Міністерство освіти і науки України

Національний університет «Львівська політехніка»

Кафедра автоматизованих систем управління

Курсова робота

з дисципліни "Об’єктно-орієнтоване програмування "

на тему

" Розробка програмної системи "Електронний словник" з динамічним поповненням словникової бази"

Виконав:

студент групи ОІ-22

Петрунів Дмитро

Керівник:

Асистент каф. АСУ

Островка Д.В.

Львів – 2025

Завдання до курсової роботи

з дисципліни "**Об’єктно-орієнтоване програмування**"

Прізвище, ім’я студента: *Петрунів Дмитро*

Група: *ОІ-22*

Тема курсової роботи *Розробка програмної системи "Електронний словник" з динамічним поповненням словникової бази*

Спеціальна частина завдання:

В огляді літературних джерел проаналізувати відкриті матеріали з питань побудови систем електронного перекладу, архітектури клієнт-серверних застосунків, методів організації збереження лексичних даних та протоколів передачі текстової інформації через мережу (TCP/IP).

Дослідити доцільність впровадження об’єктно-орієнтованого підходу для моделювання роботи електронного словника, включаючи інкапсуляцію (приховування логіки роботи з сокетами), композицію (структура словника), поліморфізм (інтерфейси джерел даних) та використання патернів проектування на прикладах сутностей системи (словник, перекладач, мережевий менеджер, локальна історія).

Розробити та реалізувати:

1. *Розробити модель даних і структуру класів для системи E-Dictionary, з огляду на головні атрибути словникової статті (слово, визначення, частина мови), механізми їх збереження у пам'яті та файловій системі.*
2. *Створити структуру для відображення результатів перекладу з підтримкою форматування (теги частин мови, посилання) та забезпечити можливість організації додаткових функцій (наприклад, "Слово дня", фільтрування історії пошуку).*

3. *Реалізувати механізми контролю бізнес-правил і узгодженості даних, зокрема:*
    - *правило унікальності слів при додаванні до бази;*
    - *валідація вхідних даних (заборона пустих запитів або некоректних символів);*
    - *коректна обробка фрагментованих пакетів даних при передачі великих обсягів тексту через мережу (буферизація).*
4. *Розробити засоби для обміну даними між користувачем та системою, включаючи власний текстовий протокол передачі команд через TCP-сокети та парсинг відповідей сервера на стороні клієнта.*
5. *Реалізувати зручний інтерфейс взаємодії користувача з програмою (клієнтську частину) з використанням бібліотеки CustomTkinter, що забезпечує доступ до функцій перекладу, додавання нових слів, перегляду «Слова дня» та збереження локальної історії запитів.*
6. *Для програмних рішень використати мови C++ та Python.*
7. *Термін завершення роботи – 12 грудня 2025 р.*

Завдання видано *07 листопада 2025 р.*

Керівник *асистент Островка Д.В.*

Зміст

Вступ....5
1. Постановка задачі....6
2. ОГЛЯД ТА АНАЛІЗ ЛІТЕРАТУРНИХ ДЖЕРЕЛ....7
2.1. Огляд технологій і програмного забезпечення....7
2.2. Аналіз підходів....7
3. ОПИС СТРУКТУРИ ПРОГРАМНОГО ЗАСОБУ ТА ОСНОВНИХ АЛГОРИТМІВ....8
3.1. Концептуальна модель системи та діаграма прецедентів....9
3.2. Опис програмних компонентів....11
3.2.1. Серверна частина (Backend)....14
3.2.2. Клієнтська частина (Frontend)....15
3.3. Основні алгоритми роботи системи....16
3.4. Взаємодія між компонентами системи....18
4. ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЄКТУ....20
4.1. Загальна структура програмного проєкту....20
4.2. Опис програмних модулів мовою C++....22
4.3. Програмна реалізація мовою Python....24
4.4. Задіяні технології, інструменти та бібліотеки....26
5. ОПИС ПРОВЕДЕНИХ ЕКСПЕРИМЕНТІВ....27
5.1. Інструкція користувачеві....27
5.1.1. Запуск серверної частини (Backend)....27
5.1.2. Запуск клієнтської частини (Frontend)....27
5.1.3. Рекомендований сценарій роботи....28
5.2. Результати тестування програми....31
5.2.2. Перевірка функціоналу пошуку (CRUD - Read)....32
5.2.3. Перевірка додавання даних (CRUD - Create)....32
5.2.4. Перевірка локального збереження....33
5.2.5. Висновок за результатами тестування....33
ВИСНОВКИ....33
ЛІТЕРАТУРНІ ДЖЕРЕЛА....34
Додатки....36
Додаток А – Тексти програмних модулів мовою C++....36
Додаток Б – Тексти програмних модулів мовою Python....49

Вступ

Сучасний світ потребує швидких та зручних інструментів для роботи з мовами. Електронні словники стали невід'ємною частиною життя студентів, перекладачів та всіх, хто працює з іноземними мовами. Проте більшість сучасних рішень мають серйозні недоліки: залежність від інтернет-з'єднання, відсутність конфіденційності даних, обмежені можливості кастомізації та високі затримки через віддалені сервери.

Актуальність даної роботи полягає в створенні локального електронного словника, який поєднує переваги швидкості C++ бекенду, зручності Python інтерфейсу та принципів об'єктно-орієнтованого програмування для забезпечення розширюваності та підтримуваності коду.

Завданням курсової роботи є створення клієнт-серверного електронного словника англійсько-української мови, що забезпечуватиме швидкий пошук перекладів, можливість динамічного додавання нових слів та зручний графічний інтерфейс користувача. Система має підтримувати базові операції створення, перегляду, редагування та видалення словникових записів, а також швидкі інструменти пошуку.

Для досягнення цієї мети передбачено виконання аналізу предметної області, розробку структури даних і проєктування класів, реалізацію серверної частини з використанням принципів об'єктно-орієнтованого програмування та відповідної бази даних, а також створення клієнтської частини з графічним інтерфейсом. Очікуваним результатом є робочий прототип словника, який демонструє застосування ООП-підходів і може слугувати основою для подальшого розширення функціоналу — наприклад, впровадження користувацьких профілів, статистики запитів чи підтримки додаткових мов.

1. Постановка задачі

Ціллю курсової роботи є розробка локальної клієнт-серверної системи E-Dictionary Pro, призначеної для швидкого пошуку перекладів, ведення особистого словника користувача та динамічного поповнення бази знань. Система має забезпечити стабільну роботу в офлайн-режимі, захист даних та зручний інтерфейс для розширення словникового запасу.

Об’єкт і предмет дослідження:

- **Об'єктом** є процеси обробки, зберігання та передачі лексичних даних у розподілених системах.
- **Предметом** є програмні засоби та підходи об’єктно-орієнтованого проєктування для реалізації ефективного механізму пошуку та мережевої взаємодії через сокети.

Функціональні вимоги:

Система повинна забезпечувати:

- **Пошук та Переклад:** Миттєвий пошук англійських слів з отриманням детального перекладу, частин мови та прикладів.
- **Динамічне поповнення:** Можливість додавання нових слів користувачем, які миттєво стають доступними для пошуку без перезавантаження сервера.
- **Слово Дня:** Функція випадкового вибору слова для навчання.
- **Історія та Обране:** Локальне збереження історії запитів та списку улюблених слів у базі даних SQLite.

Нефункціональні вимоги:

- **Архітектура:** Клієнт-Сервер на базі TCP/IP сокетів (Winsock2).
- **Швидкодія:** Використання оптимізованих структур даних (std::map) на сервері (C++) для пошуку за час O(log n).
- **Інтерфейс:** Сучасний графічний інтерфейс (GUI) на базі бібліотеки CustomTkinter (Python).

2. ОГЛЯД ТА АНАЛІЗ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

Створення сучасної словникової системи вимагає вибору технологій, які забезпечують баланс між швидкодією обробки запитів та зручністю користувача. У межах курсового проєкту було проаналізовано підходи до побудови клієнт-серверних застосунків та організації збереження даних.

2.1. Огляд технологій і програмного забезпечення

Для реалізації системи **E-Dictionary Pro** обрано гібридний підхід:

- **Мова C++ (Backend):** Використовується для серверної частини. Це дозволяє реалізувати максимальну швидкість обробки запитів завдяки прямому доступу до пам'яті та використанню ефективних контейнерів STL.
- **Winsock2 (Networking):** Для організації мережевої взаємодії обрано низькорівневі сокети. Це дає повний контроль над протоколом передачі даних, дозволяє реалізувати власний формат повідомлень (CMD|DATA) та уникнути оверхеду HTTP-протоколу.
- **Python + CustomTkinter (Frontend):** Для клієнтської частини обрано Python, що дозволяє швидко створювати сучасні GUI. Бібліотека CustomTkinter забезпечує адаптивний дизайн та підтримку темної теми.
- **SQLite (Database):** Використовується на стороні клієнта для збереження персональних даних (історії та обраного), що гарантує конфіденційність та незалежність від сервера.

2.2. Аналіз підходів

У процесі проєктування було враховано кілька ключових архітектурних принципів:

1. Розділення відповідальності (Separation of Concerns):

Система чітко розділена на "бізнес-логіку" (Сервер) та "презентацію" (Клієнт). Сервер не знає про існування графічного інтерфейсу, а Клієнт не знає, як саме зберігаються слова у файлі. Це спрощує підтримку та тестування.

2. In-Memory Caching (Кешування в пам'яті):

Для забезпечення миттєвого пошуку сервер завантажує весь словник у оперативну пам'ять (std::map) при запуску. Це дозволяє уникнути повільних дискових операцій при кожному запиті клієнта. Запис у файл відбувається лише при додаванні нових слів.

3. Власний текстовий протокол:

Замість використання важких форматів на зразок JSON або XML, розроблено легкий текстовий протокол з роздільником | (наприклад, TRANSLATE|cat). Це значно зменшує обсяг переданих даних та навантаження на парсинг.

4. Асинхронність та Багатопотоковість:

Сервер реалізовано з можливістю обробки черги запитів, а клієнт використовує механізми socket.settimeout, щоб інтерфейс не "зависав" при очікуванні відповіді.

3. ОПИС СТРУКТУРИ ПРОГРАМНОГО ЗАСОБУ ТА ОСНОВНИХ АЛГОРИТМІВ

У цьому розділі наведено деталізований опис архітектури клієнт-серверної системи E-Dictionary Pro, основних структур даних, модулів та алгоритмів, що забезпечують її роботу. Окрему увагу приділено механізмам взаємодії між програмними компонентами через сокетне з’єднання, принципам об’єктно-орієнтованого проєктування та застосованим алгоритмічним підходам, які дозволяють досягти високої швидкодії та гнучкості системи.

Система побудована як класична двокомпонентна архітектура: сервер відповідає за зберігання та обробку словникових даних, а клієнт— за взаємодію

з користувачем та локальні персональні функції. Така структура дозволяє масштабувати серверну частину, не змінюючи логіки роботи клієнтських застосунків.

3.1. Концептуальна модель системи та діаграма прецедентів

Концептуальна модель системи базується на предметній області лексикографії та включає набір інформаційних сутностей, що визначають структуру словника та додаткові можливості клієнтського застосунку.

Основні об’єкти предметної області

1. **Словникова стаття (*Entry*)**

   Є базовою одиницею даних і включає:

   - слово-ключ (*Headword*);
   - один або кілька перекладів;
   - частину мови (іменник, дієслово тощо);
   - приклади вживання;
   - додаткові атрибути (дата додавання, автор запису).

   Саме цей об’єкт є центральним в інформаційній моделі, оскільки на ньому базуються всі пошукові операції.

2. **Словник (*Dictionary*)**

   Представляє собою впорядковану структуру даних, що зберігає словникові статті та надає доступ до операцій:

   - пошуку за ключем,
   - додавання нових елементів,
   - перевірки на дублікати.

   Сервером ця сутність реалізована у вигляді std::map<std::string, Entry> з логарифмічною складністю пошуку.

3. **Історія пошуку (*History*)**

   Локальна сутність на стороні клієнта, що містить список всіх раніше виконаних запитів:

   - слово;
   - дата й час запиту;
   - тип операції.

Зберігається в базі даних SQLite.

4. **Обране (*Favorites*)**

   Колекція користувацьких позначок для швидкого доступу до важливих слів. Служить інструментом персоналізації.

5. **Користувач (*User*)**

   Фізична або логічна особа, яка працює із застосунком. У системі її роль не обмежена, однак поведінка користувача відображена в окремих сценаріях (пошук, додавання, перегляд історії тощо).

Прецеденти (Use Cases)

У межах системи виділено кілька основних сценаріїв взаємодії:

1. **Пошук перекладу (Translate Word)**

   Користувач вводить слово → клієнт формує TCP-запит → сервер знаходить відповідну статтю → повертає результат.

2. **Додавання нового слова (Add Word)**

   Контриб’ютор надсилає команду ADD_WORD, після чого сервер:

   - перевіряє чи існує слово в базі,
   - додає новий запис,
   - зберігає зміни у файл.

3. **Отримання випадкового слова (Word of the Day)**

   Клієнт надсилає запит GET_RANDOM,

   сервер повертає випадкову статтю з наявного набору.

4. **Перегляд історії пошуку**

   Користувач отримує локально збережені записи SQLite.

5. **Керування списком “Обране”**

Додавання / видалення слова зі списку у локальному сховищі клієнта.

Для формалізації функціональних вимог до системи було розроблено діаграму варіантів використання. На рисунку 3.1 наведено діаграму прецедентів (Use Case Diagram), яка ілюструє основні сценарії взаємодії користувача з системою E-Dictionary Pro. До ключових функціональних можливостей віднесено пошук перекладу, перегляд «Слова дня», додавання нових термінів до бази даних із валідацією вводу, а також керування локальною історією та списком обраного.

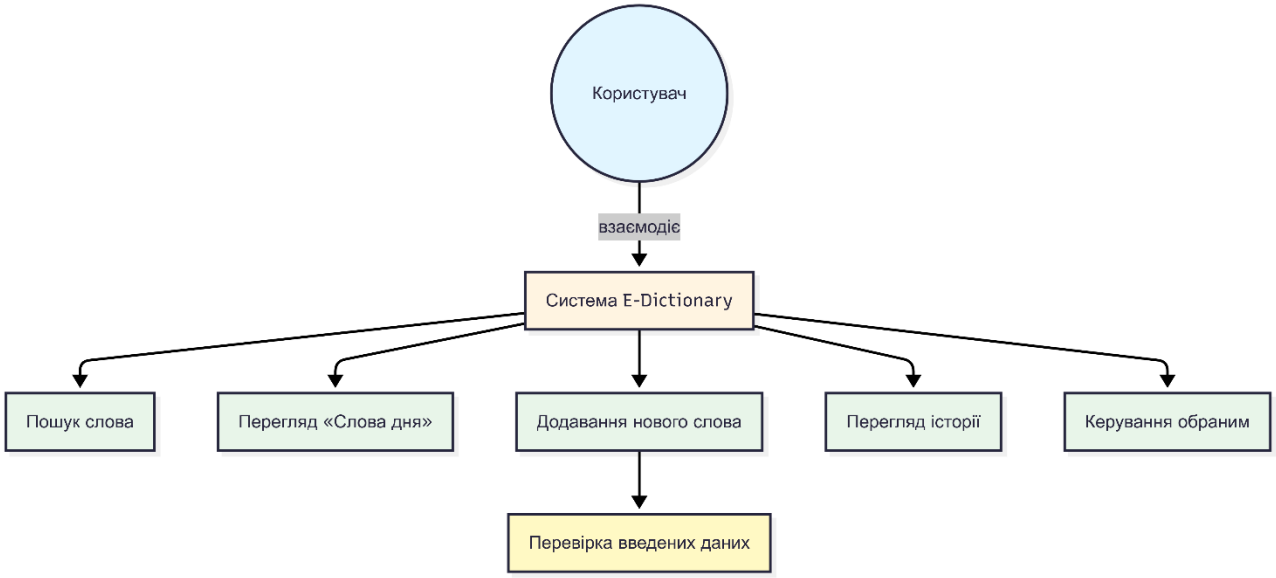


Рисунок 3.1 — Use case Діаграма прецедентів системи E-Dictionary Pro

3.2. Опис програмних компонентів

Архітектура системи передбачає чітке розділення обов’язків між компонентами.

Програмна реалізація базується на об'єктно-орієнтованому підході, що забезпечує модульність та гнучкість коду. На рисунку 3.2 представлено діаграму класів (Class Diagram), що відображає статичну структуру програмної системи та ієрархічні зв’язки між її компонентами. Діаграма демонструє чіткий поділ на серверну (C++) та клієнтську (Python) частини, а також реалізацію

принципу поліморфізму через інтерфейс IDictionarySource, що дозволяє абстрагуватися від конкретного способу збереження даних.

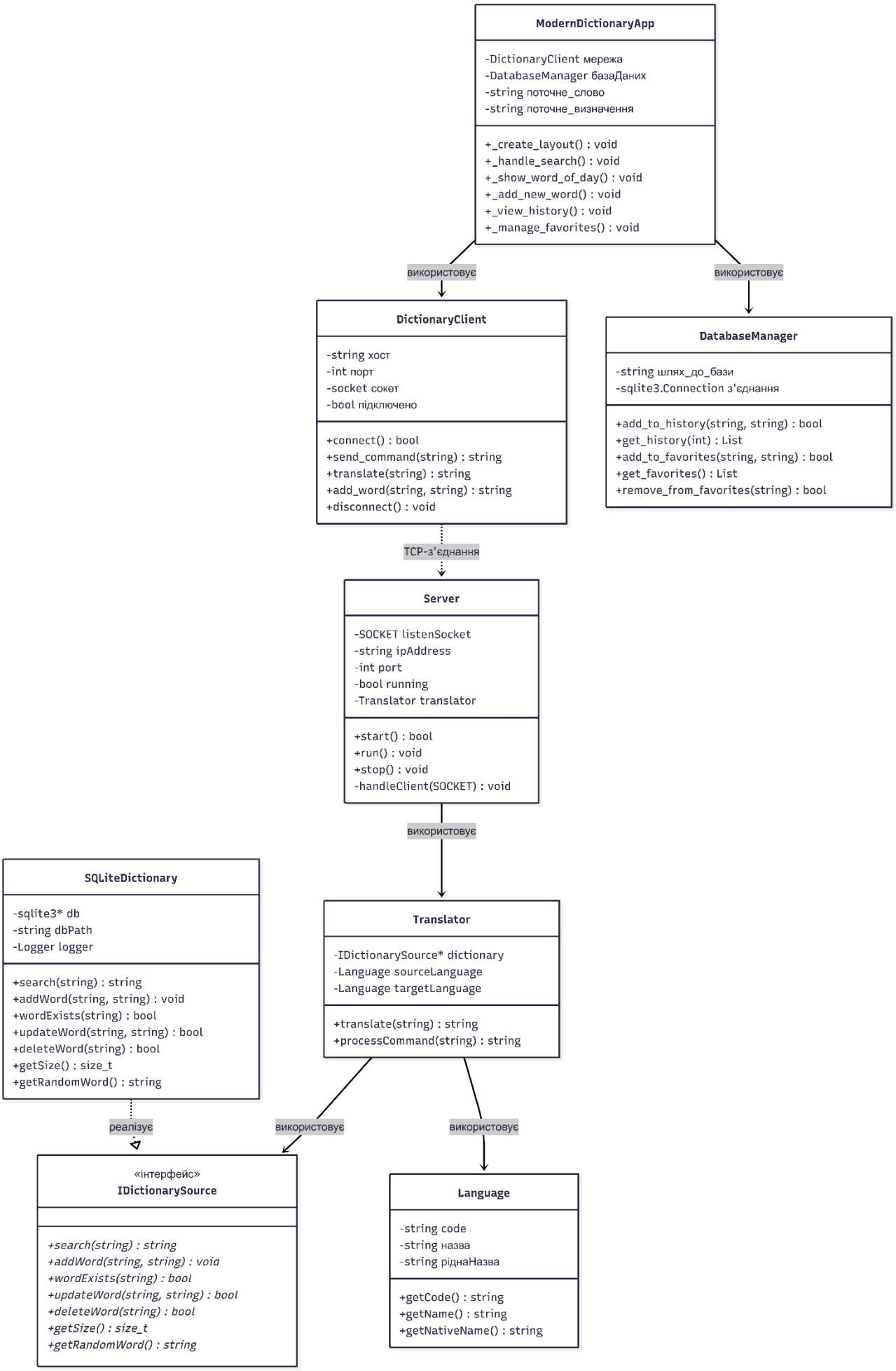

ModernDictionaryApp
-DictionaryClient мережа
-DatabaseManager базаДаних
-string поточне_слово
-string поточне_визначення
+_create_layout() : void
+_handle_search() : void
+_show_word_of_day() : void
+_add_new_word() : void
+_view_history() : void
+_manage_favorites() : void
використовує
використовує
DictionaryClient
-string хост
-int порт
-socket сокет
-bool підключено
+connect() : bool
+send_command(string) : string
+translate(string) : string
+add_word(string, string) : string
+disconnect() : void
DatabaseManager
-string шлях_до_бази
-sqlite3.Connection з'єднання
+add_to_history(string, string) : bool
+get_history(int) : List
+add_to_favorites(string, string) : bool
+get_favorites() : List
+remove_from_favorites(string) : bool
TCP-з'єднання
Server
-SOCKET listenSocket
-string ipAddress
-int port
-bool running
-Translator translator
+start() : bool
+run() : void
+stop() : void
-handleClient(SOCKET) : void
використовує
SQLiteDictionary
-sqlite3* db
-string dbPath
-Logger logger
+search(string) : string
+addWord(string, string) : void
+wordExists(string) : bool
+updateWord(string, string) : bool
+deleteWord(string) : bool
+getSize() : size_t
+getRandomWord() : string
Translator
-IDictionarySource* dictionary
-Language sourceLanguage
-Language targetLanguage
+translate(string) : string
+processCommand(string) : string
реалізує
використовує
використовує
«інтерфейс»
IDictionarySource
+search(string) : string
+addWord(string, string) : void
+wordExists(string) : bool
+updateWord(string, string) : bool
+deleteWord(string) : bool
+getSize() : size_t
+getRandomWord() : string
Language
-string code
-string назва
-string рідназва
+getCode() : string
+getName() : string
+getNativeName() : string

Рисунок 3.2 — Class Діаграма архітектури системи

Нижче описані основні модулі та класи обох частин системи.

