < x1 or ix1 > x2 or iy2 < y1 or iy1 > y2):
    # якщо так, малює рамку навколо зображення для підсвічування
    rect = canvas.create_rectangle(
        ix1, iy1, ix2, iy2,
        outline="#3ba4f5",     # колір контуру
        width=3,               # товщина лінії
        tags=SELECTED_TAG    # тег для простого очищення
підсвічувань пізніше
    )
    # зберігання ID прямокутника, щоб при потребі можна було
його видалити
    highlight_rects[img_id] = rect
    # додавання ідентифікатору зображення до списку для
видалення
    to_delete_images.append(img_id)
```

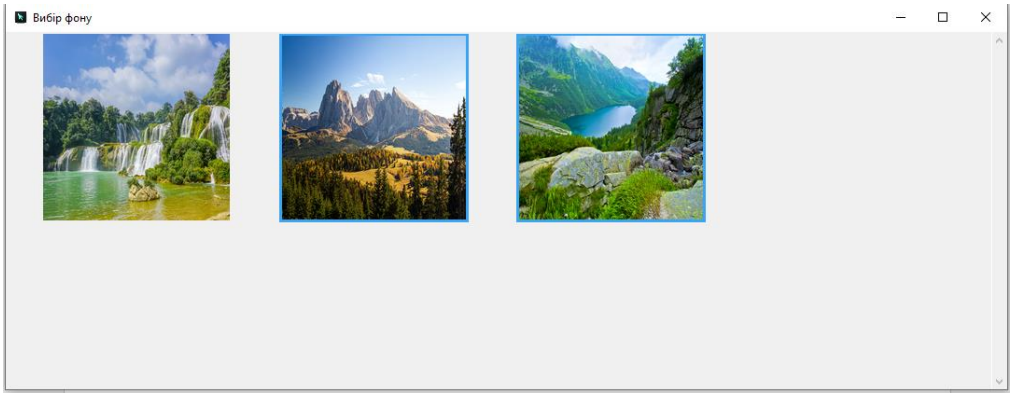

Рисунок 3.25 – Вікно вибору фону з виділенням декількох зображень

Видалення фонових зображень – функція delete_background() видаляє вибрані користувачем файли з папки background і оновлює інтерфейс. Якщо

користувач обрав декілька ескізів, вони видаляються у циклі (див. рис. 3.26). При цьому очищаються виділення та оновлюються вікна: викликаються update_background(), refresh_background_window() і update_main_window_background(). Наприклад:

```
if to_delete_images:
    # Видаляємо файли обраних зображень із диску
    for img_id in to_delete_images:
        file_path = image_gallery[img_id].filename
        os.remove(file_path)
    to_delete_images.clear() # Очищаємо список вибраних для
видалення

    # Прибираємо всі прямокутники підсвітки з Canvas
    for rect_id in highlight_rects.values():
        canvas.delete(rect_id)
    highlight_rects.clear() # Скидаємо мапу підсвіток

    update_background() # Оновлюємо дані фонів у пам'яті
    refresh_background_window() # Оновлюємо вікно перегляду
фонів
    update_main_window_background() # Оновлюємо фон головного
вікна
```

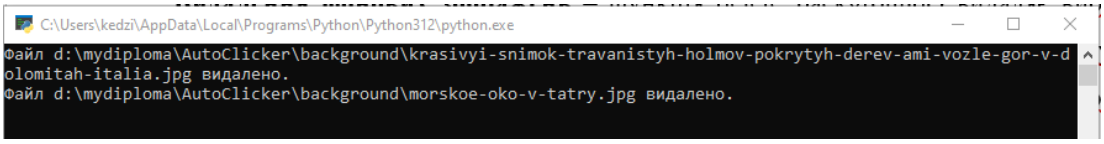

Рисунок 3. 26 – Відображення повідомлення в консоль після видалення 2 виділених зображень

Застосування фону до декількох вікон – вибране фонового зображення автоматично встановлюється не лише в головному вікні, а й в інших діалогах (наприклад, у вікні збереження запису, автоклікері тощо). Це забезпечує функція update_main_window_background(), що перевіряє current_background_image і змінює зображення Canvas головного вікна при зміні розмірів або натисканні відповідних кнопок.

Використання зображень у мультиклікері – окрім фонів, в інтерфейсі

програми використовуються різноманітні іконки (курсор, кнопка мінус, папка, пуск, стоп). У конструкторі класу Recorder ці файли (mouse.png, minus.png, folder.png, triangle.png, stop.png) завантажуються через Pillow, змінюються розміром і передаються в ImageTk.PhotoImage. Наприклад:

```
mouse_img =
Image.open(resource_path("icons/mouse.png")).resize((25, 30),
Image.LANCZOS)
self.img_mouse = ImageTk.PhotoImage(mouse_img)
minus_img =
Image.open(resource_path("icons/minus.png")).resize((25, 30),
Image.LANCZOS)
self.img_minus = ImageTk.PhotoImage(minus_img)
# ... аналогічно для folder.png, triangle.png, stop.png ...
```

Таким чином, кнопки інтерфейсу мають зрозумілі позначки, що покращує сприйняття програми.

Кросплатформенність та формати – завантаження і обробка зображень забезпечується бібліотекою Pillow, яка підтримує формати PNG, JPG, BMP тощо. Користувач може додавати будь–які з цих форматів (як видно з фільтрів у filedialog). Універсальна логіка побудови шляхів (функція resource_path, що використовує os.path.join) гарантує, що програма працюватиме однаково на Windows і Linux.

Використання фонів підвищує привабливість програми та зручність її використання: надається можливість персоналізації інтерфейсу залежно від вподобань користувача. Це робить програму більш привабливою, а візуальна складова позитивно впливає на вибір користувача.

3.5 Автоклік та мультиклік

Усі можливості автокліку та мультикліку реалізовано у вікні з назвою «Автоклік та мультиклік» (див. рис. 3.27–3.28). Це вікно відкривається з головного інтерфейсу програми, натисканням кнопки «Автоклікер» в

головному вікні.

3.5.1 Елементи керування

Вікно містить наступні елементи управління: кнопки «Налаштування автокліка курсором», «Запустити автоклік курсором», «Зупинити автоклік курсором», «Створити мультиклік», «Вибрати запис мультикліку» та «Інструкція до функції створення мультикліку». Кожна кнопка прив’язана до відповідного методу: наприклад, btn_start з текстом «Запустити автоклік курсором» викликає self.start_autoclick(), а btn_multi – self.open_multiwindow().

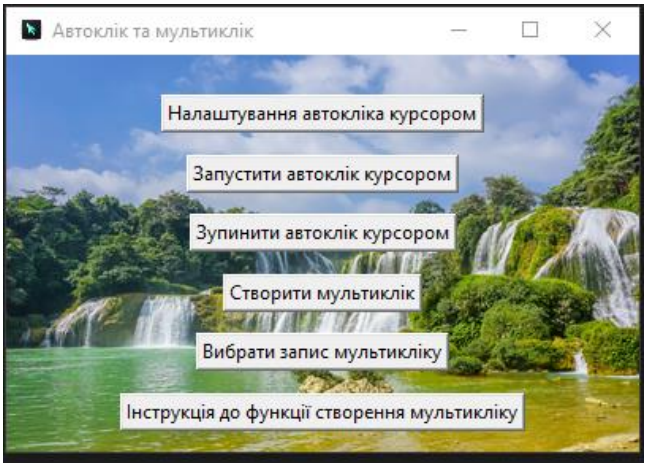

Рисунок 3.27 – Вікно автокліку (ОС – Windows)

Запуск і зупинка автокліку. Вікно «Автоклік та мультиклік» містить кнопки «Запустити автоклік курсором» і «Зупинити автоклік курсором» які викликають відповідні методи. Метод start_autoclick() запускає фоновий цикл, який регулярно отримує поточні координати курсору (get_cursor_pos()) і емулює клік по ним у нескінченному циклі з інтервалом self.auto_click_interval який спочатку встановлено в 1 клік конжні 500 мс, користувач може змінити це значення в вікні «Налаштування автокліка

курсором» (див. рис. 3.31–3.32).

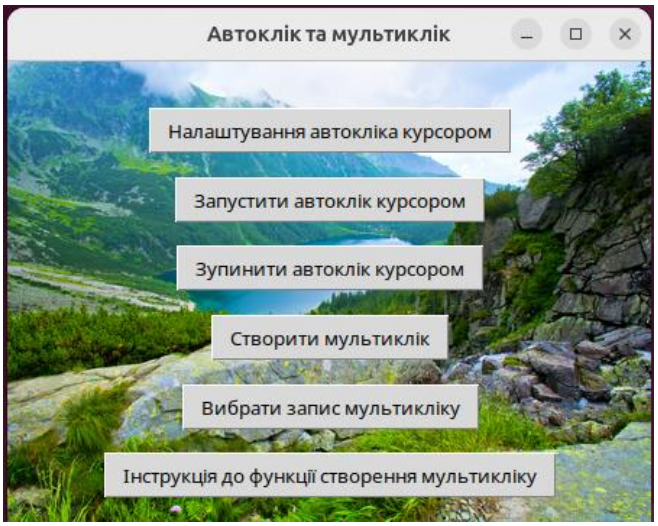

Рисунок 3.28 – Вікно автокліку (ОС – Linux)

Приклад коду запуску кліків:

```
def start_autoclick(self):
    if self.auto_clicking:
        return  # Якщо автоклік вже активний – нічого не 
робиться

    self.auto_click_event.clear() # скидає прапорець зупинки 
автокліку
    self.auto_clicking = True  # встановлює прапорець, що 
автоклік запущено

    def _loop():
        try:
            # виконує кліки доти, доки не буде встановлено подію 
зупинки
            while not self.auto_click_event.is_set():
                x, y = get_cursor_pos() # отримання поточних 
координат курсору
                mouse_down_at(x, y) # емуляція натискання кнопки 
миші
                mouse_up_at(x, y) # емуляція відпускання кнопки 
миші
                time.sleep(self.auto_click_interval) # затримка 
між кліками
        finally:
            self.auto_clicking = False # скидання прапорця, коли
```

```
цикл завершується

    # створення та запуск фонового потоку для автокліку
    self.auto_click_thread = threading.Thread(target=_loop,
daemon=True)
    self.auto_click_thread.start()
```

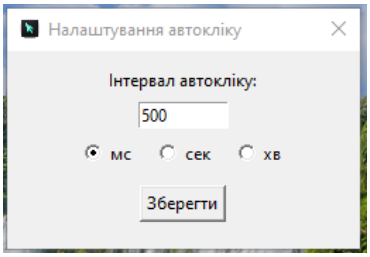

Рисунок 3.29– Вікно «Налаштування автокліку» (ОС – Windows)

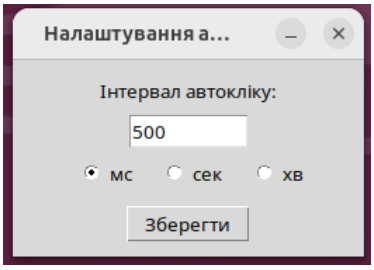

Рисунок 3.30 – Вікно «Налаштування автокліку» (ОС – Linux)

Зупинка автокліку виконується stop_autoclick(), яка встановлює подію auto_click_event і чекає завершення потоку:

```
def stop_autoclick(self):
    if not self.auto_clicking:
        return  # якщо автоклік не запущено – нічого не робиться

    self.auto_click_event.set()  # сигнал потоку завершити цикл
автокліку

    # чекає завершення фонового потоку автокліку
    if self.auto_click_thread and
self.auto_click_thread.is_alive():
        self.auto_click_thread.join()

    self.auto_clicking = False  # скидає прапорець, що автоклік
```

```
активний
```

Після зупинки вікно програми та саме вікно автокліку знову виводяться на передній план. Для автокліку також передбачені гарячі клавіші (Ctrl+A для запуску, Ctrl+S для примусової зупинки, що розповідалось в пункті «Функціонал гарячих клавіш»).

Додавання точки мультикліку. Перша кнопка з іконкою плюса у вікні «Мультиклік» відкривається через кнопку «Створити мультиклік» (рис. 3.31), викликає метод on_click_mouse().

Рисунок 3.31 – Вікно «Мультиклік»

У коді це створює нове крихітне вікно Toplevel, що відображає іконку курсору (25×30 пікселів) і дозволяє його перетягувати мишею (рис. 3.32). Нове вікно додається до списку self.multi_cursors, а в словник self.multi_configs записуються початкові параметри (координати X,Y, кількість кліків, швидкість, затримка). Наприклад, при натисканні кнопки «+» код виконує:

```
# створюється нове плаваюче вікно для курсора
tl = tk.Toplevel(self.window)
tl.overrideredirect(True)  # прибирається рамка та заголовок
вікна
tl.geometry(f"25x30+{x}+{y}")  # встановлюється розмір та
позицію на екрані

# додається мітка з зображенням курсора без рамки
lbl = tk.Label(tl, image=self.img_mouse, bd=0)
lbl.pack(fill="both", expand=True)  # розтягується зображення на
весь простір вікна
```

```
# зберігається посилання на це вікно в списку для подальшого
керування
self.multi_cursors.append(tl)

# ініціалізування конфігурації для цього курсора:
# ключ – унікальний id вікна, значення – початкові параметри:
# x, y – позиція; clicks – лічильник кліків;
# speed – інтервал кліків; delay – затримка перед кліком
self.multi_configs[id(tl)] = {
    'x': x,
    'y': y,
    'clicks': 0,
    'speed': 100,
    'delay': 0
}
```

Усі додані точки можна перетягувати курсором, а активна точка зберігається в self.active_cursor.

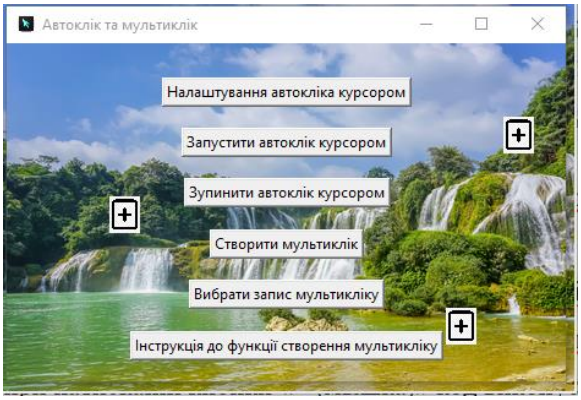

Рисунок 3.32– Відображення додавання іконок курсорів на вікно «Автоклік та мультиклік» (ОС – Windows)

3.5.2 Налаштування параметрів іконки.

Подвійний клік по іконці точки викликає метод open_cursor_settings(), який відкриває діалог «Налаштування курсору». У цьому вікні користувач задає кількість кліків (0 = нескінченно), інтервал між кліками (у мс/сек/хв), та затримку старту (мілісекунди, секунди, хвилини). На рис. 3.33–3.35 показано вікно налаштування іконки на двух ОС. Приклад коду:

```
# поле для вводу кількості кліків
tk.Label(win, text="Кількість кліків (0 =
нескінченно):").grid(...)
entry_clicks = tk.Entry(win, width=10)
# встановлюється початкове значення з конфігурації (за
замовчуванням 1)
entry_clicks.insert(0, str(cfg.get('clicks', 1)))

# поле для вводу інтервалу між кліками
tk.Label(win, text="Інтервал між кліками:").grid(...)
entry_speed = tk.Entry(win, width=10)
# встановлюється значення швидкості з конфігурації (за
замовчуванням 100 мс)
entry_speed.insert(0, str(int(cfg.get('speed', 100))))