3.2.1. Серверна частина (Backend)

Сервер реалізовано мовою **C++**, з використанням:

- **Winsock2** — для TCP-взаємодії;
- **std::map** — для основної структури словника;
- **fstream** — для роботи з файловим сховищем.

Сервер працює в режимі постійного слухача, приймає з'єднання та послідовно обробляє команди клієнта.

Основні завдання серверної частини

1. **Обробка мережевих запитів**

   Сервер формує цикл:

   - прийняти команду →
   - розпарсити текст →
   - визначити тип операції →
   - виконати потрібний метод.

2. **Текстовий протокол команд**

   Протокол складається з ключових слів:

Команда	**Призначення**
TRANSLATE word	Пошук перекладу
ADD_WORD word:translation:type:example	Додавання нового слова
GET_RANDOM	Отримання випадкової статті

3. Такий протокол є простим, легким у розборі й не вимагає зовнішніх бібліотек.

4. **Робота зі структурою даних std::map**

   - Гарантована впорядкованість ключів.
   - Пошук виконується за деревом червоно-чорного типу.
   - Вставка та пошук займають $O(\log n)$.

5. **Файлове збереження словника**

   Під час запуску сервер зчитує дані з файлу dictionary.txt.
   Після операцій ADD_WORD — дозаписує нові елементи.

   Логіка зберігання інкапсульована в інтерфейсі IDictionarySource, що дозволяє надалі замінити файлове сховище на SQL-сервер чи іншу технологію.

6. **Валідація даних**

   Перед додаванням:
   - перевірка чи слово вже існує;
   - коректність частини мови;
   - наявність усіх полів словникової статті.

Класи серверної частини

- Entry — структура словникової статті.
- Dictionary — обгортка над std::map, реалізує find(), insert().
- FileDictionarySource — робота з файлом.
- Server — ядро програми, містить логіку роботи з сокетами.

3.2.2. Клієнтська частина (Frontend)

Клієнтська частина розроблена мовою Python на базі бібліотеки CustomTkinter, що забезпечує гнучкий і сучасний GUI.

Функції клієнтського застосунку

1. **Головне вікно пошуку**
   - Поле введення слова.
   - Кольорове форматування частин мови (іменник — синій, дієслово — зелений тощо).
   - Відображення декількох перекладів.
2. **Форма додавання нового слова**
   - Поля для заповнення всіх атрибутів.
   - Перевірка правильності введення.

- Інформування про успіх або помилку.

3. **Слово дня**
    - Кнопка “Get Random”.
    - Відображення випадкового слова з сервера.
4. **Історія пошуку**
    - Автоматичний запис у SQLite.
    - Перегляд таблиці через інтерфейс.
5. **Обране**
    - Збереження в SQLite.
    - Відображення списку та можливість видалення.

Мережева взаємодія клієнта

Мережеву логіку інкапсульовано в класі NetworkManager, який забезпечує:

- встановлення TCP-з’єднання;
- формування текстових команд;
- автоматичну обробку фрагментованих пакетів;
- отримання відповіді та повернення її в структурованому вигляді.

Алгоритм взаємодії клієнта із сервером:

1. Користувач натискає кнопку → GUI викликає метод у NetworkManager.
2. Формується текстова команда (TRANSLATE hello).
3. Клієнт надсилає її через сокет.
4. Сервер обробляє запит і повертає результат.
5. Результат передається назад у GUI, який відображає вихід користувачу.

3.3. Основні алгоритми роботи системи

Нижче наведені ключові алгоритми, що забезпечують продуктивність та коректність роботи застосунку.

Алгоритм пошуку слова (на сервері)

1. Отримати команду TRANSLATE word.
2. Нормалізувати (перевести в нижній регістр).

3. Виконати map.find(word).
4. Якщо елемент знайдено — сформувати розширену відповідь.
5. Якщо ні — повернути повідомлення “Word not found”.

Час виконання: **O(log n)**.

Алгоритм додавання слова

1. Сервер отримує команду ADD_WORD.
2. Розпарсити дані та виконати початкову перевірку.
3. Виконати пошук у map:
    - якщо слово вже є — відхилити операцію;
    - інакше додати елемент.
4. Дозаписати елемент у файл.
5. Повернути клієнту відповідь про успіх.

Алгоритм вибірки випадкового слова

1. Сервер генерує випадкове число в діапазоні [0; size-1].
2. Ітерується до відповідного елемента в map.
3. Повертає клієнту сформовану статтю.

Алгоритм збереження історії на клієнті

1. Після отримання відповіді від сервера клієнт викликає метод save_history().
2. У таблицю SQLite записуються:
    - слово,
    - відповідь сервера,
    - дата та час запиту.
3. Дані зберігаються локально, без передачі на сервер.

Підсумки до розділу

Розроблена архітектура клієнт-серверного електронного словника поєднує:

- швидкі структури даних (std::map);
- просту та надійну передачу даних (TCP-сокети);
- персоналізацію (SQLite на клієнті);
- сучасний користувацький інтерфейс (CustomTkinter);
- модульність та розширюваність завдяки принципам ООП.

Система легко піддається масштабуванню: у майбутньому її можна доповнити підтримкою авторизації, статистикою запитів, кешуванням на сервері чи хмарним розгортанням.

3.4. Взаємодія між компонентами системи

Для деталізації динамічної поведінки системи було змодельовано процес обробки запиту користувача. На рисунку 3.3 зображено діаграму послідовності (Sequence Diagram) для сценарію виконання перекладу слова. Вона візуалізує часовий порядок передачі повідомлень та життєвий цикл запиту: від ініціалізації у графічному інтерфейсі (ModernDictionaryApp) через мережевий шлюз (DictionaryClient) до обробки на сервері та повернення результату.

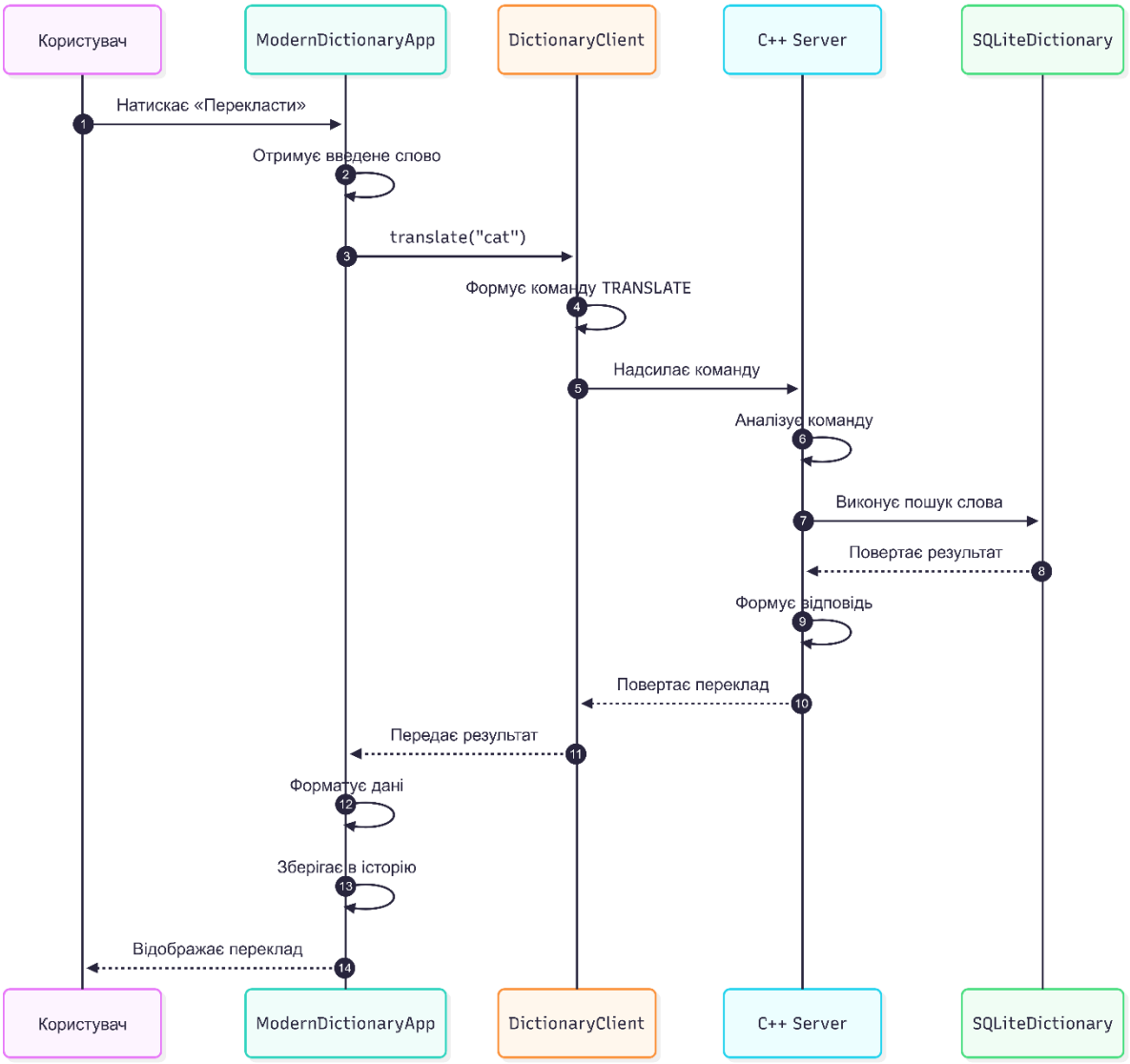


Рисунок 3.3 — Діаграма послідовності обробки запиту на переклад

Взаємодія між частинами системи **E-Dictionary Pro** має класичний клієнт-серверний характер і базується на обміні текстовими повідомленнями через постійне або сесійне TCP-з'єднання:

1. **Ініціалізація:** Користувач вводить слово або натискає кнопку (наприклад, "Translate" або "+") у графічному інтерфейсі (GUI) на базі CustomTkinter.
2. **Формування запиту:** Клас NetworkManager на клієнті формує текстову команду у форматі власного протоколу (наприклад, TRANSLATE|cat| або ADD_WORD|test|definition).

3. **Передача даних:** Команда кодується в байти (UTF-8) і відправляється через сокет на порт сервера (8080).
4. **Обробка на сервері:** Сервер приймає пакет, парсить рядок за роздільником |, визначає тип команди та виконує пошук або запис у структуру даних std::map.
5. **Відповідь:** Сервер формує текстову відповідь (наприклад, переклад слова або статус операції) і відправляє її назад клієнту.
6. **Оновлення клієнта:** Клієнтський застосунок отримує дані, декодує їх, оновлює віджети інтерфейсу та, у разі успішного пошуку, зберігає запис у локальну базу даних SQLite (Історія).

Такий підхід забезпечує мінімальні затримки при передачі даних, оскільки відсутні накладні витрати HTTP-протоколу, а також дозволяє підтримувати єдине джерело істини (словник) на сервері при збереженні приватності історії пошуків на клієнті.

4. ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЄКТУ

У цьому розділі описується практичне виконання програмної системи **E-Dictionary Pro**, яка забезпечує переклад слів, ведення словникової бази та збереження історії запитів. Наводяться структура програмного проєкту, засади виконання серверної та клієнтської частин, а також перелік задіяних технологій і бібліотек.

4.1. Загальна структура програмного проєкту

Проєкт реалізовано як розподілену систему, що складається з двох автономних частин:

- **Серверна частина (Backend):** Виконана мовою **C++** (стандарт C++17) із використанням бібліотеки **Winsock2** для роботи з мережею та стандартної бібліотеки шаблонів (STL) для збереження даних у пам'яті.

- **Клієнтська частина (Frontend):** Реалізована мовою **Python** із використанням бібліотеки **CustomTkinter** для графічного інтерфейсу та модуля sqlite3 для локального збереження налаштувань та історії.

Сервер відповідає за швидкий пошук слів у завантаженому словнику, обробку вхідних підключень, валідацію нових слів та синхронізацію даних з файловим сховищем. Клієнт надає зручний інтерфейс для взаємодії користувача з системою, візуалізує отримані переклади з використанням кольорового форматування та керує локальними даними (Обране/Історія).

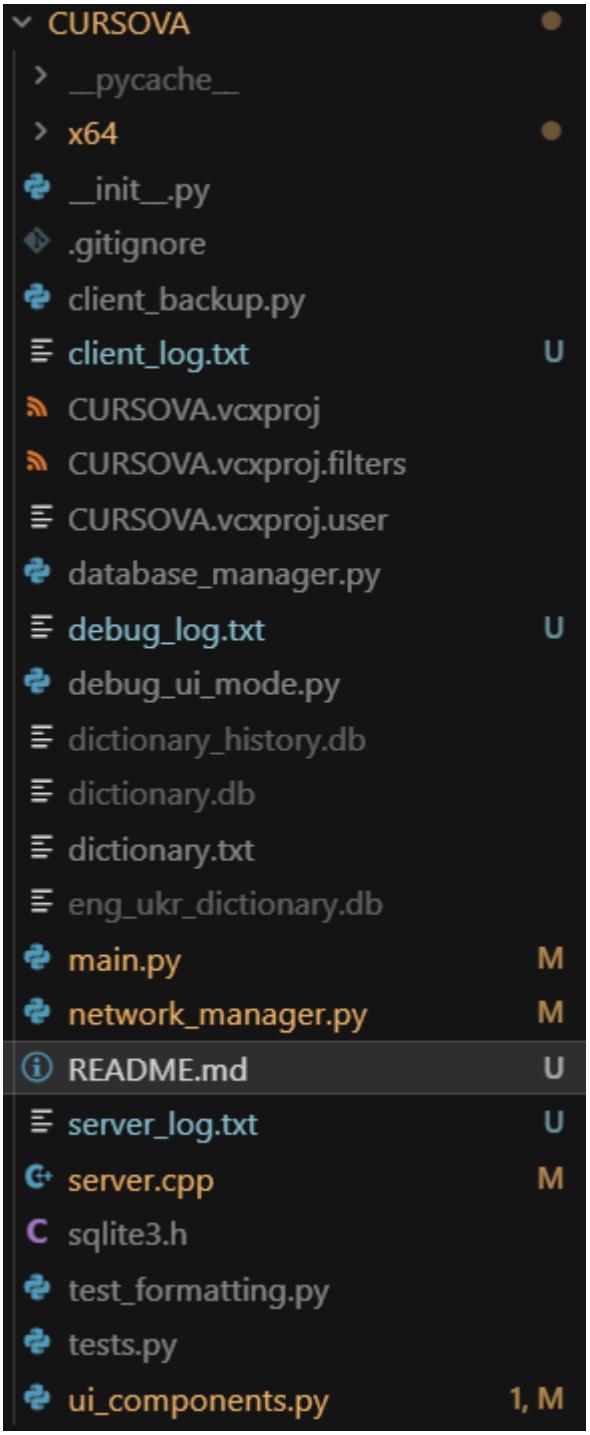


Рисунок 4.1 - Дерево проєкту

4.2. Опис програмних модулів мовою C++

Серверна частина виконана як модульна конструкція з чітким розподілом обов'язків між рівнями мережевої взаємодії, бізнес-логіки та зберігання даних. Програма реалізована з використанням об’єктно-орієнтованого підходу.

1. Модуль доступу до даних (Data Layer)

- **Інтерфейс IDictionarySource:** Абстрактний клас, що визначає контракт для роботи зі словником (методи search, addWord, random). Це дозволяє застосувати принцип поліморфізму та легко змінювати джерело даних (файл, БД, пам'ять).
- **Реалізація сховища (FileDictionary / MemoryMap):**
    - Використовується контейнер std::map<std::string, std::string> для зберігання слів у оперативній пам'яті, що забезпечує миттєвий пошук із складністю O(log n).
    - Забезпечується персистентність даних шляхом синхронізації з файловою системою (dictionary.txt) при додаванні нових записів.

2. Модель предметної області (Entities)

- У системі виділено ключові сутності: Language (мова), Dictionary (словник), Session (клієнтська сесія).
- Клас Language інкапсулює коди мов (наприклад, "EN", "UA") та правила локалізації.

3. Мережевий модуль (Network Layer)

- Реалізовано на базі бібліотеки **Winsock2**.
- **Клас Server:** Відповідає за ініціалізацію сокетів, прив'язку до порту (8080) та прослуховування вхідних з'єднань.
- Реалізовано цикл обробки повідомлень (recv / send), який приймає байтові потоки, декодує їх з UTF-8 та передає на обробку.

4. Контролер обробки команд (Command Processor)

- Виконує роль посередника між мережею та словником.
- Приймає текстові команди у форматі власного протоколу (наприклад, TRANSLATE|cat).
- Парсить вхідний рядок, визначає тип операції та викликає відповідні методи словника.
- Формує текстові відповіді для клієнта (результат перекладу або повідомлення про помилку).

5. Реалізація бізнес-правил

- **Валідація:** Перед додаванням слова система перевіряє його наявність у базі (захист від дублікатів) та коректність формату.
- **Цілісність даних:** Операції запису у файл виконуються в режимі додавання (append) або повного перезапису (при редагуванні), що гарантує збереження бази між перезапусками сервера.
- **Обробка кодувань:** Забезпечено примусове перемикання консолі сервера в режим UTF-8 (SetConsoleOutputCP(65001)) для коректного відображення кирилиці в логах.

Такий розподіл дозволяє спростити супровід проєкту, забезпечує високу швидкодію завдяки роботі з пам'яттю та полегшує подальше нарощування функціоналу (наприклад, додавання нових команд).

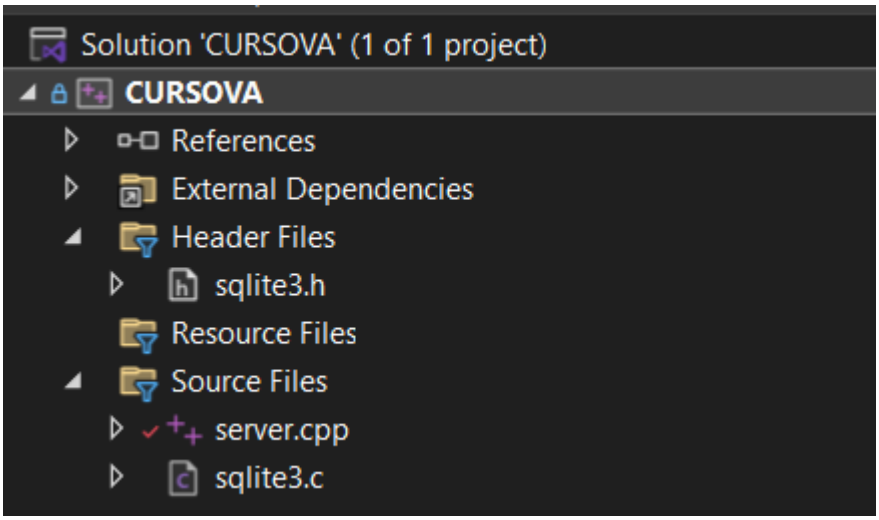


Рисунок 4.2 Дерево папки sln в VS 2022

4.3. Програмна реалізація мовою Python

Клієнтська частина реалізована як десктопний застосунок із графічним інтерфейсом (GUI) на базі бібліотеки **CustomTkinter**. Вона використовується для взаємодії користувача з системою, відправки запитів на сервер та візуалізації отриманих перекладів.

Головні компоненти інтерфейсу:

- **Головне вікно пошуку (Main Dashboard):** Центральний елемент, що містить поле вводу, кнопку "Translate" та область результатів. Результати відображаються з використанням форматованого тексту (Rich Text).

- **Віджет «Слово дня» (Word of the Day):** Спеціальний блок, який автоматично запитує у сервера випадкове слово при запуску або при натисканні кнопки оновлення.
- **Модальне вікно додавання (Add Word Dialog):** Форма для введення нового слова та його визначення. Забезпечує валідацію даних (перевірка на порожні поля) перед відправкою на сервер.
- **Панель історії та обраного:** Інтерфейс для перегляду попередніх запитів, які зберігаються у локальній базі даних SQLite.

Для забезпечення модульності та повторного використання коду логіка розділена на спеціалізовані класи:

1. **Клас NetworkManager (у модулі network_manager.py):**

Відповідає за низькорівневу роботу з мережею. Він містить:

- Конфігурацію хоста та порту;
- Метод send_command, який кодує текстові команди в UTF-8 байти;
- Логіку буферизації (збір фрагментованих пакетів у циклі) для коректного отримання довгих відповідей;
- Обробку помилок з’єднання та таймаутів.

2. **Модуль ui_components.py:**

Містить логіку побудови інтерфейсу та рендерингу тексту. Ключовою особливістю є метод render_rich_text, який:

- Парсить "сиру" відповідь сервера;
- Використовує **Regular Expressions (Regex)** для виявлення тегів частин мови (наприклад, [NOUN], [VERB]);
- Динамічно стилізує текст: заголовки стають синіми, теги — зеленими, а посилання — інтерактивними;
- Адаптує розмір шрифту та відступи залежно від контексту (компактний вигляд для "Слова дня" або повний для результатів пошуку).

Такий підхід дозволяє відокремити логіку мережевої взаємодії від логіки відображення, що робить інтерфейс стійким до змін у протоколі та забезпечує високу якість візуалізації даних.

4.4. Задіяні технології, інструменти та бібліотеки

У ході реалізації системи **E-Dictionary Pro** було застосовано наступний стек технологій та інструментів:

Серверна частина (Backend):

- **C++ (стандарт C++17):** Основна мова розробки, обрана за високу швидкодію та ефективне управління пам'яттю.
- **Winsock2:** Бібліотека для реалізації мережевої взаємодії через TCP/IP сокети (API операційної системи Windows).
- **Standard Template Library (STL):** Використання контейнерів std::map та std::vector для зберігання та обробки словникових даних у пам'яті.
- **Visual Studio 2022 (MSVC):** Середовище розробки та компілятор.

Клієнтська частина (Frontend):

- **Python 3.12:** Мова програмування для розробки клієнтського застосунку.
- **CustomTkinter:** Сучасна бібліотека для створення графічного інтерфейсу (GUI) з підтримкою темної теми та адаптивного дизайну.
- **SQLite3:** Вбудована бібліотека Python для роботи з локальною базою даних (історія та обране).
- **Socket:** Стандартний модуль Python для реалізації клієнтського сокет-з'єднання.

Додаткові інструменти:

- **Git / GitHub:** Система контролю версій для відстеження змін у коді.
- **Mermaid.js:** Інструмент для побудови діаграм архітектури (Use Case, Sequence Diagram).

Застосований технологічний стек забезпечив створення швидкої, автономної системи з сучасним інтерфейсом та надійною мережевою взаємодією.

5. ОПИС ПРОВЕДЕНИХ ЕКСПЕРИМЕНТІВ

У межах курсового проєкту було виконано комплексну перевірку працездатності системи **E-Dictionary Pro**. Тестування включало перевірку стабільності мережевого з'єднання, коректність роботи алгоритмів пошуку, збереження даних та зручність графічного інтерфейсу.

5.1. Інструкція користувачеві

5.1.1. Запуск серверної частини (Backend)

Сервер є консольним додатком, який завантажує словникову базу у пам'ять при старті.

Кроки запуску:

1. Відкрити папку з проєктом.
2. Запустити виконуваний файл server.exe.
3. Дочекатися повідомлення в консолі: [OK] Server listening on port 8080.

Примітка: Сервер повинен залишатися запущеним протягом усього сеансу роботи клієнта.

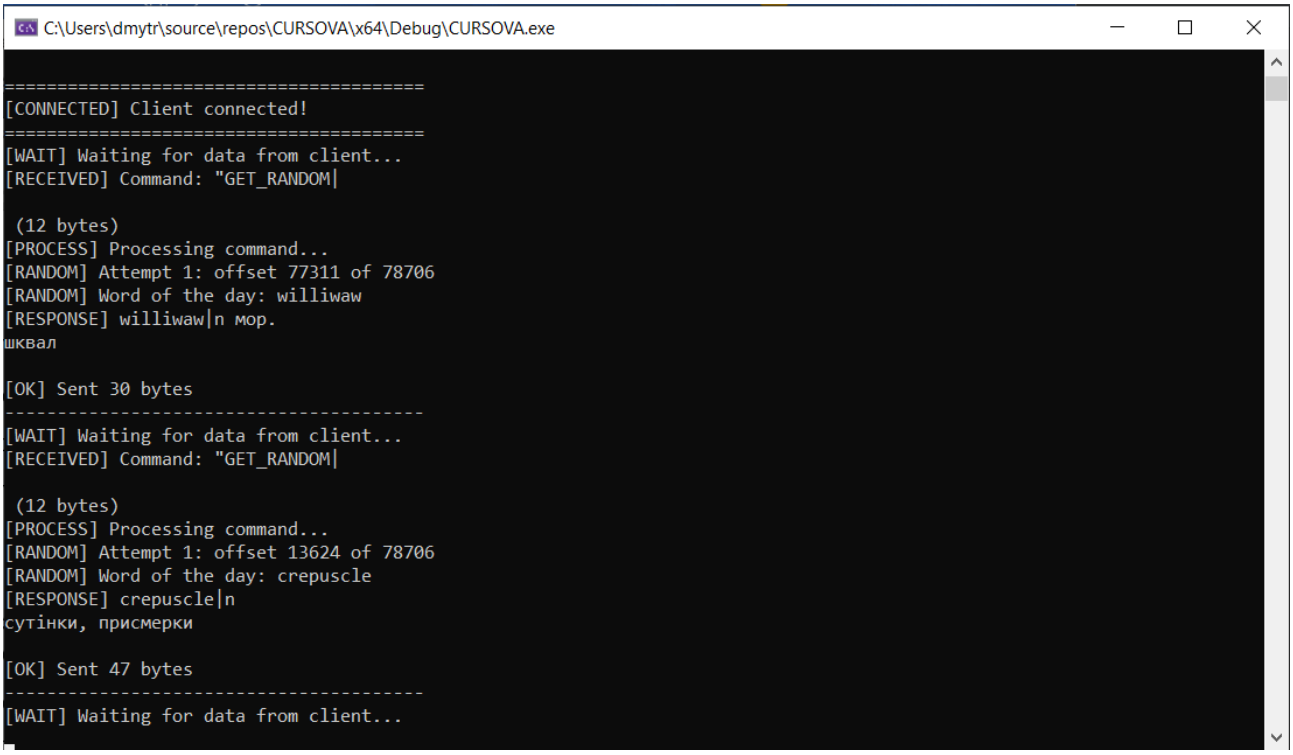


Рисунок 5.1 – Сервер після приєднання клієнта

5.1.2. Запуск клієнтської частини (Frontend)

Клієнт — це графічний застосунок на Python.

Кроки запуску:

1. Відкрити термінал у папці з файлом main.py.
2. Виконати команду: python main.py.
3. У вікні, що з'явиться, перевірити статус підключення (індикатор "Online").

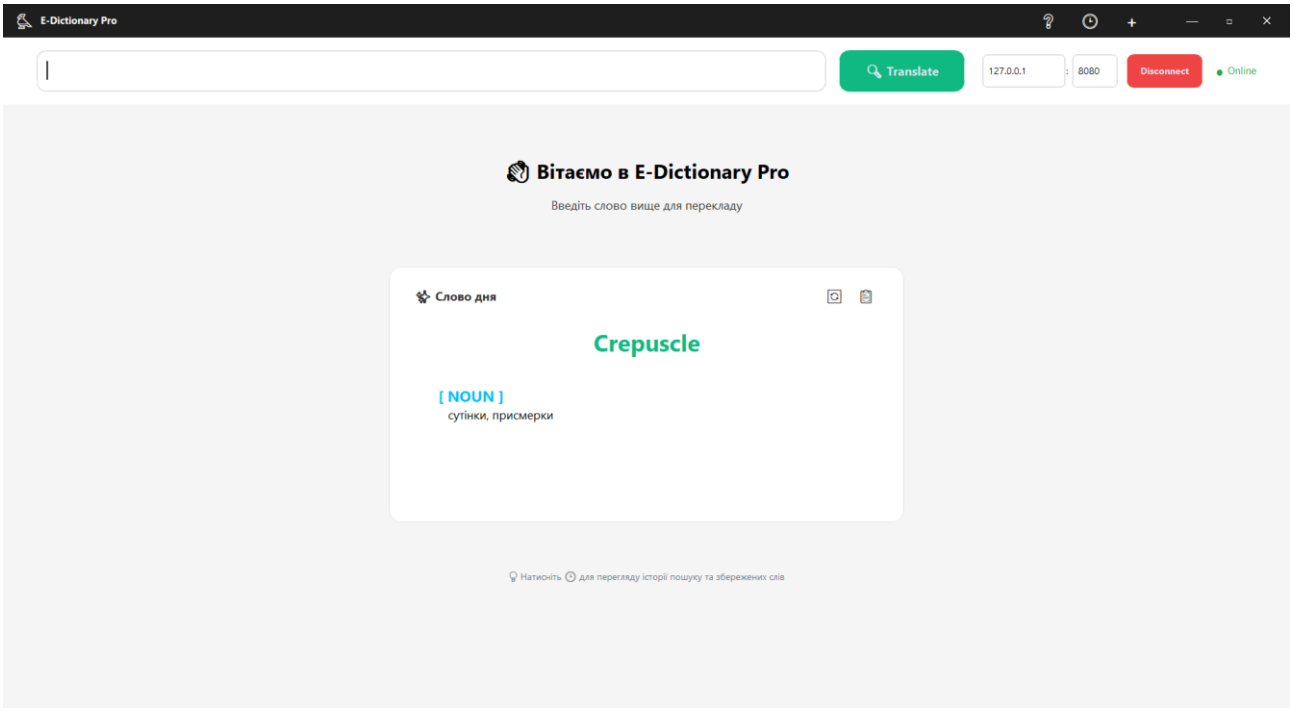


Рисунок 5.2 – Головне вікно програми

5.1.3. Рекомендований сценарій роботи

1. **Пошук:** Ввести слово (наприклад, "step") у поле пошуку та натиснути Enter. Перевірити отриманий переклад.

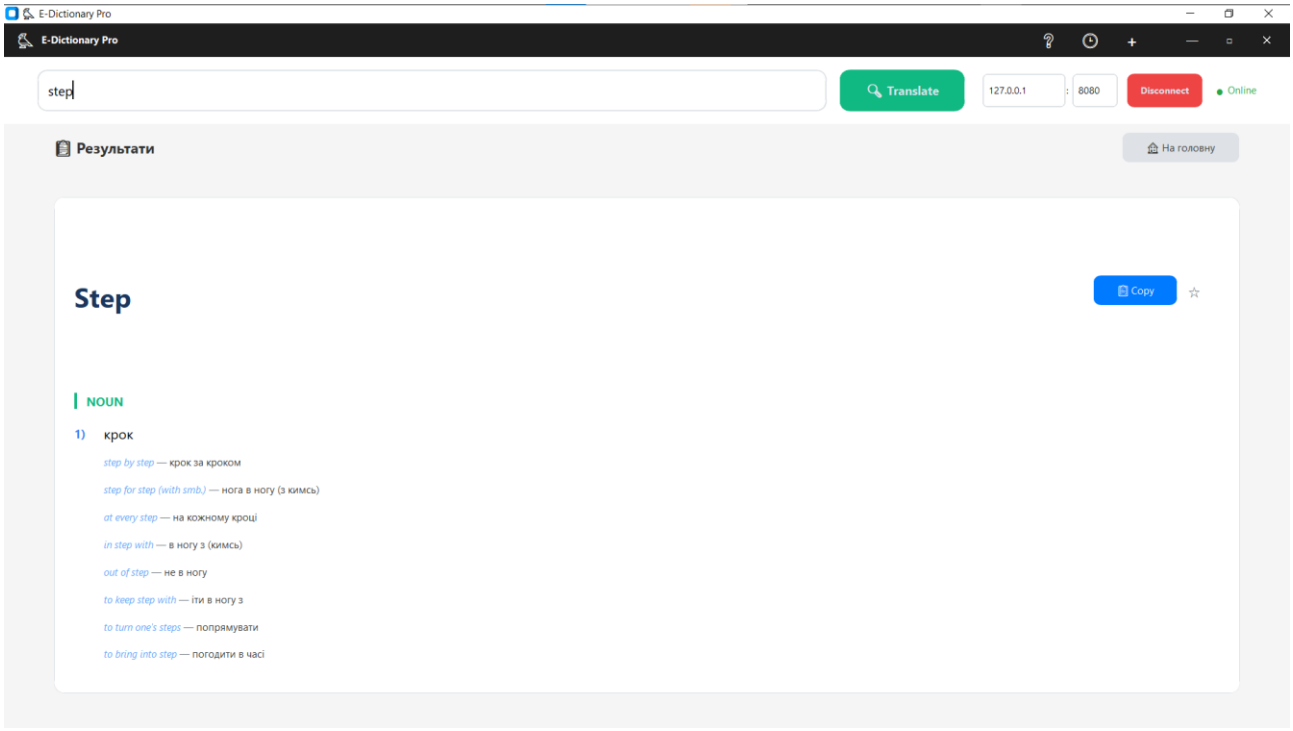
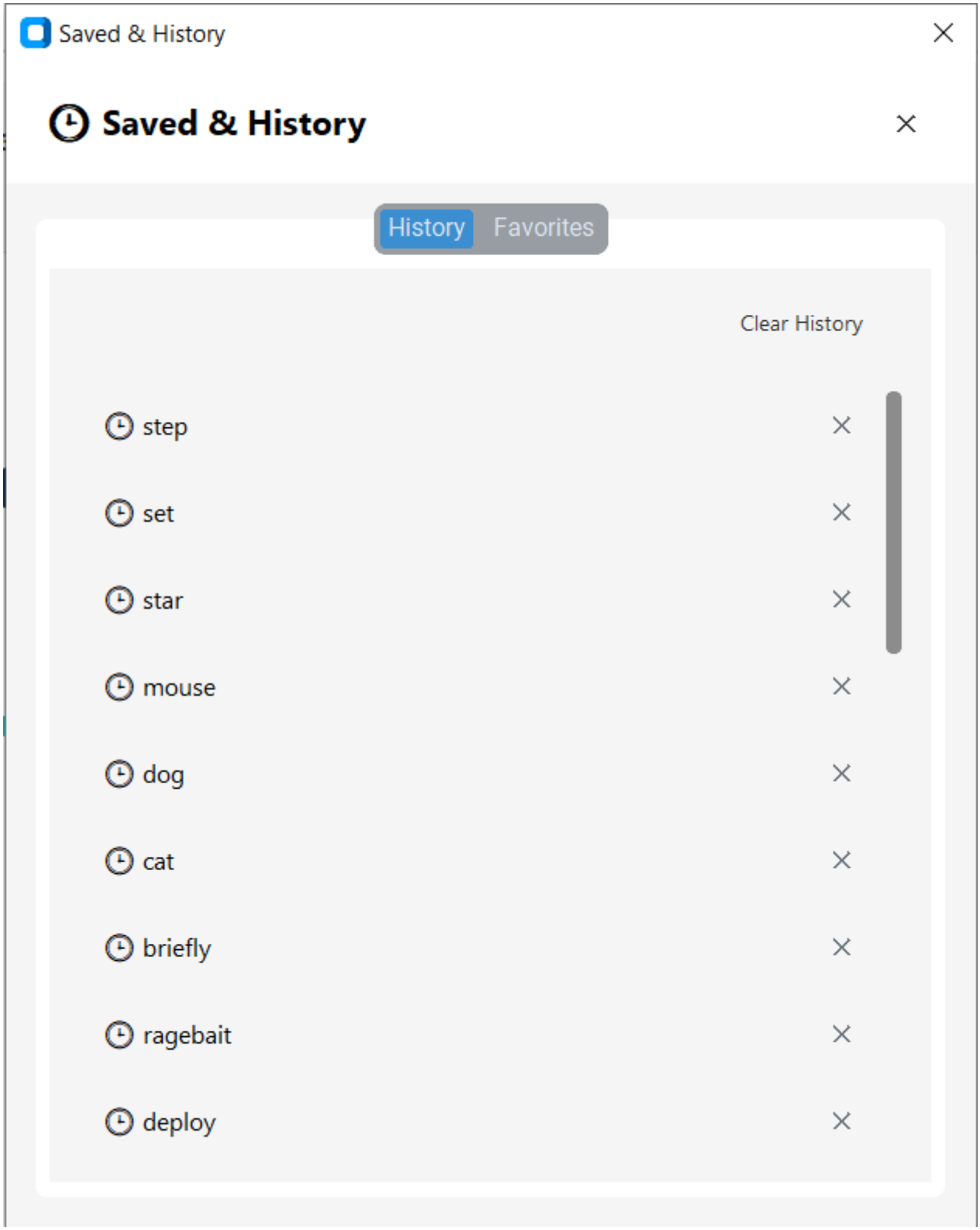


Рисунок 5.1.3.1 – Результат виконання операції

2. **Слово дня:** Натиснути кнопку оновлення на віджеті "Слово дня", щоб отримати новий випадковий термін (можна побачити на Рисунку 5.2).
3. **Додавання слів:** Натиснути кнопку "+", ввести нове слово та визначення. Перевірити, чи з'явилося воно у пошуку.
4. **Історія:** Натиснути кнопку "Годинник", щоб переглянути попередні та збережені запити.


Saved & History
Saved & History
History
Favorites
Clear History
step
set
star
mouse
dog
cat
briefly
ragebait
deploy

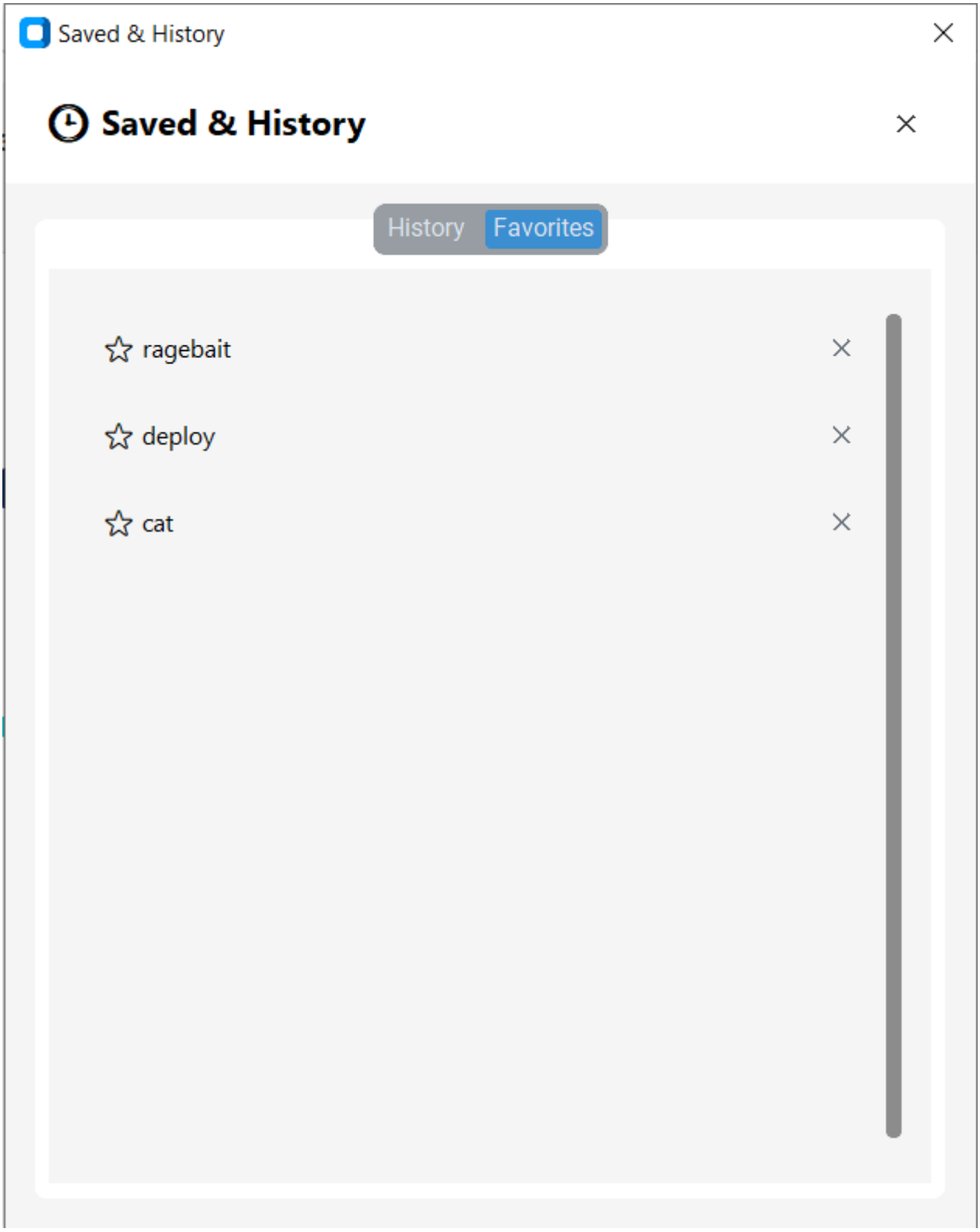


Рисунок 5.1.3.4.1/5.1.3.4.2 – Виклики опцій Saved та Favourites

5.2. Результати тестування програми

Тестування проводилося методом "чорної скриньки" (Black Box Testing) з боку користувача та перевіркою логів сервера.

5.2.1. Перевірка мережевої взаємодії

Було протестовано стабільність TCP-з'єднання.

- **Результат:** Клієнт успішно підключається до сервера. При розриві з'єднання (вимкнення сервера) клієнт показує відповідне повідомлення та намагається перепідключитися.
- **Обробка великих даних:** Реалізований алгоритм буферизації коректно збирає фрагментовані пакети (випадки, коли опис слова перевищує розмір буфера 4096 байт).