# поле для вводу затримки перед стартом автокліку
tk.Label(win, text="Затримка старту:").grid(...)
entry_delay = tk.Entry(win, width=10)
# встановлюється затримка зі збережених налаштувань (за
замовчуванням 0 мс)
entry_delay.insert(0, str(int(cfg.get('delay', 0))))
```

Ці поля створюються в коді open_cursor_settings(). При натисканні «Зберегти» введені значення читаються, конвертуються у мілісекунди та записуються назад у self.multi_configs[id(tl)] для даної точки. Таким чином параметри кожної точки (кількість кліків, інтервал, затримка) зберігаються у відповідному словнику.

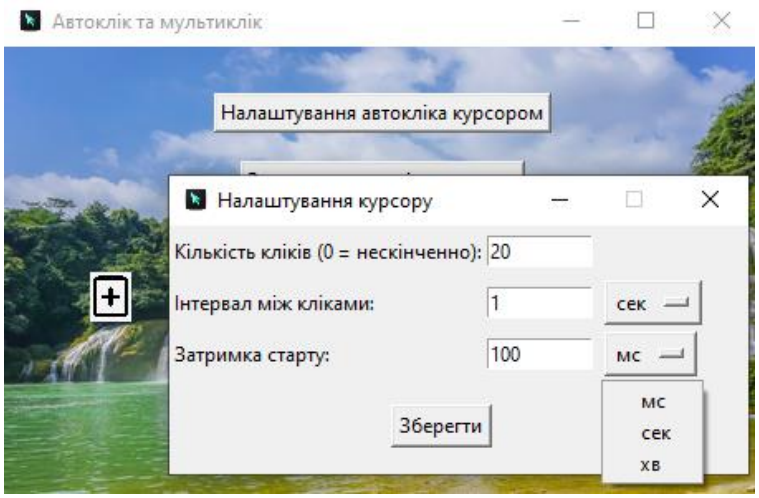

Рисунок 3.33 – Вікно налаштування курсору (ОС – Windows)

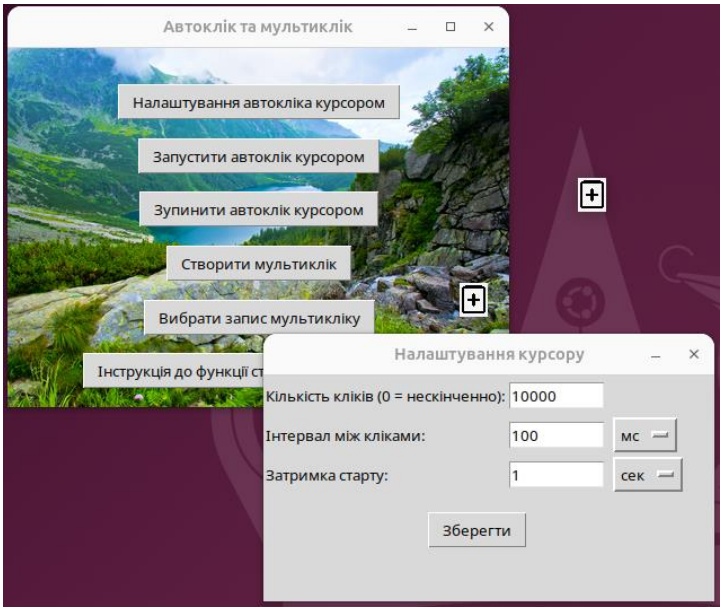

Рисунок 3.34 – Вікно налаштування курсору (ОС – Linux)

Видалення іконки курсору. При натисканні кнопки «–» у вікні «Мультиклік» видаляється остання обрана іконка або, остання створена. Приклад коду:

```
def on_click_minus(self):
    # якщо список курсорів порожній, нічого не діється
    if not self.multi_cursors:
        return

    # якщо активний курсор є в списку - видаляє його, інакше - останній доданий
    tl = self.active_cursor if self.active_cursor in self.multi_cursors else self.multi_cursors[-1]

    # видаляє обраний курсор зі списку
    self.multi_cursors.remove(tl)

    # видаляється збережена конфігурація цього курсора (якщо є)
    self.multi_configs.pop(id(tl), None)

    # знищується віджет курсора
    tl.destroy()


    # скидається активний курсор
    self.active_cursor = None
```

3.5.3 Збереження та завантаження конфігурацій мультикліку.

Для збереження існуючих точок використовується кнопка папки (загрузки): вона викликає on_click_folder(), який просить ввести ім’я запису та зберігає всі точки у JSON–файл у папці multiclick_records. Формується словник:

```
data = {'cursors': []}  # структура для збереження налаштувань
курсорів

for tl in self.multi_cursors:
    cfg = self.multi_configs[id(tl)]  # конфігурація для
поточного вікна-курсора
    data['cursors'].append({
# x-координата (або поточна позиція вікна)
        'x': cfg.get('x', tl.winfo_x()),
# y-координата (або поточна позиція вікна)
        'y': cfg.get('y', tl.winfo_y()),
# кількість кліків (0 = нескінченно)
        'clicks': cfg.get('clicks', 0),
# інтервал між кліками в мс
        'speed': cfg.get('speed', 0),
# затримка перед стартом в мс
        'delay': cfg.get('delay', 0),
# затримка перед стартом в мс
        'speed_raw':  cfg.get('speed_raw', cfg.get('speed', 0)),
# початкове значення швидкості яке обрав користувач (мс, сек,
хв)
        'speed_unit': cfg.get('speed_unit', 'мс'),         #
одиниця виміру швидкості
        'delay_raw':  cfg.get('delay_raw', cfg.get('delay', 0)),
# # початкове значення затримки яке обрав користувач (мс, сек,
хв)
        'delay_unit': cfg.get('delay_unit', 'мс'),         #
одиниця виміру затримки
    })

# запускаються всі налаштування в JSON-файл з відступами та без
екранування Unicode
json.dump(data, f, ensure_ascii=False, indent=2)
```

Після натискання кнопки «Завантажити» (рис. 3.35) у діалоговому вікні «Вибрати запис мультикліка» (яке відкривається через кнопку «Вибрати запис мультикліку» у вікні «Автоклік та мультиклік») читає JSON, очищує

поточні точки та створює нові за збереженими координатами і параметрами. Також можна видалити непотрібні записи через кнопку «Видалити».

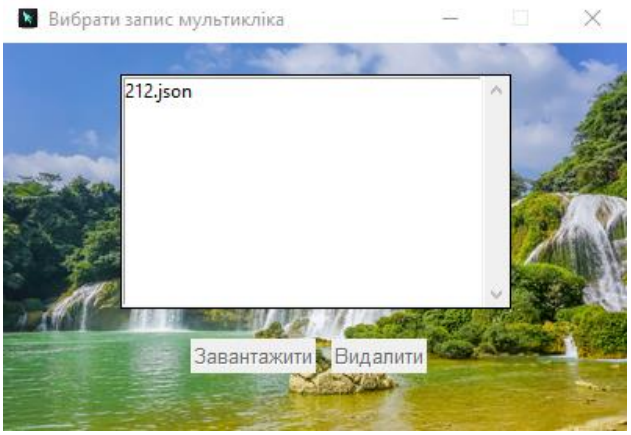

Рисунок 3.35 – Вікно «Вибрати запис мультикліку

3.5.4 Запуск/зупинка мультикліку та особливості Windows/Linux

Після налаштування точок натискання кнопки «трикутник» (btn_triangle) у вікні «Мультиклік» викликає on_click_triangle(). Код починає послідовне виконання по всіх точках, запустивши окремий потік для кожної:

```
# скидається прапорець зупинки для режиму мультикурсорів
self.multi_stop_event.clear()

threads = []
for tl in self.multi_cursors:
    # отримуються конфігурація для поточного вікна-курсора
    cfg = self.multi_configs[id(tl)]