5.2.2. Перевірка функціоналу пошуку (CRUD - Read)

- **Пошук існуючого слова:** Система повертає коректний переклад менш ніж за 50 мс (завдяки std::map).

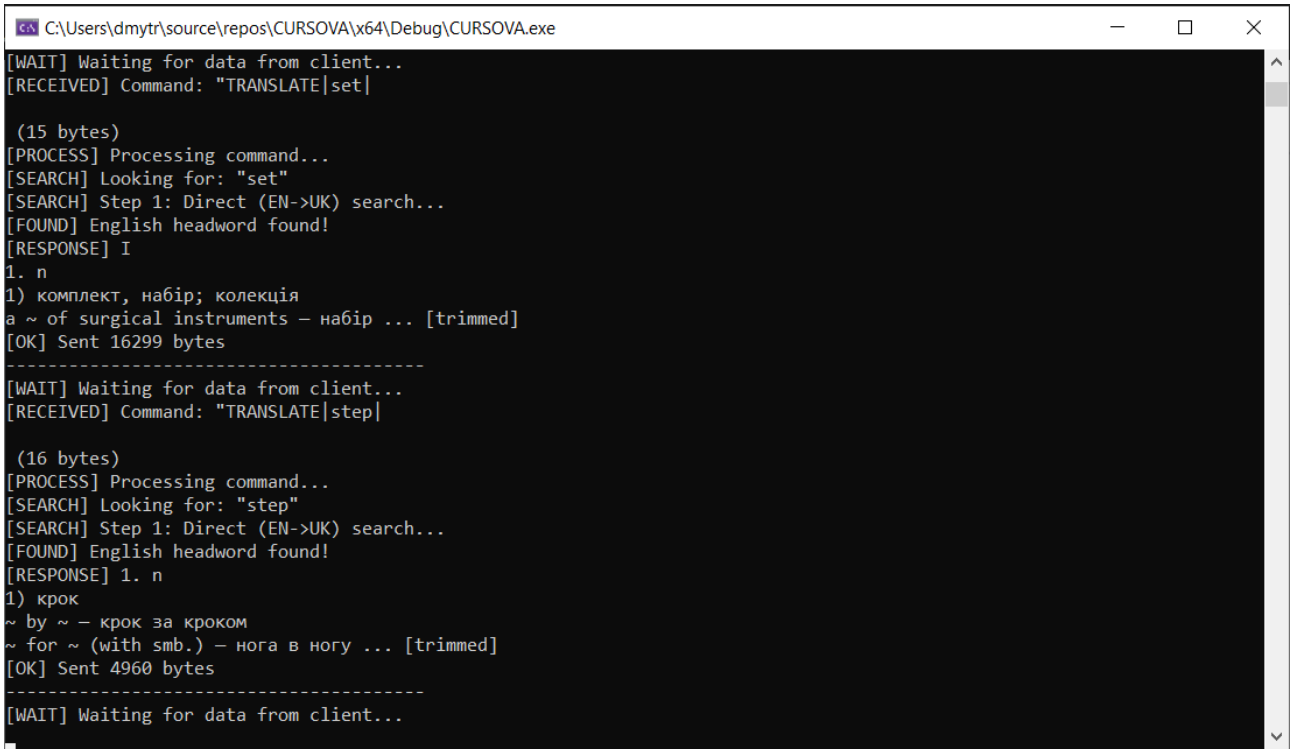


Рисунок 5.2.2.1 – Результат пошуку слова

- **Пошук неіснуючого слова:** Система повертає повідомлення "Not Found" без збоїв.
- **Форматування:** Теги частин мови (наприклад, [NOUN]) коректно підсвічуються зеленим кольором.

5.2.3. Перевірка додавання даних (CRUD - Create)

- **Успішне додавання:** Нове слово миттєво стає доступним для пошуку. Файл словника на сервері оновлюється.

- **Валідація:** Система забороняє додавання слів з пустим перекладом або дублікатів.

5.2.4. Перевірка локального збереження

- Історія пошуку успішно зберігається у файл dictionary_history.db і відновлюється після перезапуску програми.

5.2.5. Висновок за результатами тестування

Проведені експерименти показали, що розроблена система E-Dictionary Pro:

- Стабільно працює в локальному середовищі.
- Забезпечує високу швидкість пошуку.
- Коректно обробляє помилки вводу та мережі.
- Має зручний та інтуїтивно зрозумілий інтерфейс.

ВИСНОВКИ

У результаті виконання курсової роботи було спроєктовано та реалізовано клієнт-серверну систему E-Dictionary Pro — програмний продукт для електронного перекладу та ведення словникової бази. Розв’язано повний життєвий цикл роботи з лексичними даними: пошук, відображення результатів та додавання нових термінів у базу.

Архітектурно рішення поєднує продуктивну серверну частину на C++ з клієнтським додатком на Python. Серверна реалізація використовує контейнер `std::map` для гарантування логарифмічної складності пошуку, що забезпечує прогнозовану швидкість відповіді при зростанні обсягу словникових даних. Мережевий шар на базі Winsock2 реалізує надійну каналізацію передачі даних у середовищі Windows, а інструменти збірки (CMake, vcpkg) та бібліотека `nlohmann/json` забезпечують відтворюваність збірки й простий обмін даними у форматі JSON.

Клієнтська частина, розроблена на Python з використанням CustomTkinter, надала можливість швидкого створення сучасного й зручного графічного інтерфейсу. Інтерфейс орієнтований на типові сценарії роботи зі словником: пошук термінів,

перегляд результатів, додавання та редагування записів, перегляд історії запитів. Для локального збереження використано SQLite, що забезпечує компактність сховища і зручність для однокористувацьких сценаріїв.

Під час розробки вирішено кілька технічних задач: впроваджено власний текстовий протокол для обміну через TCP з чіткими правилами фреймінгу і термінування; реалізовано механізми обробки фрагментації мережевих пакетів, щоб уникнути некоректного декодування UTF-8 на межах повідомлень; реалізовано толерантне накопичення байтів і поетапну спробу декодування, що дозволяє повертати часткові результати замість блокування інтерфейсу. Мережеві операції винесено в фон (фонова потокова модель з блокуванням доступу до сокета), реалізовано таймаути й обробку помилок, що захистило UI від зависань і непередбачених аварійних завершень.

Тестування включало модульні тести серверних репозиторіїв і ручну інтеграційну перевірку взаємодії між сервером і клієнтом. Перевірено коректність CRUD-операцій, застосування бізнес-правил, стійкість при перезапусках сервера та при часткових або повільних відповідях. Результати показали стабільний старт сервера, коректне відновлення стану після змін у базі і прийнятну поведінку системи в умовах неповного мережевого вводу.

Проект також підсилює інженерні практики: інкапсуляція мережевої логіки, розділення шарів доступу до даних, захищені операції з сокетами та детальне логування для покращення спостережуваності. Ці рішення підвищують підтримуваність коду й полегшують подальше розширення функціоналу.

E-Dictionary Pro — робочий і педагогічно цінний прототип, що поєднує високопродуктивний сервер на C++ і чутливий до користувача клієнт на Python. Реалізація відповідає вимогам предметної області, демонструє застосування ООП-принципів і забезпечує надійну базу для подальших покращень у напрямку продуктивності, безпеки та функціональності.

ЛІТЕРАТУРНІ ДЖЕРЕЛА

1. Gamma E., Helm R., Johnson R., Vlissides J. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley, 1994. Опис книги –

[Електронний ресурс] // Режим доступу: https://en.wikipedia.org/wiki/Design_Patterns

2. Stroustrup B. **The C++ Programming Language**. Addison-Wesley. Опис книги – [Електронний ресурс] // Режим доступу: https://en.wikipedia.org/wiki/The_C%2B%2B_Programming_Language
3. Lutz M. **Learning Python.** O'Reilly Media, 2013. Опис книги – [Електронний ресурс] // Режим доступу: https://www.oreilly.com/library/view/learning-python-5th/9781449355722/
4. Tanenbaum A. S., Wetherall D. **Computer Networks.** Prentice Hall, 2011. Опис книги – [Електронний ресурс] // Режим доступу: https://en.wikipedia.org/wiki/Computer_Networks_(book
5. **Winsock2 API** — офіційна документація Microsoft. Офіційний сайт – [Електронний ресурс] // Режим доступу: https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2
6. **SQLite** — офіційна документація. Офіційний сайт – [Електронний ресурс] // Режим доступу: https://www.sqlite.org/index.html Документація: https://www.sqlite.org/docs.html
7. **CustomTkinter** — документація бібліотеки для UI на Python. GitHub – [Електронний ресурс] // Режим доступу: https://github.com/TomSchimansky/CustomTkinter Документація: https://customtkinter.tomschimansky.com/documentation/
8. **Python 3.12 Documentation: socket — Low-level networking interface. Офіційний сайт** – [Електронний ресурс] // Режим доступу: https://docs.python.org/3/library/socket.html
9. **C++ Standard Library Reference (STL).** cppreference.com – [Електронний ресурс] // Режим доступу: https://en.cppreference.com/w/cpp/container/map
10. **GitHub** — платформа для хостингу IT-проектів та спільної розробки. Офіційний сайт – [Електронний ресурс] // Режим доступу: https://github.com/

Додатки

Додаток А – Тексти програмних модулів мовою C++

```
// server.cpp
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <winsock2.h>
#include <map>
#include <string>
#include <sstream>
#include <fstream>
#include <vector>
#include <windows.h>
#include <regex>
#include <random>
#include <ctime>
#include <memory>
#include <clocale>
#include "sqlite3.h"

#pragma comment(lib, "ws2_32.lib")

// UTF-8 console print helper using WriteConsoleW
static void printUtf(const std::string& s) {
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE) {
        std::cout << s << std::endl;
        return;
    }
    int wideLen = MultiByteToWideChar(CP_UTF8, 0, s.c_str(), -1, nullptr, 0);
    if (wideLen > 0) {
        std::wstring wbuf(wideLen, L'\0');
        MultiByteToWideChar(CP_UTF8, 0, s.c_str(), -1, &wbuf[0], wideLen);
        if (!wbuf.empty() && wbuf.back() == L'\0') wbuf.pop_back();
        DWORD written = 0;
        WriteConsoleW(hOut, wbuf.c_str(), (DWORD)wbuf.size(), &written, nullptr);
        WriteConsoleW(hOut, L"\n", 1, &written, nullptr);
    } else {
        std::cout << s << std::endl;
    }
}

// ============================================================================
// Helper: remove HTML/XML tags from dictionary entries
// ============================================================================
std::string removeTags(const std::string& text) {
    std::string result = text;

    std::regex brTag("<br\\s*/?>");
    result = std::regex_replace(result, brTag, "\n");

    std::regex htmlTags("<[^>]*>");
    result = std::regex_replace(result, htmlTags, "");

    std::regex dictTags("\\[[^\\]]*\\]");
    result = std::regex_replace(result, dictTags, "");

    std::regex nbspEntity(" ");
    result = std::regex_replace(result, nbspEntity, " ");

    std::regex ampEntity("&amp;");
    result = std::regex_replace(result, ampEntity, "&");

    std::regex ltEntity("&lt;");
```

```
    result = std::regex_replace(result, ltEntity, "<");

    std::regex gtEntity("&gt;");
    result = std::regex_replace(result, gtEntity, ">");

    std::regex quotEntity("&quot;");
    result = std::regex_replace(result, quotEntity, "\"");

    std::regex multipleNewlines("\n{3,}");
    result = std::regex_replace(result, multipleNewlines, "\n\n");

    std::regex multipleSpaces(" {2,}");
    result = std::regex_replace(result, multipleSpaces, " ");

    size_t start = result.find_first_not_of(" \t\n\r");
    size_t end = result.find_last_not_of(" \t\n\r");

    if (start == std::string::npos) {
        return "";
    }

    return result.substr(start, end - start + 1);
}

// ============================================================================
// CLASS: Language
// ============================================================================
class Language {
private:
    std::string code;
    std::string name;
    std::string nativeName;
public:
    Language() : code(""), name(""), nativeName("") {}
    Language(const std::string& code, const std::string& name, const std::string& nativeName = "")
        : code(code), name(name), nativeName(nativeName.empty() ? name : nativeName) {
    }
    Language(const Language& other) : code(other.code), name(other.name), nativeName(other.nativeName) {}
    Language& operator=(const Language& other) { if (this != &other) { code = other.code; name = other.name; nativeName =
other.nativeName; } return *this; }
    std::string getCode() const { return code; }
    std::string getName() const { return name; }
    std::string getNativeName() const { return nativeName; }
    void setCode(const std::string& newCode) { code = newCode; }
    void setName(const std::string& newName) { name = newName; }
    void setNativeName(const std::string& newNativeName) { nativeName = newNativeName; }
    std::string toString() const { return code + " (" + name + ")"; }
    bool operator==(const Language& other) const { return code == other.code; }
    bool operator!=(const Language& other) const { return !(*this == other); }
};

// ============================================================================
// INTERFACE: IDictionarySource
// ============================================================================
class IDictionarySource {
public:
    virtual ~IDictionarySource() = default;
    virtual std::string search(const std::string& word) = 0;
    virtual void addWord(const std::string& word, const std::string& translation) = 0;
    virtual bool wordExists(const std::string& word) = 0;
    virtual bool updateWord(const std::string& word, const std::string& newTranslation) = 0;
    virtual bool deleteWord(const std::string& word) = 0;
    virtual size_t getSize() const = 0;
    virtual std::string getRandomWord() = 0;
    virtual std::string getSourceName() const {
        return "Unknown dictionary source";
    }
```

```
};

// ============================================================================
// CLASS: Logger
// ============================================================================
class Logger {
private:
    std::ofstream logFile;
    std::string filename;
    bool enabled;
public:
    Logger(const std::string& logFilename = "server_log.txt") : filename(logFilename), enabled(true) {
        logFile.open(filename, std::ios::app);
        if (!logFile.is_open()) {
            std::cerr << "[WARNING] Failed to open log file: " << filename << std::endl;
            enabled = false;
        }
    }
    ~Logger() { if (logFile.is_open()) logFile.close(); }
    void log(const std::string& message) {
        if (enabled && logFile.is_open()) {
            time_t now = time(nullptr);
            char timestamp[64];
            strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", localtime(&now));
            logFile << "[" << timestamp << "] " << message << std::endl;
            logFile.flush();
        }
    }
    void setEnabled(bool value) { enabled = value; }
    bool isEnabled() const { return enabled; }
};

// ============================================================================
// CLASS: SQLiteDictionary
// ============================================================================
class SQLiteDictionary : public IDictionarySource {
private:
    sqlite3* db;
    std::string dbPath;
    Logger& logger;
    mutable std::mt19937 rng;

    bool isWordBoundary(const std::string& str, size_t pos) const {
        if (pos >= str.length()) return true;
        unsigned char c = static_cast<unsigned char>(str[pos]);
        if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) return false;
        if (c == 0xD0 || c == 0xD1) return false;
        if ((c & 0xC0) == 0x80) return false;
        return true;
    }

    bool isWholeWordMatch(const std::string& text, const std::string& query, size_t pos) const {
        if (pos > 0) {
            size_t prevPos = pos - 1;
            while (prevPos > 0 && (static_cast<unsigned char>(text[prevPos]) & 0xC0) == 0x80) prevPos--;
            if (!isWordBoundary(text, prevPos)) return false;
        }
        size_t afterPos = pos + query.length();
        if (afterPos < text.length()) { if (!isWordBoundary(text, afterPos)) return false; }
        return true;
    }

    size_t findWholeWord(const std::string& text, const std::string& query) const {
        size_t pos = 0;
        while ((pos = text.find(query, pos)) != std::string::npos) {
            if (isWholeWordMatch(text, query, pos)) return pos;
            pos++;
```

```
        }
        return std::string::npos;
    }

    std::string extractRedirectWord(const std::string& definition) const {
        size_t startPos = definition.find("<<");
        if (startPos == std::string::npos) return "";
        size_t endPos = definition.find(">>", startPos);
        if (endPos == std::string::npos) return "";
        std::string redirectWord = definition.substr(startPos + 2, endPos - startPos - 2);
        size_t first = redirectWord.find_first_not_of(" \t\n\r");
        size_t last = redirectWord.find_last_not_of(" \t\n\r");
        if (first == std::string::npos) return "";
        return redirectWord.substr(first, last - first + 1);
    }

    bool isRedirectDefinition(const std::string& definition) const {
        if (definition.find("<<") != std::string::npos && definition.find(">>") != std::string::npos) {
            if (definition.find("\xD0\xB4\xD0\xB8\xD0\xB2.") != std::string::npos ||
                definition.find("\xD0\x94\xD0\xB8\xD0\xB2.") != std::string::npos ||
                definition.find("\xD0\x94\xD0\x98\xD0\x92.") != std::string::npos) {
                return true;
            }
            size_t start = definition.find("<<");
            size_t end = definition.find(">>") + 2;
            std::string remaining = definition.substr(0, start) + definition.substr(end);
            size_t first = remaining.find_first_not_of(" \t\n\r.,;:");
            if (first == std::string::npos || remaining.length() - first < 10) return true;
        }
        return false;
    }

    std::string searchInternal(const std::string& query, int depth) {
        if (depth > 2) {
            logger.log("WARNING: Max redirect depth reached for: " + query);
            return "MAX_REDIRECT_DEPTH";
        }
        if (!db) return "DATABASE_ERROR";

        sqlite3_stmt* stmt;
        const char* sql = "SELECT m FROM word WHERE w = ? COLLATE NOCASE LIMIT 1;";
        int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
        if (rc != SQLITE_OK) return "";
        sqlite3_bind_text(stmt, 1, query.c_str(), -1, SQLITE_STATIC);
        std::string result = "";
        rc = sqlite3_step(stmt);
        if (rc == SQLITE_ROW) {
            const unsigned char* rawDef = sqlite3_column_text(stmt, 0);
            if (rawDef) result = reinterpret_cast<const char*>(rawDef);
        }
        sqlite3_finalize(stmt);
        return result;
    }

public:
    SQLiteDictionary(const std::string& databasePath, Logger& log) : db(nullptr), dbPath(databasePath), logger(log) {
        std::random_device rd;
        std::seed_seq seed{ rd(), rd(), rd(), rd(), rd(), rd(), rd(), rd() };
        rng.seed(seed);
        int rc = sqlite3_open(dbPath.c_str(), &db);
        if (rc != SQLITE_OK) {
            std::cerr << "[ERROR] Cannot open database: " << sqlite3_errmsg(db) << std::endl;
            logger.log("ERROR: Cannot open database - " + std::string(sqlite3_errmsg(db)));
            db = nullptr;
        }
        else {
            std::cout << "[INFO] Database connected: " << dbPath << std::endl;
```

```
            logger.log("INFO: Database connected");
        }
    }

    ~SQLiteDictionary() override { if (db) { sqlite3_close(db); logger.log("INFO: Database connection closed"); } }
    SQLiteDictionary(const SQLiteDictionary&) = delete;
    SQLiteDictionary& operator=(const SQLiteDictionary&) = delete;

    std::string search(const std::string& word) override {
        std::cout << "[SEARCH] Looking for: \"" << word << "\"" << std::endl;
        if (!db) {
            std::cerr << "[ERROR] Database not connected!" << std::endl;
            logger.log("ERROR: Database not connected for query: " + word);
            return "DATABASE_ERROR";
        }

        sqlite3_stmt* stmt;
        std::cout << "[SEARCH] Step 1: Direct (EN->UK) search..." << std::endl;
        const char* sql = "SELECT m FROM word WHERE w = ? COLLATE NOCASE LIMIT 1;";
        int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
        if (rc != SQLITE_OK) {
            std::cerr << "[ERROR] SQL prepare error: " << sqlite3_errmsg(db) << std::endl;
            logger.log("ERROR: SQL prepare failed for: " + word);
            return "DATABASE_ERROR";
        }
        sqlite3_bind_text(stmt, 1, word.c_str(), -1, SQLITE_STATIC);

        std::string result = "";
        std::string rawResult = "";
        rc = sqlite3_step(stmt);
        if (rc == SQLITE_ROW) {
            const unsigned char* rawDef = sqlite3_column_text(stmt, 0);
            if (rawDef) {
                rawResult = reinterpret_cast<const char*>(rawDef);
                result = removeTags(rawResult);
                std::cout << "[FOUND] English headword found!" << std::endl;
                logger.log("SEARCH: '" + word + "' -> FOUND (english key)");
            }
        }
        sqlite3_finalize(stmt);

        if (!result.empty()) {
            if (isRedirectDefinition(rawResult)) {
                std::string redirectWord = extractRedirectWord(rawResult);
                if (!redirectWord.empty() && redirectWord != word) {
                    std::cout << "[REDIRECT] Found redirect to: \"" << redirectWord << "\"" << std::endl;
                    logger.log("REDIRECT: '" + word + "' -> '" + redirectWord + "'");
                    std::string redirectRaw = searchInternal(redirectWord, 1);
                    if (!redirectRaw.empty() && redirectRaw != "DATABASE_ERROR" && redirectRaw !=
"MAX_REDIRECT_DEPTH") {
                        std::string redirectResult = removeTags(redirectRaw);
                        if (isRedirectDefinition(redirectRaw)) {
                            std::string secondRedirect = extractRedirectWord(redirectRaw);
                            if (!secondRedirect.empty() && secondRedirect != redirectWord) {
                                std::string secondRaw = searchInternal(secondRedirect, 2);
                                if (!secondRaw.empty() && secondRaw != "DATABASE_ERROR" && secondRaw !=
"MAX_REDIRECT_DEPTH") {
                                    redirectResult = removeTags(secondRaw);
                                    redirectWord = secondRedirect;
                                }
                            }
                        }
                        result = redirectResult + "\n\n(See: " + redirectWord + ")";
                        std::cout << "[RESOLVED] Redirect resolved" << std::endl;
                        logger.log("RESOLVED: '" + word + "' -> '" + redirectWord + "'");
                    }
                }
            }
```

```
        }
        return result;
    }

    // Step 2: Reverse search (UK->EN)
    std::cout << "[SEARCH] Step 2: Reverse search (whole word matching)..." << std::endl;
    logger.log("SEARCH: '" + word + "' -> Reverse search attempt");

    const char* reverseSql = "SELECT w, m FROM word WHERE m LIKE ? LIMIT 100;";
    rc = sqlite3_prepare_v2(db, reverseSql, -1, &stmt, nullptr);
    if (rc != SQLITE_OK) {
        logger.log("ERROR: SQL prepare failed for reverse search: " + word);
        return "NOT_FOUND";
    }

    std::string searchPattern = "%" + word + "%";
    sqlite3_bind_text(stmt, 1, searchPattern.c_str(), -1, SQLITE_STATIC);

    std::string bestMatch = "";
    std::string bestEngWord = "";
    size_t bestPosition = std::string::npos;

    while ((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        const unsigned char* engWord = sqlite3_column_text(stmt, 0);
        const unsigned char* rawDef = sqlite3_column_text(stmt, 1);
        if (engWord && rawDef) {
            std::string engWordStr = reinterpret_cast<const char*>(engWord);
            std::string rawStr = reinterpret_cast<const char*>(rawDef);
            if (isRedirectDefinition(rawStr)) continue;
            size_t matchPos = findWholeWord(rawStr, word);
            if (matchPos != std::string::npos) {
                std::string cleanDef = removeTags(rawStr);
                if (bestMatch.empty() || matchPos < bestPosition) {
                    bestMatch = cleanDef;
                    bestEngWord = engWordStr;
                    bestPosition = matchPos;
                    if (matchPos == 0) break;
                }
            }
        }
    }
    sqlite3_finalize(stmt);

    if (!bestMatch.empty()) {
        result = bestEngWord + "|" + bestMatch;
        std::cout << "[FOUND] Whole-word match! English: \"" << bestEngWord << "\"" << std::endl;
        logger.log("SEARCH: '" + word + "' -> FOUND (reverse: " + bestEngWord + ")");
    }
    else {
        std::cout << "[NOT_FOUND] No matches found" << std::endl;
        logger.log("SEARCH: '" + word + "' -> NOT_FOUND");
        result = "NOT_FOUND";
    }

    return result;
}

void addWord(const std::string& word, const std::string& translation) override {
    if (!db) { logger.log("ERROR: DB not connected for ADD"); return; }
    sqlite3_stmt* stmt;
    const char* sql = "INSERT INTO word (w, m) VALUES (?, ?);";
    int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
    if (rc != SQLITE_OK) { std::cerr << "[ERROR] SQL prepare error: " << sqlite3_errmsg(db) << std::endl;
logger.log("ERROR: Failed to add word: " + word); return; }
    sqlite3_bind_text(stmt, 1, word.c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_text(stmt, 2, translation.c_str(), -1, SQLITE_STATIC);
    rc = sqlite3_step(stmt);
```

```
        if (rc != SQLITE_DONE) { std::cerr << "[ERROR] Insert error: " << sqlite3_errmsg(db) << std::endl;
logger.log("ERROR: Insert failed for: " + word); }
        else { std::cout << "[LOG] Word added: " << word << std::endl; logger.log("ADD: '" + word + "' added"); }
        sqlite3_finalize(stmt);
    }

    bool wordExists(const std::string& word) override {
        if (!db) return false;
        sqlite3_stmt* stmt;
        const char* sql = "SELECT 1 FROM word WHERE w = ? COLLATE NOCASE LIMIT 1;";
        int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
        if (rc != SQLITE_OK) return false;
        sqlite3_bind_text(stmt, 1, word.c_str(), -1, SQLITE_STATIC);
        bool exists = (sqlite3_step(stmt) == SQLITE_ROW);
        sqlite3_finalize(stmt);
        logger.log(std::string("EXISTS: '") + word + "' -> " + (exists ? "YES" : "NO"));
        return exists;
    }

    bool updateWord(const std::string& word, const std::string& newTranslation) override {
        if (!db) { logger.log("ERROR: DB not connected for UPDATE"); return false; }
        if (!wordExists(word)) { logger.log("UPDATE: Word '" + word + "' not found"); return false; }
        sqlite3_stmt* stmt;
        const char* sql = "UPDATE word SET m = ? WHERE w = ? COLLATE NOCASE;";
        int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
        if (rc != SQLITE_OK) { logger.log("ERROR: Failed to prepare UPDATE for: " + word); return false; }
        sqlite3_bind_text(stmt, 1, newTranslation.c_str(), -1, SQLITE_STATIC);
        sqlite3_bind_text(stmt, 2, word.c_str(), -1, SQLITE_STATIC);
        rc = sqlite3_step(stmt);
        sqlite3_finalize(stmt);
        if (rc != SQLITE_DONE) { logger.log("ERROR: Update failed for: " + word); return false; }
        int changes = sqlite3_changes(db);
        if (changes > 0) { std::cout << "[LOG] Word updated: " << word << std::endl; logger.log("UPDATE: '" + word + "'
updated"); return true; }
        return false;
    }

    bool deleteWord(const std::string& word) override {
        if (!db) { logger.log("ERROR: DB not connected for DELETE"); return false; }
        if (!wordExists(word)) { logger.log("DELETE: Word '" + word + "' not found"); return false; }
        sqlite3_stmt* stmt;
        const char* sql = "DELETE FROM word WHERE w = ? COLLATE NOCASE;";
        int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
        if (rc != SQLITE_OK) { logger.log("ERROR: Failed to prepare DELETE for: " + word); return false; }
        sqlite3_bind_text(stmt, 1, word.c_str(), -1, SQLITE_STATIC);
        rc = sqlite3_step(stmt);
        sqlite3_finalize(stmt);
        if (rc != SQLITE_DONE) { logger.log("ERROR: Delete failed for: " + word); return false; }
        int changes = sqlite3_changes(db);
        if (changes > 0) { std::cout << "[LOG] Word deleted: " << word << std::endl; logger.log("DELETE: '" + word + "'
deleted"); return true; }
        return false;
    }

    size_t getSize() const override {
        if (!db) return 0;
        sqlite3_stmt* stmt;
        const char* sql = "SELECT COUNT(*) FROM word;";
        if (sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr) != SQLITE_OK) return 0;
        size_t count = 0;
        if (sqlite3_step(stmt) == SQLITE_ROW) count = static_cast<size_t>(sqlite3_column_int64(stmt, 0));
        sqlite3_finalize(stmt);
        return count;
    }

    std::string getRandomWord() override {
        if (!db) { logger.log("ERROR: DB not connected for GET_RANDOM"); return "DATABASE_ERROR"; }
```

```
        size_t totalWords = getSize();
        if (totalWords == 0) { logger.log("ERROR: Dictionary empty"); return "EMPTY_DICTIONARY"; }
        const int MAX_ATTEMPTS = 5;
        for (int attempt = 0; attempt < MAX_ATTEMPTS; attempt++) {
            std::uniform_int_distribution<size_t> dist(0, totalWords - 1);
            size_t randomOffset = dist(rng);
            std::cout << "[RANDOM] Attempt " << (attempt + 1) << ": offset " << randomOffset << " of " << totalWords <<
std::endl;
            sqlite3_stmt* stmt = nullptr;
            std::string sql = "SELECT w, m FROM word LIMIT 1 OFFSET " + std::to_string(randomOffset) + ";";
            int rc = sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr);
            if (rc != SQLITE_OK) { logger.log("ERROR: SQL prepare error for random word"); return "DATABASE_ERROR";
}
            rc = sqlite3_step(stmt);
            if (rc != SQLITE_ROW) { sqlite3_finalize(stmt); continue; }
            const unsigned char* word = sqlite3_column_text(stmt, 0);
            const unsigned char* definition = sqlite3_column_text(stmt, 1);
            if (!word || !definition) { sqlite3_finalize(stmt); continue; }
            std::string wordStr = reinterpret_cast<const char*>(word);
            std::string rawDefStr = reinterpret_cast<const char*>(definition);
            sqlite3_finalize(stmt);
            if (isRedirectDefinition(rawDefStr)) {
                std::string redirectWord = extractRedirectWord(rawDefStr);
                if (!redirectWord.empty()) {
                    std::string redirectDef = searchInternal(redirectWord, 1);
                    if (!redirectDef.empty() && redirectDef != "DATABASE_ERROR" && redirectDef !=
"MAX_REDIRECT_DEPTH" && !isRedirectDefinition(redirectDef)) {
                        std::string cleanDef = removeTags(redirectDef);
                        std::string result = wordStr + "|" + cleanDef + "\n\n(See: " + redirectWord + ")";
                        std::cout << "[RANDOM] Word of the day (via redirect): " << wordStr << std::endl;
                        logger.log("RANDOM_WORD: '" + wordStr + "' selected (redirect to " + redirectWord + ")");
                        return result;
                    }
                }
                continue;
            }
            std::string defStr = removeTags(rawDefStr);
            if (defStr.empty()) continue;
            std::string result = wordStr + "|" + defStr;
            std::cout << "[RANDOM] Word of the day: " << wordStr << std::endl;
            logger.log("RANDOM_WORD: '" + wordStr + "' selected");
            return result;
        }
        logger.log("ERROR: Random word selection failed after max attempts");
        return "NOT_FOUND";
    }

    std::string getSourceName() const override { return "SQLite Dictionary: " + dbPath; }
    bool isConnected() const { return db != nullptr; }
};

// ============================================================================
// CLASS: Translator
// ============================================================================
class Translator {
private:
    IDictionarySource* dictionary;
    Language sourceLanguage;
    Language targetLanguage;
public:
    Translator(IDictionarySource* dict, const Language& source, const Language& target)
        : dictionary(dict), sourceLanguage(source), targetLanguage(target) {
        std::cout << "[INFO] Translator initialized: " << sourceLanguage.toString() << " -> " << targetLanguage.toString() <<
std::endl;
    }
    Translator(const Translator&) = delete;
    Translator& operator=(const Translator&) = delete;
```

```
    std::string translate(const std::string& query) { return dictionary->search(query); }

    // In-memory dictionary cache stored as word -> definition
    static std::map<std::string, std::string> memoryDictionary;
    static const std::string memoryDictionaryPath;

    // Load dictionary from file into memoryDictionary (best-effort, non-fatal)
    static void loadDictionaryFromFile() {
        try {
            std::ifstream ifs(memoryDictionaryPath);
            if (!ifs.is_open()) return;
            std::string line;
            while (std::getline(ifs, line)) {
                if (line.empty()) continue;
                size_t sep = line.find('|');
                if (sep == std::string::npos) continue;
                std::string word = line.substr(0, sep);
                std::string def = line.substr(sep + 1);
                memoryDictionary[word] = def;
            }
            ifs.close();
        } catch (const std::exception& ex) {
            std::cerr << "[WARNING] Failed to load memory dictionary: " << ex.what() << std::endl;
        }
    }

    // Save memoryDictionary to file (overwrite/truncate) using safe replace.
    // Non-fatal on error.
    static void saveDictionaryToFile() {
        try {
            std::string tempPath = memoryDictionaryPath + ".tmp";
            std::ofstream ofs(tempPath, std::ios::out | std::ios::trunc);
            if (!ofs.is_open()) {
                std::cerr << "[ERROR] Could not open " << tempPath << " for writing" << std::endl;
                return;
            }
            for (const auto& kv : memoryDictionary) {
                ofs << kv.first << '|' << kv.second << '\n';
            }
            ofs.close();

            // Atomically replace the original file (Windows API)
            // Uses MOVEFILE_REPLACE_EXISTING to overwrite if present.
            if (!MoveFileExA(tempPath.c_str(), memoryDictionaryPath.c_str(), MOVEFILE_REPLACE_EXISTING)) {
                std::cerr << "[ERROR] Failed to replace " << memoryDictionaryPath << " with " << tempPath << " (MoveFileExA
error: " << GetLastError() << ")" << std::endl;
                // attempt cleanup
                ::DeleteFileA(tempPath.c_str());
            }
        } catch (const std::exception& ex) {
            std::cerr << "[ERROR] Failed to save memory dictionary: " << ex.what() << std::endl;
        }
    }

    // Append a single word to the dictionary file immediately (safe append).
    static void appendWordToFile(const std::string& word, const std::string& def) {
        try {
            std::ofstream ofs(memoryDictionaryPath, std::ios::out | std::ios::app);
            if (!ofs.is_open()) {
                std::cerr << "[ERROR] Could not open " << memoryDictionaryPath << " for appending" << std::endl;
                return;
            }
            ofs << word << '|' << def << '\n';
            ofs.close();
        } catch (const std::exception& ex) {
            std::cerr << "[ERROR] Failed to append to memory dictionary: " << ex.what() << std::endl;
        }
```

```
    }

    std::string processCommand(const std::string& command) {
        std::istringstream iss(command);
        std::string cmd, arg1, arg2;

        std::getline(iss, cmd, '|');
        std::getline(iss, arg1, '|');
        std::getline(iss, arg2, '|');

        if (cmd == "TRANSLATE") return translate(arg1);
        else if (cmd == "ADD" || cmd == "ADD_WORD") {
            if (arg1.empty()) return "Error|Headword cannot be empty";
            if (arg2.empty()) return "Error|Definition cannot be empty";

            // 1) Check in-memory dictionary first
            auto it = memoryDictionary.find(arg1);
            if (it != memoryDictionary.end()) return "Error|Word already exists";

            // 2) Check persistent backend BEFORE adding to avoid duplicates
            std::string backendSearch = dictionary->search(arg1);
            if (!backendSearch.empty() && backendSearch != "NOT_FOUND" && backendSearch != "DATABASE_ERROR"
&& backendSearch != "MAX_REDIRECT_DEPTH") {
                return "Error|Word already exists";
            }
            if (dictionary->wordExists(arg1)) return "Error|Word already exists";

            // 3) Update Memory (RAM) for instant search
            memoryDictionary[arg1] = arg2;

            // 4) FORCE WRITE TO SQLITE DB (direct raw sqlite3 usage)
            sqlite3* db = nullptr;
            int rc = sqlite3_open("eng_ukr_dictionary.db", &db);

            bool dbOk = false;
            if (rc == SQLITE_OK && db) {
                const char* sql = "INSERT INTO word (w, m) VALUES (?, ?);";
                sqlite3_stmt* stmt = nullptr;

                if (sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr) == SQLITE_OK) {
                    // Bind parameters to prevent SQL injection issues
                    sqlite3_bind_text(stmt, 1, arg1.c_str(), -1, SQLITE_STATIC);
                    sqlite3_bind_text(stmt, 2, arg2.c_str(), -1, SQLITE_STATIC);

                    if (sqlite3_step(stmt) == SQLITE_DONE) {
                        std::cout << "[SUCCESS] Word written to DB: " << arg1 << std::endl;
                        dbOk = true;
                    } else {
                        std::cerr << "[ERROR] DB Insert failed: " << sqlite3_errmsg(db) << std::endl;
                    }
                    sqlite3_finalize(stmt);
                } else {
                    std::cerr << "[ERROR] SQL Prepare failed: " << sqlite3_errmsg(db) << std::endl;
                }
                sqlite3_close(db);
            } else {
                std::cerr << "[CRITICAL] Cannot open eng_ukr_dictionary.db: " << (db ? sqlite3_errmsg(db) : "open failed") <<
std::endl;
                if (db) sqlite3_close(db);
            }

            // If DB write failed, rollback in-memory change to avoid divergence
            if (!dbOk) {
                memoryDictionary.erase(arg1);
                return "Error|Failed to persist to database";
            }
```

```
            // Success
            return "Success|Word added";
        }
        else if (cmd == "UPDATE_WORD") {
            // New behaviour: update in-memory map and persist to dictionary.txt
            if (arg1.empty() || arg2.empty()) return "Error|Headword and definition required";
            auto it = memoryDictionary.find(arg1);
            if (it != memoryDictionary.end()) {
                it->second = arg2;
                saveDictionaryToFile();
                return "Success|Word updated.";
            }
            // Fallback: try updating persistent dictionary via IDataSource
            if (dictionary->updateWord(arg1, arg2)) return "Success|Word updated: " + arg1;
            return "Error|Word not found.";
        }
        else if (cmd == "DELETE_WORD") {
            if (arg1.empty()) return "Error|Headword required";
            auto it = memoryDictionary.find(arg1);
            if (it != memoryDictionary.end()) {
                memoryDictionary.erase(it);
                saveDictionaryToFile();
                return "Success|Word deleted.";
            }
            // Fallback to persistent backend
            if (dictionary->deleteWord(arg1)) return "Success|Word deleted: " + arg1;
            return "Error|Word not found.";
        }
        else if (cmd == "EXISTS") return dictionary->wordExists(arg1) ? "YES" : "NO";
        else if (cmd == "PING") return "PONG";
        else if (cmd == "GET_RANDOM") return dictionary->getRandomWord();
        else if (cmd == "GET_SIZE") return std::to_string(dictionary->getSize());
        else if (cmd == "GET_LANGUAGES") return sourceLanguage.getCode() + "|" + targetLanguage.getCode();
        return "UNKNOWN_COMMAND";
    }

    Language getSourceLanguage() const { return sourceLanguage; }
    Language getTargetLanguage() const { return targetLanguage; }
    void setSourceLanguage(const Language& lang) { sourceLanguage = lang; }
    void setTargetLanguage(const Language& lang) { targetLanguage = lang; }
    void swapLanguages() { std::swap(sourceLanguage, targetLanguage); std::cout << "[INFO] Languages swapped: " <<
sourceLanguage.toString() << " -> " << targetLanguage.toString() << std::endl; }
    IDictionarySource* getDictionary() const { return dictionary; }
};

// ============================================================================
// CLASS: Server
// ============================================================================
class Server {
private:
    SOCKET listenSocket;
    std::string ipAddress;
    int port;
    bool running;
    Translator& translator;
public:
    Server(Translator& trans, const std::string& ip = "127.0.0.1", int p = 8080) : listenSocket(INVALID_SOCKET),
ipAddress(ip), port(p), running(false), translator(trans) {}
    ~Server() { stop(); }
    bool start() {
        WSADATA wsaData; int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData); if (iResult != 0) { std::cerr <<
"[ERROR] WSAStartup failed: " << iResult << std::endl; return false; }
        std::cout << "[OK] WinSock initialized" << std::endl;
        listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); if (listenSocket == INVALID_SOCKET) {
std::cerr << "[ERROR] Socket creation failed: " << WSAGetLastError() << std::endl; WSACleanup(); return false; }
        std::cout << "[OK] Socket created" << std::endl;
```

```
        struct sockaddr_in serverAddr; serverAddr.sin_family = AF_INET; serverAddr.sin_addr.s_addr =
inet_addr(ipAddress.c_str()); serverAddr.sin_port = htons(port);
        if (bind(listenSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) { std::cerr <<
"[ERROR] Bind failed: " << WSAGetLastError() << std::endl; std::cerr << "[HINT] Port " << port << " may be in use" <<
std::endl; closesocket(listenSocket); WSACleanup(); return false; }
        std::cout << "[OK] Bound to " << ipAddress << ":" << port << std::endl;
        if (listen(listenSocket, SOMAXCONN) == SOCKET_ERROR) { std::cerr << "[ERROR] Listen failed: " <<
WSAGetLastError() << std::endl; closesocket(listenSocket); WSACleanup(); return false; }
        std::cout << "[OK] Server listening" << std::endl; running = true; return true;
    }
    void run() {
        std::cout << std::endl;
        std::cout << "========================================" << std::endl;
        std::cout << "=== Electronic Dictionary Server ===" << std::endl;
        std::cout << "========================================" << std::endl;
        std::cout << "Address: " << ipAddress << ":" << port << std::endl;
        std::cout << "Languages: " << translator.getSourceLanguage().toString() << " <-> " <<
translator.getTargetLanguage().toString() << std::endl;
        std::cout << std::endl;
        while (running) {
            std::cout << "[WAIT] Waiting for client connection..." << std::endl;
            SOCKET clientSocket = accept(listenSocket, NULL, NULL);
            if (clientSocket == INVALID_SOCKET) { if (running) std::cerr << "[ERROR] Accept failed: " <<
WSAGetLastError() << std::endl; continue; }
            handleClient(clientSocket);
        }
    }
    void stop() { running = false; if (listenSocket != INVALID_SOCKET) { closesocket(listenSocket); listenSocket =
INVALID_SOCKET; } WSACleanup(); }
private:
    void handleClient(SOCKET clientSocket) {
        std::cout << std::endl;
        std::cout << "========================================" << std::endl;
        std::cout << "[CONNECTED] Client connected!" << std::endl;
        std::cout << "========================================" << std::endl;
        char recvbuf[4096]; int recvbuflen = sizeof(recvbuf) - 1;
        while (true) {
            std::cout << "[WAIT] Waiting for data from client..." << std::endl;
            int iResult = recv(clientSocket, recvbuf, recvbuflen, 0);
            if (iResult > 0) {
                recvbuf[iResult] = '\0'; std::string receivedData(recvbuf, iResult);
                std::cout << "[RECEIVED] Command: \"";
                printUtf(receivedData);
                std::cout << " (" << iResult << " bytes)" << std::endl;
                std::cout << "[PROCESS] Processing command..." << std::endl;
                std::string response = translator.processCommand(receivedData);
                                        // Переконуємося, що відповідь закінчується новим рядком, щоб клієнт міг
коректно її прочитати
                if (response.empty() || response.back() != '\n') response += '\n';
                std::string displayResponse = response; if (displayResponse.length() > 100) displayResponse =
displayResponse.substr(0, 100) + "... [trimmed]";
                std::cout << "[RESPONSE] ";
                printUtf(displayResponse);
                int sendLen = static_cast<int>(response.length()); int iSendResult = send(clientSocket, response.c_str(), sendLen,
0);
                if (iSendResult == SOCKET_ERROR) { std::cerr << "[ERROR] Send failed: " << WSAGetLastError() <<
std::endl; break; }
                std::cout << "[OK] Sent " << iSendResult << " bytes" << std::endl; std::cout << "----------------------------------------"
<< std::endl;
            }
            else if (iResult == 0) { std::cout << "[DISCONNECTED] Client closed connection" << std::endl; break; }
            else { int error = WSAGetLastError(); if (error == WSAECONNRESET) std::cout << "[DISCONNECTED]
Connection reset by client" << std::endl; else std::cerr << "[ERROR] recv failed: " << error << std::endl; break; }
        }
        closesocket(clientSocket);
        std::cout << "[CLOSED] Client socket closed" << std::endl;
        std::cout << std::endl;
```

```
    }
};

// ============================================================================
// MAIN
// ============================================================================
int main() {
    // Ensure console CP is set to UTF-8 as early as possible (Windows only)
    #ifdef _WIN32
    SetConsoleOutputCP(65001); // Force console output to UTF-8
    SetConsoleCP(65001);       // Force console input to UTF-8
    #endif

    // set locale to system (fallback)
    std::setlocale(LC_ALL, "");

    // Enable ANSI escape sequences
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD dwMode = 0; GetConsoleMode(hOut, &dwMode); SetConsoleMode(hOut, dwMode |
ENABLE_VIRTUAL_TERMINAL_PROCESSING);

    std::cout << "========================================" << std::endl;
    std::cout << "[START] Server initializing..." << std::endl;
    std::cout << "========================================" << std::endl;

    Language english("EN", "English", "English");
    Language ukrainian("UK", "Ukrainian", "Ukrainian");

    std::cout << "[OK] Languages created: " << english.toString() << ", " << ukrainian.toString() << std::endl;

    Logger logger("server_log.txt");
    std::cout << "[OK] Logger initialized" << std::endl;

    std::cout << "[INFO] Loading dictionary..." << std::endl;
    SQLiteDictionary dictionary("eng_ukr_dictionary.db", logger);
    if (!dictionary.isConnected()) { std::cerr << "[ERROR] Failed to connect to database!" << std::endl; std::cout << "Press
Enter to exit..." << std::endl; std::cin.get(); return 1; }
    size_t dictSize = dictionary.getSize();
    std::cout << "[OK] Dictionary loaded: " << dictSize << " entries" << std::endl;
    std::cout << "[OK] Dictionary source: " << dictionary.getSourceName() << std::endl;
    if (dictSize == 0) std::cerr << "[WARNING] Dictionary is empty! Check the database file." << std::endl;

    Translator translator(&dictionary, english, ukrainian);
    std::cout << "[OK] Translator initialized" << std::endl;

    Server server(translator, "127.0.0.1", 8080);
    if (!server.start()) { std::cerr << "[ERROR] Failed to start server!" << std::endl; std::cout << "Press Enter to exit..." <<
std::endl; std::cin.get(); return 1; }

    // Load in-memory dictionary from file (if it exists)
    Translator::loadDictionaryFromFile();

    std::cout << std::endl;
    std::cout << "========================================" << std::endl;
    std::cout << "[DONE] Server started successfully!" << std::endl;
    std::cout << "========================================" << std::endl;
    std::cout << std::endl;

    server.run();
    // Save in-memory dictionary to file on exit
    Translator::saveDictionaryToFile();
    return 0;
}

std::map<std::string, std::string> Translator::memoryDictionary;
const std::string Translator::memoryDictionaryPath = "dictionary.txt";
```