    # створюється демон-потік, що виконає _run_cursor_task
з цією конфігурацією
    thr = threading.Thread(
        target=self._run_cursor_task,  # функція, яка керує
рухом/кліками одного курсора
        args=(cfg,), # передається налаштування курсора як
аргумент
        daemon=True # демон-потік, автоматично завершується
при виході з програми
    )
    thr.start() # запускається потік одразу після створення
    threads.append(thr)  # зберігається об’єкт потоку
```

Метод _run_cursor_task(self, cfg) переміщує курсор у координати x,y і виконує послідовність клацань лівою кнопкою з інтервалом speed та загальною кількістю clicks.

Якщо встановлено затримку, вона виконується перед початком. Усі потоки виконуються паралельно. Якщо для всіх точок задано скінченну кількість кліків, запускається допоміжний моніторинг, який після завершення роботи потоків знову відображає головні вікна програми.

Кнопка «Стоп» (з іконкою img_stop) викликає метод _stop_multi(), що ставить подію self.multi_stop_event, скидає стан UI та повертає видимість вікон. Також передбачена гаряча клавіша Ctrl+D (за замовчуванням) для зупинки мультикліку.

Підтримка Windows/Linux. У коді використовується platform.system() (змінна SYSTEM) для визначення ОС. Якщо SYSTEM == "Windows", то для емуляції кліків застосовується ctypes та виклики WinAPI (SendInput, GetCursorPos). Наприклад, функція move_mouse(x,y) використовує ctypes.windll.user32.SendInput з флагом MOUSEEVENTF_ABSOLUTE, а get_cursor_pos() – GetCursorPos. Для Linux застосовано бібліотеку pynput: створюється контролер mouse.Controller(), а функції move_mouse, mouse_down_at, mouse_up_at просто змінюють аналогічні на Windows.Така логіка забезпечує кросплатформну роботу автоклікера під обома ОС.

ВИСНОВКИ

У процесі виконання дипломної роботи було проведено комплексне дослідження актуальності та вимог до інструментів автоматизації повторюваних дій користувача. На етапі аналізу вивчено існуючі рішення, виявлено їхні переваги й обмеження, що дало змогу сформулювати чіткі завдання для розробки кросплатформного автоклікера з розширеним функціоналом.

Під час проєктування створено UML–моделі на різних рівнях: модель прецедентів демонструє взаємодію користувача із системою; компонентна схема відображає підсистеми й їхні зв’язки; діаграма класів ілюструє статичну структуру об’єктів згідно з принципами SOLID; послідовність виконання макросу відображена в sequence–діаграмі. Це забезпечило гнучку, зрозумілу й легко підтримувану архітектуру.

У ході реалізації вибрано стек технологій, що гарантує кросплатформеність і стабільність роботи під Windows і Linux. Розроблено модулі інтерфейсу, обробки подій, запису та відтворення макросів, глобальних гарячих клавіш, автокліку, мультикліку й керування фоновими зображеннями. Результатом стала програма, що поєднує простий інтерфейс із високою точністю емуляції дій користувача.

Практична користь запропонованого рішення полягає в суттєвому скороченні часу на виконання рутинних операцій у різних сценаріях (офісні завдання, тестування ПЗ, ігри), а також у можливості гнучко налаштовувати макроси та режими роботи. Запропонований підхід із модульною архітектурою й використанням стандартних бібліотек сприяє подальшому розширенню функціоналу та адаптації до нових вимог.

Подальші напрями розвитку включають впровадження сучасного графічного фреймворку для розширення можливостей UI, додавання підтримки macOS і створення візуального редактора макросів із графічним комбінуванням кроків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. GS Auto Clicker – офіційна сторінка утиліти автоматичного кліку миші GS Auto Clicker. URL: https://goldensoft.org/ (дата звернення: 25.05.2025).

2. Документація AutoHotkey – офіційна документація скриптової мови AutoHotkey. URL: https://www.autohotkey.com/docs/ (дата звернення: 25.05.2025).

3. Документація PyAutoGUI – офіційна документація бібліотеки кросплатформної автоматизації GUI PyAutoGUI. URL: https://pyautogui.readthedocs.io/en/latest/ (дата звернення: 25.05.2025).

4. Офіційна документація Python 3.12 (мова Python та стандартна бібліотека, що включає модулі json, threading, platform, sys, shutil, ctypes, subprocess тощо). URL: https://docs.python.org/3.12/ (дата звернення: 25.05.2025).

5. Документація pynput (офіційна документація бібліотеки для керування і моніторингу клавіатури та миші. URL: https://pynput.readthedocs.io/en/latest/ (дата звернення: 25.05.2025).

6. GitHub: boppreh/keyboard (репозиторій Python–бібліотеки для роботи з глобальними подіями клавіатури). URL: https://github.com/boppreh/keyboard (дата звернення: 25.05.2025).

7. Офіційна документація tkinter (стандартний GUI–інтерфейс Python до Tcl/Tk, включаючи діалоги файлів). URL: https://docs.python.org/3.12/library/tkinter.html (дата звернення: 25.05.2025).

8. Документація Pillow (форк PIL) – офіційна документація бібліотеки обробки зображень Pillow. URL: https://pillow.readthedocs.io/ (дата звернення: 25.05.2025).

9. Microsoft Docs: SendInput (WinAPI) – довідка щодо функції SendInput для синтезу подій клавіатури/миші. URL: https://learn.microsoft.com/windows-win32/api/winuser/nf–winuser–sendinput (дата звернення: 25.05.2025).