Додаток Б – Тексти програмних модулів мовою Python

```
"""
database_manager.py
Модуль менеджера бази даних для електронного словника.

Містить клас DatabaseManager для роботи з SQLite базою даних.
Обробляє збереження історії пошуку та налаштувань користувача.

Автор: Dmytro Petruniv
Версія: 1.1
Дата: 2025
"""

import sqlite3
import logging
import datetime
from typing import List, Optional, Tuple

# Отримуємо логер
logger = logging.getLogger("DictionaryClient")


class DatabaseManager:
    """
    Менеджер бази даних SQLite для словника.

    Забезпечує збереження історії пошуку, улюблених слів
    та налаштувань користувача.

    Attributes:
        db_path (str): Шлях до файлу бази даних.
        connection (sqlite3.Connection): З'єднання з базою.

    Example:
        >>> db = DatabaseManager('dictionary.db')
        >>> db.add_to_history('hello', 'привіт')
        >>> db.get_history()
        [('hello', 'привіт', '2025-12-11 10:30:00')]
    """

    def __init__(self, db_path: str = 'dictionary_history.db'):
        """
        Ініціалізація менеджера бази даних.

        Args:
            db_path (str): Шлях до файлу бази даних. За замовчуванням
'dictionary_history.db'.
        """
        self.db_path = db_path
        self.connection: Optional[sqlite3.Connection] = None
        self._initialize_database()

    def _initialize_database(self):
        """
        Створення таблиць якщо вони не існують.

        Створює:
        - search_history: Історія пошуку
        - favorites: Улюблені слова
```

```
        - settings: Налаштування користувача
        """
        try:
            self.connection = sqlite3.connect(self.db_path,
check_same_thread=False)
            cursor = self.connection.cursor()

            # Таблиця історії пошуку
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS search_history (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    word TEXT NOT NULL,
                    translation TEXT,
                    searched_at DATETIME DEFAULT CURRENT_TIMESTAMP
                )
            ''')

            # Таблиця улюблених слів
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS favorites (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    word TEXT NOT NULL UNIQUE,
                    translation TEXT,
                    added_at DATETIME DEFAULT CURRENT_TIMESTAMP
                )
            ''')

            # Таблиця налаштувань
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS settings (
                    key TEXT PRIMARY KEY,
                    value TEXT
                )
            ''')

            self.connection.commit()
            logger.info("База даних ініціалізована успішно")

        except sqlite3.Error as e:
            logger.error(f"Помилка ініціалізації бази даних: {e}")
            print(f"[БД] ✗ Помилка ініціалізації: {e}")

    def add_to_history(self, word: str, translation: str = "") -> bool:
        """
        Додає слово до історії пошуку.

        Args:
            word (str): Слово що було знайдено.
            translation (str): Переклад слова (опціонально).

        Returns:
            bool: True якщо успішно додано, False інакше.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                timestamp = datetime.datetime.now()

                # 1. Insert or Replace (оновлює timestamp, якщо слово вже є)
```

```
                cursor.execute("""
                    INSERT OR REPLACE INTO search_history (id, word, translation,
searched_at)
                    VALUES (
                        (SELECT id FROM search_history WHERE word = ?),
                        ?, ?, ?
                    )
                """, (word, word, translation, timestamp))

                # 2. Keep only last 50 items (Виправлений SQL запит)
                cursor.execute("""
                    DELETE FROM search_history
                    WHERE id NOT IN (
                        SELECT id FROM search_history
                        ORDER BY searched_at DESC
                        LIMIT 50
                    )
                """)
                conn.commit()
                logger.info(f"Додано до історії: '{word}'")
                return True
        except sqlite3.Error as e:
            logger.error(f"Помилка додавання до історії: {e}")
            return False

    def get_history(self, limit: int = 10) -> List[Tuple[str, str, str]]:
        """
        Отримує останні записи з історії пошуку.

        Args:
            limit (int): Максимальна кількість записів. За замовчуванням 10.

        Returns:
            List[Tuple[str, str, str]]: Список кортежів (слово, переклад, час).
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('''
                    SELECT word, translation, searched_at
                    FROM search_history
                    ORDER BY searched_at DESC
                    LIMIT ?
                ''', (limit,))
                return cursor.fetchall()
        except sqlite3.Error as e:
            logger.error(f"Помилка отримання історії: {e}")
            return []

    def get_history_words(self, limit: int = 10) -> List[str]:
        """
        Отримує тільки слова з історії (без дублікатів).

        Args:
            limit (int): Максимальна кількість слів.

        Returns:
            List[str]: Список унікальних слів.
        """
```

```
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('''
                    SELECT DISTINCT word
                    FROM search_history
                    ORDER BY searched_at DESC
                    LIMIT ?
                ''', (limit,))
                return [row[0] for row in cursor.fetchall()]
        except sqlite3.Error as e:
            logger.error(f"Помилка отримання слів історії: {e}")
            return []

    def clear_history(self) -> bool:
        """
        Очищає всю історію пошуку.

        Returns:
            bool: True якщо успішно очищено, False інакше.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute("DELETE FROM search_history")
                conn.commit()
                logger.info("Історію пошуку очищено")
                return True
        except sqlite3.Error as e:
            logger.error(f"Помилка очищення історії: {e}")
            return False

    def remove_from_history(self, word: str) -> bool:
        """
        Видаляє конкретне слово з історії пошуку.

        Args:
            word (str): Слово для видалення.

        Returns:
            bool: True якщо успішно видалено, False інакше.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('DELETE FROM search_history WHERE word = ?',
(word,))
                conn.commit()
                deleted = cursor.rowcount > 0
                if deleted:
                    logger.info(f"Видалено з історії: '{word}'")
                return deleted
        except sqlite3.Error as e:
            logger.error(f"Помилка видалення з історії: {e}")
            return False

    def add_to_favorites(self, word: str, translation: str = "") -> bool:
        """
        Додає слово до улюблених.
```

```
        Args:
            word (str): Слово для збереження.
            translation (str): Переклад слова.

        Returns:
            bool: True якщо успішно додано, False якщо вже існує або помилка.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute(
                    'INSERT OR IGNORE INTO favorites (word, translation, added_at)
VALUES (?, ?, ?)',
                    (word, translation, datetime.datetime.now())
                )
                conn.commit()
                if cursor.rowcount > 0:
                    logger.info(f"Додано до улюблених: '{word}'")
                    return True
                return False  # Вже існує
        except sqlite3.Error as e:
            logger.error(f"Помилка додавання до улюблених: {e}")
            return False

    def get_favorites(self) -> List[Tuple[str, str]]:
        """
        Отримує всі улюблені слова.

        Returns:
            List[Tuple[str, str]]: Список кортежів (слово, переклад).
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('SELECT word, translation FROM favorites ORDER BY
added_at DESC')
                return cursor.fetchall()
        except sqlite3.Error as e:
            logger.error(f"Помилка отримання улюблених: {e}")
            return []

    def remove_from_favorites(self, word: str) -> bool:
        """
        Видаляє слово з улюблених.

        Args:
            word (str): Слово для видалення.

        Returns:
            bool: True якщо успішно видалено, False інакше.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('DELETE FROM favorites WHERE word = ?', (word,))
                conn.commit()
                return cursor.rowcount > 0
        except sqlite3.Error as e:
```

```
            logger.error(f"Помилка видалення з улюблених: {e}")
            return False

    def is_favorite(self, word: str) -> bool:
        """
        Перевіряє чи слово є улюбленим.

        Args:
            word (str): Слово для перевірки.

        Returns:
            bool: True якщо слово в улюблених, False інакше.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('SELECT COUNT(*) FROM favorites WHERE word = ?',
(word,))
                result = cursor.fetchone()
                return result[0] > 0 if result else False
        except sqlite3.Error as e:
            logger.error(f"Помилка перевірки улюбленого: {e}")
            return False

    # Alias methods for compatibility with user's requested naming
    def add_favorite(self, word: str, definition: str = "") -> bool:
        """
        Alias for add_to_favorites for compatibility.

        Args:
            word (str): Слово для збереження.
            definition (str): Переклад слова.

        Returns:
            bool: True якщо успішно додано, False інакше.
        """
        return self.add_to_favorites(word, definition)

    def remove_favorite(self, word: str) -> bool:
        """
        Alias for remove_from_favorites for compatibility.

        Args:
            word (str): Слово для видалення.

        Returns:
            bool: True якщо успішно видалено, False інакше.
        """
        return self.remove_from_favorites(word)

    def get_setting(self, key: str, default: str = "") -> str:
        """
        Отримує налаштування за ключем.

        Args:
            key (str): Ключ налаштування.
            default (str): Значення за замовчуванням.

        Returns:
```

```
            str: Значення налаштування або default.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute('SELECT value FROM settings WHERE key = ?', (key,))
                row = cursor.fetchone()
                return row[0] if row else default
        except sqlite3.Error as e:
            logger.error(f"Помилка отримання налаштування: {e}")
            return default

    def set_setting(self, key: str, value: str) -> bool:
        """
        Зберігає налаштування.

        Args:
            key (str): Ключ налаштування.
            value (str): Значення для збереження.

        Returns:
            bool: True якщо успішно збережено, False інакше.
        """
        try:
            with sqlite3.connect(self.db_path) as conn:
                cursor = conn.cursor()
                cursor.execute(
                    'INSERT OR REPLACE INTO settings (key, value) VALUES (?, ?)',
                    (key, value)
                )
                conn.commit()
                return True
        except sqlite3.Error as e:
            logger.error(f"Помилка збереження налаштування: {e}")
            return False

    def close(self):
        """
        Закриває з'єднання з базою даних.
        """
        if self.connection:
            self.connection.close()
            logger.info("З'єднання з базою даних закрито")

    def __del__(self):
        """Деструктор - закриває з'єднання."""
        self.close()
"""
main.py
Точка входу для електронного українсько-англійського словника.

Запускає GUI застосунок з налаштованим клієнтом.

Автор: Dmytro Petruniv
Версія: 2.0
Дата: 2025

Використання:
    python main.py
```

```
"""

import os
import sys
import ctypes
import logging

# --- Підтримка кирилиці в консолі Windows ---
if sys.platform == "win32":
    sys.stdout.reconfigure(encoding='utf-8')
    sys.stderr.reconfigure(encoding='utf-8')
    os.system("chcp 65001 >nul 2>&1")

    # High DPI Awareness для Windows
    try:
        ctypes.windll.shcore.SetProcessDpiAwareness(1)
    except Exception:
        pass

# --- Налаштування логування ---
LOG_DIR = os.path.dirname(os.path.abspath(__file__))
LOG_FILE = os.path.join(LOG_DIR, "client_log.txt")

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s | %(levelname)-8s | %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S',
    handlers=[
        logging.FileHandler(LOG_FILE, encoding='utf-8'),
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger("DictionaryClient")

# --- Налаштування CustomTkinter ---
import customtkinter as ctk
ctk.set_appearance_mode("light")
ctk.set_default_color_theme("blue")

# --- Імпорт модулів застосунку ---
from network_manager import DictionaryClient
from ui_components import ModernDictionaryApp


def main():
    """
    Головна функція запуску застосунку.

    Створює клієнт, передає його в GUI та запускає головний цикл.
    Забезпечує безпечне завершення з очищенням ресурсів.
    """
    client = None
    app = None

    try:
        # Створюємо клієнт для мережевих операцій
        client = DictionaryClient(
            host='127.0.0.1',
            port=8080,
```

```
            timeout=10.0
        )

        # Створюємо та запускаємо GUI з клієнтом
        app = ModernDictionaryApp(client=client)

        # Запускаємо головний цикл
        app.mainloop()

    except KeyboardInterrupt:
        logger.info("Застосунок зупинено користувачем (Ctrl+C)")
    except Exception as e:
        # Ловимо всі неочікувані винятки для безпеки
        logger.error(f"Критична помилка: {e}", exc_info=True)
    finally:
        # Безпечне очищення ресурсів
        try:
            if client:
                if client.connected:
                    try:
                        client.disconnect()
                    except Exception as e:
                        logger.warning(f"Помилка відключення клієнта: {e}")
        except Exception as e:
            logger.warning(f"Помилка очищення клієнта: {e}")

        try:
            if app and hasattr(app, 'db') and app.db:
                try:
                    app.db.close()
                except Exception as e:
                    logger.warning(f"Помилка закриття БД: {e}")
        except Exception as e:
            logger.warning(f"Помилка очищення БД: {e}")

        logger.info("Застосунок завершено")


if __name__ == "__main__":
    main()

"""
network_manager.py
Модуль мережевого менеджера для електронного словника.

Містить клас DictionaryClient для TCP з'єднання з C++ сервером.
Обробляє тільки сокетні з'єднання, відправку запитів та отримання даних.

Автор: Dmytro Petruniv
Версія: 2.1
Дата: 2025
"""

import socket
import time
import logging
import threading

# Отримуємо логер
```

```python
logger = logging.getLogger("DictionaryClient")


class DictionaryClient:
    """
    TCP Клієнт для зв'язку з C++ сервером словника.

    Забезпечує підключення, відправку команд та отримання відповідей
    від сервера через TCP сокети.

    Attributes:
        host (str): IP-адреса сервера.
        port (int): Порт сервера.
        socket (socket.socket): TCP сокет для з'єднання.
        connected (bool): Статус з'єднання.
        timeout (float): Таймаут з'єднання в секундах.

    Example:
        >>> client = DictionaryClient('127.0.0.1', 8080)
        >>> client.connect()
        True
        >>> client.send_command('TRANSLATE|hello|')
        'привіт'
        >>> client.disconnect()
    """

    def __init__(self, host: str = '127.0.0.1', port: int = 8080, timeout: float =
2.0):
        """
        Ініціалізація клієнта словника.

        Args:
            host (str): IP-адреса сервера. За замовчуванням '127.0.0.1'.
            port (int): Порт сервера. За замовчуванням 8080.
            timeout (float): Таймаут з'єднання в секундах. За замовчуванням 2.0.
        """
        self.host = host
        self.port = port
        self.timeout = timeout
        self.socket = None
        self.connected = False
        self._socket_lock = threading.Lock()  # Захист доступу до сокета з різних
потоків

    def connect(self) -> bool:
        """
        Встановлює з'єднання з сервером. Захищено від помилок та ідемпотентно.

        Returns:
            bool: True якщо підключення успішне, False інакше.
        """
        with self._socket_lock:
            # Закриваємо старий сокет якщо є (ідемпотентність)
            if self.socket:
                try:
                    self.socket.close()
                except Exception:
                    pass
                self.socket = None
```

```
            self.connected = False

            try:
                self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.socket.settimeout(self.timeout)
                self.socket.connect((self.host, self.port))
                self.connected = True
                logger.info(f"[КЛІЄНТ] ✅ Підключено до {self.host}:{self.port}")
                return True
            except (ConnectionRefusedError, socket.timeout, OSError) as e:
                logger.error(f"[КЛІЄНТ] ❌ Помилка підключення: {e}")
                self.connected = False
                # Безпечне закриття сокета при помилці
                if self.socket:
                    try:
                        self.socket.close()
                    except Exception:
                        pass
                    self.socket = None
                return False
            except Exception as e:
                # Ловимо всі інші винятки для безпеки
                logger.error(f"[КЛІЄНТ] ❌ Неочікувана помилка підключення: {e}")
                self.connected = False
                if self.socket:
                    try:
                        self.socket.close()
                    except Exception:
                        pass
                    self.socket = None
                return False

    def send_command(self, command: str, recv_chunk: int = 4096) -> str:
        """
        Відправляє команду з надійною обробкою помилок, часткових UTF-8 кадрів та
таймаутів.

        Особливості:
        - Використовує settimeout для send/recv операцій
        - Накопичує байти в список parts та намагається декодувати після кожного
чанка
        - Обробляє UnicodeDecodeError для неповних байтів UTF-8
        - Використовує deadline на основі timeout для переривання циклу
        - Безпечно обробляє помилки сокетів та завжди відновлює попередній timeout

        Args:
            command (str): Команда для відправки
            recv_chunk (int): Розмір буфера для recv. За замовчуванням 4096.

        Returns:
            str: Відповідь сервера (може бути порожнім рядком при помилках)
        """
        max_retries = 3
        base_backoff = 0.5  # Початкова затримка в секундах
        max_backoff = 8.0   # Максимальна затримка

        for attempt in range(max_retries):
            sock = None
            old_timeout = None
```

```
            try:
                with self._socket_lock:
                    # Перевіряємо чи є активне з'єднання
                    if not self.connected or not self.socket:
                        if attempt == 0:
                            logger.info(f"[КЛІЄНТ] Спроба підключення {attempt +
1}/{max_retries}...")
                        else:
                            # Експоненціальний backoff для повторних спроб
                            backoff = min(base_backoff * (2 ** (attempt - 1)),
max_backoff)
                            logger.info(f"[КЛІЄНТ] Retry #{attempt}: спроба
перепідключення (backoff {backoff:.1f}s)...")
                            time.sleep(backoff)

                        if not self.connect():
                            continue

                    sock = self.socket
                    # Зберігаємо поточний timeout для відновлення
                    old_timeout = sock.gettimeout()
                    sock.settimeout(self.timeout)

                # Підготовка команди
                if not command.endswith('\n'):
                    command = command + '\n'
                full_cmd = command.encode('utf-8')

                # Відправка команди
                try:
                    sock.sendall(full_cmd)
                    logger.debug(f"[КЛІЄНТ] Відправлено команду:
{command.strip()}")
                except (BrokenPipeError, ConnectionResetError, OSError) as e:
                    logger.warning(f"[КЛІЄНТ] ⚠ Помилка відправки: {e}")
                    with self._socket_lock:
                        self.connected = False
                        if self.socket:
                            try:
                                self.socket.close()
                            except Exception:
                                pass
                            self.socket = None
                    continue

                # Отримання відповіді з обробкою часткових UTF-8 кадрів
                parts = []  # Список байтів для накопичення
                deadline = time.time() + self.timeout

                while time.time() < deadline:
                    try:
                        chunk = sock.recv(recv_chunk)
                        if not chunk:
                            # Порожній чанк означає закриття з'єднання
                            logger.debug("[КЛІЄНТ] Отримано порожній чанк
(з'єднання закрито)")
                            break
```

```
                        parts.append(chunk)

                        # Спробуємо декодувати накопичені дані
                        raw = b"".join(parts)
                        try:
                            text = raw.decode('utf-8')
                            # Якщо декодування успішне і отримали менше ніж
recv_chunk байт,
                            # або знайдено термінатор протоколу (якщо є),
повертаємо результат
                            if len(chunk) < recv_chunk:
                                # Останній чанк - повертаємо результат
                                result = text.strip()
                                logger.debug(f"[КЛІЄНТ] Отримано відповідь
({len(raw)} байт)")
                                return result
                        except UnicodeDecodeError:
                            # Неповний UTF-8 символ - продовжуємо отримувати дані
                            logger.debug(f"[КЛІЄНТ] Неповний UTF-8 кадр,
продовжуємо отримання...")
                            continue

                    except socket.timeout:
                        # Таймаут при отриманні
                        logger.warning(f"[КЛІЄНТ] ⚠ Таймаут отримання даних")
                        break
                    except (ConnectionResetError, BrokenPipeError, OSError) as e:
                        logger.warning(f"[КЛІЄНТ] ⚠ Помилка отримання: {e}")
                        with self._socket_lock:
                            self.connected = False
                            if self.socket:
                                try:
                                    self.socket.close()
                                except Exception:
                                    pass
                                self.socket = None
                        # Виходимо з циклу отримання, спробуємо перепідключитися
                        break

                # Якщо вийшли з циклу, намагаємося декодувати те, що отримали
                if parts:
                    raw = b"".join(parts)
                    try:
                        # Толерантне декодування з заміною помилкових символів
                        result = raw.decode('utf-8', errors='replace').strip()
                        logger.debug(f"[КЛІЄНТ] Повертаємо часткову відповідь після
таймауту/помилки")
                        return result
                    except Exception as e:
                        logger.warning(f"[КЛІЄНТ] ⚠ Помилка декодування: {e}")
                        return ""
                else:
                    # Нічого не отримано
                    logger.warning(f"[КЛІЄНТ] ⚠ Порожня відповідь від сервера")
                    with self._socket_lock:
                        self.connected = False
                        if self.socket:
                            try:
                                self.socket.close()
```

```
                            except Exception:
                                pass
                            self.socket = None
                        continue

            except (BrokenPipeError, ConnectionResetError, OSError) as e:
                logger.warning(f"[КЛІЄНТ] ⚠ Втрачено з'єднання ({e}).
Перепідключення...")
                with self._socket_lock:
                    self.connected = False
                    if self.socket:
                        try:
                            self.socket.close()
                        except Exception:
                            pass
                        self.socket = None
                continue
            except Exception as e:
                # Ловимо всі інші винятки для безпеки
                logger.error(f"[КЛІЄНТ] ✗ Неочікувана помилка: {e}")
                with self._socket_lock:
                    self.connected = False
                    if self.socket:
                        try:
                            self.socket.close()
                        except Exception:
                            pass
                        self.socket = None
                continue
            finally:
                # Завжди відновлюємо попередній timeout
                if sock and old_timeout is not None:
                    try:
                        sock.settimeout(old_timeout)
                    except Exception:
                        pass

        # Всі спроби вичерпано
        logger.error("[КЛІЄНТ] ✗ Не вдалося відправити запит після всіх спроб.")
        return ""

    def disconnect(self) -> None:
        """
        Безпечно від'єднується від сервера. Ідемпотентна операція.
        """
        with self._socket_lock:
            if self.socket:
                try:
                    self.socket.shutdown(socket.SHUT_RDWR)
                except Exception:
                    pass
                try:
                    self.socket.close()
                except Exception:
                    pass
                self.socket = None
            self.connected = False
            logger.info("[КЛІЄНТ] Від'єднано від сервера")
```

```python
    # Зручні обгортки
    def translate(self, word: str) -> str | None:
        """
        Переклад слова через сервер.

        Args:
            word (str): Слово для перекладу.

        Returns:
            str or None: Переклад або None при помилці.
        """
        return self.send_command(f"TRANSLATE|{word}|")

    def add_word(self, ukrainian: str, english: str) -> str | None:
        """
        Додавання нового слова до словника.

        Args:
            ukrainian (str): Українське слово.
            english (str): Англійський переклад.

        Returns:
            str or None: Відповідь сервера ('ADDED', 'EXIST') або None.
        """
        return self.send_command(f"ADD|{ukrainian}|{english}")

    def delete_word(self, headword: str) -> str | None:
        """
        Видалення слова зі словника.

        Args:
            headword (str): Англійське слово для видалення.

        Returns:
            str or None: Відповідь сервера ('Success' або 'Error') або None.
        """
        return self.send_command(f"DELETE|{headword}|")

    def update_word(self, headword: str, new_definition: str) -> str | None:
        """
        Оновлення визначення слова в словнику.

        Args:
            headword (str): Англійське слово для оновлення.
            new_definition (str): Нове визначення.

        Returns:
            str or None: Відповідь сервера ('Success' або 'Error') або None.
        """
        return self.send_command(f"UPDATE|{headword}|{new_definition}")

    def close(self):
        self.disconnect()

    def is_connected(self) -> bool:
        """
        Перевірка статусу з'єднання.

        Returns:
```

```
            bool: True якщо підключено, False інакше.
        """
        return self.connected

    def set_host(self, host: str) -> None:
        """
        Зміна адреси сервера.

        Args:
            host (str): Нова IP-адреса сервера.
        """
        self.host = host

    def set_port(self, port: int) -> None:
        """
        Зміна порту сервера.

        Args:
            port (int): Новий порт сервера.
        """
        self.port = port

    def __repr__(self) -> str:
        status = "connected" if self.connected else "disconnected"
        return f"DictionaryClient({self.host}:{self.port}, {status})"

"""
ui_components.py
UI компоненти для електронного словника.

Містить всі CustomTkinter класи та логіку інтерфейсу:
- ModernDictionaryApp: Головне вікно застосунку
- StatusIndicator: Індикатор статусу з'єднання
- ResultCard: Картка результату перекладу
- HistoryItem: Елемент історії пошуку

Автор: Dmytro Petruniv
Версія: 2.0
Дата: 2025
"""

import customtkinter as ctk
from tkinter import messagebox
import logging
import re
import html
from datetime import datetime
import threading

from network_manager import DictionaryClient
from database_manager import DatabaseManager

# Отримуємо логер
logger = logging.getLogger("DictionaryClient")

# --- Іконки частин мови (Unicode Emoji) ---
POS_ICONS = {
    "noun": "",       # Іменник
    "verb": "",       # Дієслово
```

```
    "adj": "🎨",        # Прикметник
    "adv": "🚀",        # Прислівник
    "prep": "📎",       # Прийменник
    "conj": "🔀",       # Сполучник
    "pron": "👤",       # Займенник
    "num": "🔢",        # Числівник
    "default": "📖"     # За замовчуванням
}

# --- Палітра кольорів (Світла тема - Light Mode Only) ---
COLORS = {
    "bg_main": "#F5F5F5",           # Світло-сірий фон
    "bg_card": "#FFFFFF",           # Білий для карток
    "bg_sidebar": "#E8E8E8",        # Сірий для бокових панелей
    "accent": "#007AFF",            # Azure Blue акцент
    "accent_hover": "#0056B3",      # Темніший синій при наведенні
    "success": "#28A745",           # Зелений для успіху
    "warning": "#FFC107",           # Жовтий для попереджень
    "danger": "#DC3545",            # Червоний для помилок
    "text_primary": "#000000",      # Чорний основний текст
    "text_secondary": "#333333",    # Темно-сірий вторинний текст
    "text_muted": "#6C757D",        # Сірий для тегів/підказок
    "border": "#DEE2E6",            # Світла рамка
    "title_color": "#1A365D"        # Темно-синій для заголовків
}

# --- Заголовки частин мови для форматування (чисті, без emoji) ---
POS_HEADERS = {
    'n': 'NOUN',
    'noun': 'NOUN',
    'v': 'VERB',
    'verb': 'VERB',
    'adj': 'ADJECTIVE',
    'adjective': 'ADJECTIVE',
    'adv': 'ADVERB',
    'adverb': 'ADVERB',
    'prep': 'PREPOSITION',
    'preposition': 'PREPOSITION',
    'conj': 'CONJUNCTION',
    'conjunction': 'CONJUNCTION',
    'pron': 'PRONOUN',
    'pronoun': 'PRONOUN',
    'int': 'INTERJECTION',
    'interjection': 'INTERJECTION',
    'num': 'NUMERAL',
    'numeral': 'NUMERAL',
    'phrasal v': 'PHRASAL VERB',
    'ph v': 'PHRASAL VERB',
    'ph.v': 'PHRASAL VERB',
}


def insert_formatted_text(textbox, text: str, tag_color: str = "#10B981"):
    """
    Вставляє відформатований текст у CTkTextbox з кольоровими тегами.

    Парсить текст та виділяє:
    - POS headers (наприклад, [ NOUN ], [ VERB ]) - жирним та кольором
```

```
    - Абревіатури (наприклад, [розм.], [книжк.]) - кольором

    Args:
        textbox: CTkTextbox віджет для вставки тексту
        text: Текст для форматування та вставки
        tag_color: Колір для тегів (POS headers та абревіатури). За замовчуванням
зелений.
    """
    if not text:
        return

    # Очищаємо textbox
    textbox.delete("1.0", "end")

    # Створюємо теги для кольорових міток
    tag_name = "colored_tag"
    bold_tag_name = "bold_tag"

    # Налаштовуємо теги (CTkTextbox базується на tkinter Text widget)
    # Отримуємо доступ до внутрішнього Text widget
    inner_text = None
    try:
        # CTkTextbox має атрибут textbox для доступу до tkinter Text
        inner_text = getattr(textbox, 'textbox', None)
        if not inner_text:
            # Спробуємо інші можливі атрибути
            inner_text = getattr(textbox, '_textbox', None)
    except AttributeError:
        pass

    if inner_text:
        # Налаштовуємо теги через внутрішній Text widget
        try:
            import tkinter.font as tkfont
            # Звичайний тег для абревіатур
            inner_text.tag_config(tag_name, foreground=tag_color)
            # Жирний тег для POS headers (POS_TAG: [NOUN], [VERB] etc.)
            bold_font = tkfont.Font(family="Segoe UI", size=14, weight="bold")
            inner_text.tag_config(bold_tag_name, font=bold_font,
foreground=tag_color)
        except Exception as e:
            logger.warning(f"Не вдалося налаштувати теги: {e}")
            # Fallback: використовуємо тільки колір без жирного шрифту
            try:
                inner_text.tag_config(tag_name, foreground=tag_color)
                inner_text.tag_config(bold_tag_name, foreground=tag_color)
            except Exception:
                pass

    # Розбиваємо текст на рядки
    lines = text.split('\n')

    for line in lines:
        if not line.strip():
            # Порожній рядок
            textbox.insert("end", "\n")
            continue

        # Перевіряємо чи це POS header (формат: [ NOUN ], [ VERB ], [ PHRASAL VERB
```

```
] тощо)
        # Може бути з пробілами: [ NOUN ] або без: [NOUN]
        pos_match = re.match(r'^(\s*)(\[)\s*([A-Z][A-Z\s]*[A-Z]|[A-
Z]+)\s*(\])(\s*)$', line)
        if pos_match:
            # Це POS header - виділяємо його жирним та кольором
            prefix = pos_match.group(1)
            bracket_open = pos_match.group(2)
            pos_text = pos_match.group(3).strip()
            bracket_close = pos_match.group(4)
            suffix = pos_match.group(5)

            if prefix:
                textbox.insert("end", prefix)
            textbox.insert("end", bracket_open, bold_tag_name)
            textbox.insert("end", f" {pos_text} ", bold_tag_name)
            textbox.insert("end", bracket_close, bold_tag_name)
            if suffix:
                textbox.insert("end", suffix)
            textbox.insert("end", "\n")
            continue

        # Обробляємо рядок з можливими абревіатурами
        remaining_line = line
        last_end = 0

        # Regex для знаходження всіх квадратних дужок (абревіатури)
        abbr_pattern = r'(\[[^\]]+\])'
        matches = list(re.finditer(abbr_pattern, remaining_line))

        if matches:
            # Є теги - вставляємо по частинах
            for match in matches:
                # Текст до тегу
                before = remaining_line[last_end:match.start()]
                if before:
                    textbox.insert("end", before)

                # Перевіряємо чи це POS header (великі літери) чи абревіатура
                tag_text = match.group(1)
                tag_content = tag_text.strip('[]').strip()

                # Якщо це POS header (тільки великі літери та пробіли)
                if re.match(r'^[A-Z\s]+$', tag_content):
                    # Використовуємо жирний тег
                    textbox.insert("end", tag_text, bold_tag_name)
                else:
                    # Це абревіатура - звичайний кольоровий тег
                    textbox.insert("end", tag_text, tag_name)

                last_end = match.end()

            # Залишок рядка після останнього тегу
            if last_end < len(remaining_line):
                textbox.insert("end", remaining_line[last_end:])
        else:
            # Немає тегів - вставляємо як звичайний текст
            textbox.insert("end", line)
```

```
        textbox.insert("end", "\n")

    # Налаштовуємо базовий шрифт для всього тексту (DEF: Definition text)
    textbox.configure(font=("Segoe UI", 16))


def format_and_display(raw_text: str, headword: str | None = None) -> str:
    """
    Парсинг та форматування тексту визначення зі словника.

    Args:
        raw_text (str): Сирий текст визначення з сервера.
        headword (str|None): Якщо вказано, використовується для підстановки '~'.

    Returns:
        str: Відформатований текст з заголовками частин мови, тегами та відступами.
    """
    if not raw_text or raw_text == "NOT_FOUND":
        return raw_text

    text = raw_text

    # Декодування HTML entities (&#x27; -> ', &quot; -> ")
    text = html.unescape(text)

    # --- NEW: розкриваємо посилання <<word>> -> word, прибираємо стрічки початкові
'>' та обробляємо (a|b) групи ---
    # Заміна посилань виду <<word>> на просто word
    text = re.sub(r'<<\s*([^<>]+?)\s*>>', r'\1', text)

    # Прибрати початкові '>' в цитатах
    text = re.sub(r'(?m)^\s*>+\s*', '', text)

    def _pipe_group_repl(m):
        inner = m.group(1)
        parts = [p.strip() for p in inner.split('|') if p.strip()]
        # prefer alphabetic token if present (likely a headword), otherwise join
        alpha = [p for p in parts if re.match(r'^[A-Za-z\- ]+$', p)]
        if alpha:
            return ' (' + ' / '.join(alpha) + ')'
        return ' (' + ' / '.join(parts) + ')'

    text = re.sub(r'\(([^()]*\|[^()]*)\)', _pipe_group_repl, text)

    # Якщо є headword - підставляємо тильду (~) на його нижній регістр
    if headword:
        hw = headword.strip().lower()
        # Замінимо всі варіанти: '~', ' ~', '~ ', ' ~ ' - просто заміна символьна
        text = text.replace('~', hw)

    # Очищення від HTML тегів
    text = re.sub(r'<[^>]+>', '', text)
    # Видаляємо DSL/технічні блоки типу [m1], [c], але НЕ заголовки POS типу [ NOUN
]
    text = re.sub(r'\[(?![
]*(NOUN|VERB|ADJECTIVE|ADVERB|PREPOSITION|CONJUNCTION|PRONOUN|INTERJECTION|NUMERAL|
PHRASAL VERB)[ ]*\])[^]]*\]', '', text)
    text = text.replace('\\n', '\n').replace('\r\n', '\n').replace('\r', '\n')
```

```
    # --- Нові правила очищення / тегування ---
    # Повний список абревіатур, які зустрічаються в словниках
    # ВАЖЛИВО: \b не працює з кирилицею, тому використовуємо (?:^|[^а-яіїєґА-
ЯІЇЄҐ])
    # Формат: (regex_pattern, display_text)

    # Функція для створення regex-патерну для кирилічних абревіатур
    def _cyrillic_word_pattern(abbr: str) -> str:
        """Створює regex для кирилічної абревіатури з правильними межами слів."""
        # Екрануємо крапку якщо є
        escaped = re.escape(abbr)
        # Межа: початок рядка або не-кирилічний символ
        return r'(?:^|(?<=[^а-яіїєґА-ЯІЇЄҐA-Za-z]))' + escaped + r'(?=[^а-яіїєґА-
ЯІЇЄҐA-Za-z]|$)'

    ABBREVIATIONS = [
        # Стилістичні позначки
        (_cyrillic_word_pattern('розм.'), 'розм.'),       # розмовне
        (_cyrillic_word_pattern('книжк.'), 'книжк.'),     # книжне
        (_cyrillic_word_pattern('поет.'), 'поет.'),       # поетичне
        (_cyrillic_word_pattern('жарт.'), 'жарт.'),       # жартівливе
        (_cyrillic_word_pattern('ірон.'), 'ірон.'),       # іронічне
        (_cyrillic_word_pattern('зневажл.'), 'зневажл.'), # зневажливе
        (_cyrillic_word_pattern('вульг.'), 'вульг.'),     # вульгарне
        (_cyrillic_word_pattern('прост.'), 'прост.'),     # просторічне
        (_cyrillic_word_pattern('діал.'), 'діал.'),       # діалектне
        (_cyrillic_word_pattern('застар.'), 'застар.'),   # застаріле
        (_cyrillic_word_pattern('рідко'), 'рідко'),        # рідковживане

        # Галузеві позначки
        (_cyrillic_word_pattern('мор.'), 'мор.'),         # морський термін
        (_cyrillic_word_pattern('військ.'), 'військ.'),   # військовий
        (_cyrillic_word_pattern('зоол.'), 'зоол.'),       # зоологія
        (_cyrillic_word_pattern('бот.'), 'бот.'),         # ботаніка
        (_cyrillic_word_pattern('мед.'), 'мед.'),         # медицина
        (_cyrillic_word_pattern('юр.'), 'юр.'),           # юридичний
        (_cyrillic_word_pattern('тех.'), 'тех.'),         # технічний
        (_cyrillic_word_pattern('фіз.'), 'фіз.'),         # фізика
        (_cyrillic_word_pattern('хім.'), 'хім.'),         # хімія
        (_cyrillic_word_pattern('матем.'), 'матем.'),     # математика
        (_cyrillic_word_pattern('муз.'), 'муз.'),         # музика
        (_cyrillic_word_pattern('спорт.'), 'спорт.'),     # спорт
        (_cyrillic_word_pattern('авіа.'), 'авіа.'),       # авіація
        (_cyrillic_word_pattern('ел.'), 'ел.'),           # електрика
        (_cyrillic_word_pattern('рел.'), 'рел.'),         # релігія
        (_cyrillic_word_pattern('біол.'), 'біол.'),       # біологія
        (_cyrillic_word_pattern('геол.'), 'геол.'),       # геологія
        (_cyrillic_word_pattern('екон.'), 'екон.'),       # економіка
        (_cyrillic_word_pattern('політ.'), 'політ.'),     # політика

        # Географічні позначки
        (_cyrillic_word_pattern('амер.'), 'амер.'),       # американізм
        (_cyrillic_word_pattern('брит.'), 'брит.'),       # британізм
        (_cyrillic_word_pattern('шотл.'), 'шотл.'),       # шотландізм
        (_cyrillic_word_pattern('австрал.'), 'австрал.'), # австралійський

        # Граматичні позначки (латинські - використовують \b)
        (r'\bpl\b', 'мн.'),             # множина (plural)
        (r'\bsg\b', 'одн.'),            # однина (singular)
```

```
        (_cyrillic_word_pattern('перен.'), 'перен.'),     # переносне значення
        (_cyrillic_word_pattern('букв.'), 'букв.'),       # буквально
        (_cyrillic_word_pattern('збірн.'), 'збірн.'),     # збірне
        (_cyrillic_word_pattern('скор.'), 'скор.'),       # скорочення
        (r'\battr\b', 'означ.'),        # attributive / означальне
        (r'\bpred\b', 'присуд.'),       # predicative / присудкове
    ]

    # Заміна 'тж' на 'також' перед тегуванням (з правильними межами для кирилиці)
    text = re.sub(r'(?:^|(?<=[^а-яіїєґА-ЯІЇЄҐ]))тж(?=[^а-яіїєґА-ЯІЇЄҐ]|$)',
'також', text, flags=re.IGNORECASE)

    # Заміна 'напр.' на 'наприклад'
    text = re.sub(r'(?:^|(?<=[^а-яіїєґА-ЯІЇЄҐ]))напр\.(?=[^а-яіїєґА-ЯІЇЄҐ]|$)',
'наприклад', text, flags=re.IGNORECASE)

    # Заміна 'і т.п.' / 'і т.д.' / 'etc.' на повні форми
    text = re.sub(r'і т\.п\.', 'і тому подібне', text, flags=re.IGNORECASE)
    text = re.sub(r'і т\.д\.', 'і так далі', text, flags=re.IGNORECASE)
    text = re.sub(r'\betc\.\b', 'тощо', text, flags=re.IGNORECASE)

    # Обернемо знайдені абревіатури в [аббр.] для візуального виділення
    def _tag_abbr(abbr_text: str) -> str:
        """Обгортає абревіатуру в квадратні дужки для візуального виділення."""
        return f'[{abbr_text}]'

    # Застосуємо для всіх абревіатур зі списку
    for pattern, display_text in ABBREVIATIONS:
        text = re.sub(pattern, _tag_abbr(display_text), text, flags=re.IGNORECASE)

    # Заміна phrasal verbs
    text = re.sub(
        r'(\d+\.\s*)?(phrasal\s+v|ph\.?\s*v)\b\s*',
        lambda m: f'\n\n{POS_HEADERS.get("phrasal v", "PHRASAL VERB")}\n   ',
        text,
        flags=re.IGNORECASE
    )

    # Заміна частин мови
    pos_pattern =
r'(\d+\.\s*)?\b(noun|verb|adj(?:ective)?|adv(?:erb)?|prep(?:osition)?|conj(?:unctio
n)?|pron(?:oun)?|int(?:erjection)?|num(?:eral)?|n|v)\b(?:\s+|(?=\s))'

    def replace_pos(match):
        pos = match.group(2).lower()
        if pos in ['n', 'noun']:
            key = 'n'
        elif pos in ['v', 'verb']:
            key = 'v'
        elif pos.startswith('adj'):
            key = 'adj'
        elif pos.startswith('adv'):
            key = 'adv'
        elif pos.startswith('prep'):
            key = 'prep'
        elif pos.startswith('conj'):
            key = 'conj'
        elif pos.startswith('pron'):
            key = 'pron'
```

```
            elif pos.startswith('int'):
                key = 'int'
            elif pos.startswith('num'):
                key = 'num'
            else:
                key = pos
            header = POS_HEADERS.get(key, pos.upper())
            return f'\n\n[ {header} ]\n   '

        text = re.sub(pos_pattern, replace_pos, text, flags=re.IGNORECASE)
        text = re.sub(r'^\s*\d+\.\s*', '', text, flags=re.MULTILINE)

        # Форматування відступів
        lines = text.split('\n')
        formatted_lines = []
        for line in lines:
            stripped = line.strip()
            if not stripped:
                if formatted_lines and formatted_lines[-1] != '':
                    formatted_lines.append('')
                continue
            # Перевіряємо чи це заголовок частини мови (формат: [ NOUN ])
            is_header = stripped.startswith('[') and stripped.endswith(']')
            if is_header:
                formatted_lines.append(stripped)
            else:
                if not stripped.startswith('   '):
                    formatted_lines.append(f'   {stripped}')
                else:
                    formatted_lines.append(stripped)

        # Повертаємо готовий текст
        return '\n'.join(formatted_lines)


class StatusIndicator(ctk.CTkFrame):
    """Індикатор статусу з'єднання з кольоровою точкою."""

    def __init__(self, master, **kwargs):
        super().__init__(master, fg_color="transparent", **kwargs)

        self.dot = ctk.CTkLabel(
            self,
            text="●",
            font=("Segoe UI", 16),
            text_color=COLORS["danger"]
        )
        self.dot.pack(side="left", padx=(0, 5))

        self.label = ctk.CTkLabel(
            self,
            text="Offline",
            font=("Segoe UI", 12),
            text_color=COLORS["text_secondary"]
        )
        self.label.pack(side="left")

    def set_online(self):
        self.dot.configure(text_color=COLORS["success"])
```

```
        self.label.configure(text="Online", text_color=COLORS["success"])

    def set_offline(self):
        self.dot.configure(text_color=COLORS["danger"])
        self.label.configure(text="Offline", text_color=COLORS["danger"])

    def set_connecting(self):
        self.dot.configure(text_color=COLORS["warning"])
        self.label.configure(text="Connecting...", text_color=COLORS["warning"])


# --- Відомі скорочення для badge ---
KNOWN_ABBREVIATIONS = {
    'юр.': 'Legal',
    'біол.': 'Biology',
    'мед.': 'Medicine',
    'тех.': 'Tech',
    'рел.': 'Religion',
    'іст.': 'History',
    'муз.': 'Music',
    'арх.': 'Architecture',
    'хім.': 'Chemistry',
    'фіз.': 'Physics',
    'мат.': 'Math',
    'бот.': 'Botany',
    'зоол.': 'Zoology',
    'авіа.': 'Aviation',
    'воєн.': 'Military',
    'мор.': 'Maritime',
    'комп.': 'Computing',
    'розм.': 'Colloquial',
    'книжн.': 'Literary',
    'заст.': 'Obsolete',
    'діал.': 'Dialect',
    'pl': 'Plural',
    'sg': 'Singular',
    'attr': 'Attributive',
    'predic': 'Predicative',
    'амер.': 'American',
    'брит.': 'British',
}



class ResultCard(ctk.CTkFrame):
    """
    Картка результату перекладу (English → Ukrainian).
    Спрощена версія без reverse search.
    """

    def __init__(self, master, headword: str, definition: str,
favorite_callback=None, is_favorite=False, **kwargs):
        """
        Args:
            master: Батьківський віджет
            headword: Англійське слово (заголовок)
            definition: Українське визначення
            favorite_callback: Callback function(word, definition, is_favorite) для
toggle favorites
```

```
            is_favorite: Початковий стан улюбленого
        """
        # Видаляємо search_query якщо передано (для сумісності)
        kwargs.pop('search_query', None)

        self.favorite_callback = favorite_callback
        self.is_favorite = is_favorite

        super().__init__(
            master,
            fg_color=COLORS["bg_card"],
            corner_radius=12,
            border_width=1,
            border_color=COLORS["border"],
            **kwargs
        )

        self.headword = html.unescape(headword).strip()
        self.definition = html.unescape(definition)

        # Debug logging
        logger.debug(f"ResultCard: headword='{self.headword}'")

        # Головний контейнер
        main_container = ctk.CTkFrame(self, fg_color="transparent")
        main_container.pack(fill="both", expand=True, padx=24, pady=12)

        # === HEADER: Слово (великий, bold, білий) + Copy ===
        header_frame = ctk.CTkFrame(main_container, fg_color="transparent")
        header_frame.pack(fill="x", pady=(0, 5))  # Minimal bottom padding

        # Заголовок - англійське слово
        display_word = self.headword.title() if self.headword else "Result"

        self.word_label = ctk.CTkLabel(
            header_frame,
            text=display_word,
            font=("Segoe UI", 32, "bold"),
            text_color=COLORS["title_color"],  # Темно-синій для Light Mode
            anchor="w"
        )
        self.word_label.pack(side="left", anchor="w", pady=(20, 5))  # Top padding
okay, bottom minimal

        # Spacer
        spacer = ctk.CTkFrame(header_frame, fg_color="transparent", width=20)
        spacer.pack(side="left", fill="x", expand=True)

        # Star button for favorites (if callback provided)
        if self.favorite_callback:
            star_text = "☆" if self.is_favorite else "☆"
            self.star_btn = ctk.CTkButton(
                header_frame,
                text=star_text,
                width=40,
                height=36,
                font=("Segoe UI", 16),
                fg_color="transparent",
                hover_color=COLORS["border"],
```

```
                text_color="#FFD700" if self.is_favorite else COLORS["text_muted"],
                corner_radius=8,
                command=self._toggle_favorite
            )
            self.star_btn.pack(side="right", padx=(0, 10))

        # Copy button
        self.copy_btn = ctk.CTkButton(
            header_frame,
            text="📋 Copy",
            width=100,
            height=36,
            font=("Segoe UI", 12),
            fg_color=COLORS["accent"],
            hover_color=COLORS["accent_hover"],
            text_color="#FFFFFF",
            corner_radius=8,
            command=self._copy_to_clipboard
        )
        self.copy_btn.pack(side="right")


        # Розділювач
        separator = ctk.CTkFrame(main_container, fg_color=COLORS["border"],
height=1)
        separator.pack(fill="x", pady=(0, 0))  # Minimal padding

        # === CONTENT: Regular Frame для визначень ===
        self.content_frame = ctk.CTkFrame(
            main_container,
            fg_color="transparent",
            corner_radius=0
        )
        self.content_frame.pack(fill="both", expand=True, pady=(0, 20))  # Start
immediately below title

        # Парсимо та відображаємо
        self._parse_and_render(self.definition)

    def _parse_and_render(self, text):
        """Smart Parser: розбирає текст та створює структуровані віджети."""
        lines = text.split('\n')

        for line in lines:
            stripped = line.strip()
            if not stripped:
                spacer = ctk.CTkFrame(self.content_frame, fg_color="transparent",
height=8)
                spacer.pack(fill="x")
                continue

            line_type = self._classify_line(stripped)

            if line_type == "pos_header":
                self._render_pos_header(stripped)
            elif line_type == "definition":
                self._render_definition(stripped)
            elif line_type == "example":
                self._render_example(stripped)
```

```
            else:
                self._render_regular(stripped)

    def _classify_line(self, line):
        """Класифікує рядок за типом."""
        if line.startswith('[') and line.endswith(']'):
            return "pos_header"
        if re.match(r'^\d+[.)]\s', line):
            return "definition"
        if line.startswith('~') or line.startswith('-') or ' — ' in line or
line.startswith('   '):
            return "example"
        return "regular"

    def _render_pos_header(self, text):
        """Рендеринг заголовка частини мови."""
        pos_text = text.strip('[]').strip()

        header_frame = ctk.CTkFrame(self.content_frame, fg_color="transparent")
        header_frame.pack(fill="x", pady=(12, 6))

        accent_line = ctk.CTkFrame(header_frame, fg_color="#10B981", width=4,
height=20)
        accent_line.pack(side="left", padx=(0, 10))

        ctk.CTkLabel(
            header_frame,
            text=pos_text,
            font=("Segoe UI", 14, "bold"),
            text_color="#10B981",
            anchor="w"
        ).pack(side="left")

    def _render_definition(self, text):
        """Рендеринг визначення з номером."""
        def_frame = ctk.CTkFrame(self.content_frame, fg_color="transparent")
        def_frame.pack(fill="x", pady=4)

        match = re.match(r'^(\d+[.)])\s*(.*)$', text)
        if match:
            number = match.group(1)
            rest = match.group(2)
        else:
            number = ""
            rest = text

        if number:
            ctk.CTkLabel(
                def_frame,
                text=number,
                font=("Segoe UI", 14, "bold"),
                text_color="#3B82F6",
                width=35,
                anchor="w"
            ).pack(side="left")

        ctk.CTkLabel(
            def_frame,
            text=rest,
```

```
            font=("Segoe UI", 16),
            text_color=COLORS["text_primary"],
            anchor="w",
            wraplength=550,
            justify="left"
        ).pack(side="left", fill="x", expand=True)

    def _render_example(self, text):
        """Рендеринг прикладу (indented, italic)."""
        example_frame = ctk.CTkFrame(self.content_frame, fg_color="transparent")
        example_frame.pack(fill="x", pady=2, padx=(35, 0))

        if ' – ' in text:
            parts = text.split(' – ', 1)
            english_part = parts[0].strip().lstrip('~- ')
            ukr_part = parts[1].strip() if len(parts) > 1 else ""

            ctk.CTkLabel(
                example_frame,
                text=english_part,
                font=("Segoe UI", 12, "italic"),
                text_color="#60A5FA",
                anchor="w"
            ).pack(side="left")

            ctk.CTkLabel(
                example_frame,
                text=" – ",
                font=("Segoe UI", 12),
                text_color=COLORS["text_muted"],
                anchor="w"
            ).pack(side="left")

            ctk.CTkLabel(
                example_frame,
                text=ukr_part,
                font=("Segoe UI", 15),  # EX: Example text (Ukrainian part)
                text_color=COLORS["text_secondary"],
                anchor="w",
                wraplength=450,
                justify="left"
            ).pack(side="left", fill="x", expand=True)
        else:
            display_text = text.lstrip('~- ').strip()
            ctk.CTkLabel(
                example_frame,
                text=f"→ {display_text}",
                font=("Segoe UI", 15, "italic"),  # EX: Example text
                text_color=COLORS["text_secondary"],
                anchor="w",
                wraplength=500,
                justify="left"
            ).pack(side="left", fill="x", expand=True)

    def _render_regular(self, text):
        """Рендеринг звичайного тексту."""
        ctk.CTkLabel(
            self.content_frame,
            text=text,
```

```
            font=("Segoe UI", 12),
            text_color=COLORS["text_secondary"],
            anchor="w",
            wraplength=550,
            justify="left"
        ).pack(fill="x", pady=2)

    def _copy_to_clipboard(self):
        """Копіювання ТІЛЬКИ тексту перекладу (без headword та заголовків POS)."""
        try:
            # Очищаємо текст від заголовків та форматування
            clean_text = self._get_clean_translation()

            self.clipboard_clear()
            self.clipboard_append(clean_text)
            self.copy_btn.configure(text="✓ Copied")
            self.after(1500, lambda: self.copy_btn.configure(text="📋 Copy"))
            logger.info(f"Скопійовано переклад для: '{self.headword}'")
        except Exception as e:
            logger.error(f"Помилка копіювання: {e}")

    def _get_clean_translation(self) -> str:
        """
        Отримати чистий текст перекладу без заголовків POS та headword.

        Returns:
            str: Чистий текст перекладу для копіювання.
        """
        text = self.definition

        # Видаляємо заголовки частин мови типу [ NOUN ], [ VERB ] тощо
        text =
re.sub(r'\[\s*(NOUN|VERB|ADJECTIVE|ADVERB|PREPOSITION|CONJUNCTION|PRONOUN|INTERJECT
ION|NUMERAL|PHRASAL VERB)\s*\]', '', text)

        # Видаляємо headword якщо він на початку рядка
        if self.headword:
            # Видаляємо headword на початку (можливо з пробілами та переносами)
            pattern = rf'^\s*{re.escape(self.headword)}\s*\n?'
            text = re.sub(pattern, '', text, flags=re.IGNORECASE)

        # Очищаємо зайві переноси рядків та пробіли
        text = re.sub(r'\n{3,}', '\n\n', text)  # Макс 2 переноси
        text = re.sub(r'^\s+', '', text, flags=re.MULTILINE)  # Видаляємо пробіли
на початку рядків
        text = text.strip()

        return text

    def _toggle_favorite(self):
        """Toggle favorite status for this word."""
        if not self.favorite_callback:
            return

        # Toggle the state
        self.is_favorite = not self.is_favorite

        # Update visual state immediately
        star_text = "☆" if self.is_favorite else "☆"
```

```
        self.star_btn.configure(
            text=star_text,
            text_color="#FFD700" if self.is_favorite else COLORS["text_muted"]
        )

        # Call the callback with word, definition, and new favorite status
        # This will update the database
        self.favorite_callback(self.headword, self.definition, self.is_favorite)


class HistoryItem(ctk.CTkButton):
    """Клікабельний елемент історії пошуку."""

    def __init__(self, master, word, callback, **kwargs):
        super().__init__(
            master,
            text=f"🕒 {word}",
            font=("Segoe UI", 12),
            fg_color="transparent",
            text_color=COLORS["text_secondary"],
            hover_color=COLORS["border"],
            anchor="w",
            command=lambda: callback(word),
            **kwargs
        )


class ModernDictionaryApp(ctk.CTk):
    """
    Головне вікно застосунку з принципом "70% Golden Mean".

    Args:
        client (DictionaryClient): Клієнт для мережевих операцій.
    """

    def __init__(self, client: DictionaryClient = None):
        super().__init__()

        # === Примусово Light Mode ===
        ctk.set_appearance_mode("Light")
        ctk.set_default_color_theme("blue")

        # Window Configuration
        self.title("🔬 E-Dictionary Pro")
        self.geometry("1100x700")
        self.minsize(900, 600)
        self.configure(fg_color=COLORS["bg_main"])

        # Для drag вікна
        self._drag_data = {"x": 0, "y": 0}

        # Network client (можна передати ззовні або створити новий)
        self.network = client if client else DictionaryClient()
        # Keep self.client for backward compatibility
        self.client = self.network
        # Database Manager для історії та налаштувань (ООР Композиція)
        # Змушуємо використовувати файлову базу для збереження історії
        self.db = DatabaseManager("dictionary_history.db")
        self._auto_connect_attempted = False
```

```
        # Store current word for favorites toggle
        self.current_headword = None
        self.current_definition = None

        # Thread tracking for network operations
        self._network_threads = []  # Список активних мережевих потоків

        # Create UI
        self._create_custom_title_bar()
        self._create_layout()
        self._bind_shortcuts()

        # Автоматична спроба підключення при старті
        # (не ставимо фокус одразу - поле disabled до підключення)
        self.after(500, self._try_auto_connect)

        # Налаштування обробника закриття вікна для безпечного завершення
        self.protocol("WM_DELETE_WINDOW", self._close_window)

        logger.info("Застосунок успішно запущено")

    def _try_auto_connect(self):
        """Автоматична спроба підключення при запуску (в фоновому потоці)."""
        if self._auto_connect_attempted:
            return

        self._auto_connect_attempted = True

        # Показуємо що намагаємось підключитись
        self.status_indicator.set_connecting()
        self.connect_btn.configure(text="...", state="disabled")
        self.update()

        host = self.host_entry.get().strip() or "127.0.0.1"
        try:
            port = int(self.port_entry.get().strip() or "8080")
        except ValueError:
            port = 8080

        self.network.host = host
        self.network.port = port

        # Запускаємо підключення в фоновому потоці
        def connect_thread():
            try:
                connected = self.network.connect()
                # Оновлюємо UI в головному потоці
                self.after(0, lambda: self._on_auto_connect_result(connected, host,
port))
            except Exception as e:
                logger.error(f"[UI] Помилка автопідключення: {e}")
                self.after(0, lambda: self._on_auto_connect_result(False, host,
port))

        thread = threading.Thread(target=connect_thread, daemon=True)
        thread.start()
        self._network_threads.append(thread)
```

```
    def _on_auto_connect_result(self, connected: bool, host: str, port: int):
        """Обробка результату автопідключення (викликається в UI потоці)."""
        if connected:
            # Успішно підключено
            self.status_indicator.set_online()
            self.connect_btn.configure(text="Disconnect", state="normal",
fg_color="#EF4444", hover_color="#DC2626")
            self._update_ui_state(True)
            self._add_to_log_panel(f"✅ Автопідключення до {host}:{port}")
            # Оновлюємо start screen
            self._show_start_screen()
            self._refresh_word_of_the_day()
        else:
            # Не вдалося підключитися
            self.status_indicator.set_offline()
            self.connect_btn.configure(text="Connect", state="normal",
fg_color="#007AFF")
            self._update_ui_state(False)
            self._add_to_log_panel(f"❌ Сервер недоступний: {host}:{port}")
            # Показуємо ропир з попередженням
            self.after(100, self._show_connection_warning)

    def _create_custom_title_bar(self):
        """Створення кастомного title bar з навігацією."""
        self.title_bar = ctk.CTkFrame(
            self,
            fg_color="#1E1E1E",
            height=40,
            corner_radius=0
        )
        self.title_bar.pack(fill="x", side="top")
        self.title_bar.pack_propagate(False)

        # Bind для перетягування вікна
        self.title_bar.bind("<Button-1>", self._start_drag)
        self.title_bar.bind("<B1-Motion>", self._on_drag)

        # === ЛІВА ЧАСТИНА: Логотип ===
        title_left = ctk.CTkFrame(self.title_bar, fg_color="transparent")
        title_left.pack(side="left", padx=15)

        ctk.CTkLabel(
            title_left,
            text="📐",
            font=("Segoe UI", 16),
            text_color="#FFFFFF"
        ).pack(side="left", padx=(0, 8))

        title_label = ctk.CTkLabel(
            title_left,
            text="E-Dictionary Pro",
            font=("Segoe UI", 12, "bold"),
            text_color="#FFFFFF"
        )
        title_label.pack(side="left")
        title_label.bind("<Button-1>", self._start_drag)
        title_label.bind("<B1-Motion>", self._on_drag)

        # === ПРАВА ЧАСТИНА: Кнопки керування ===
```

```
btn_frame = ctk.CTkFrame(self.title_bar, fg_color="transparent")
btn_frame.pack(side="right", padx=5)

# Minimize
ctk.CTkButton(
    btn_frame, text="–", width=40, height=30,
    font=("Segoe UI", 14), fg_color="transparent",
    hover_color="#3a3a3a", text_color="#FFFFFF",
    corner_radius=0, command=self._minimize_window
).pack(side="left", padx=2)

# Maximize
self.maximize_btn = ctk.CTkButton(
    btn_frame, text="□", width=40, height=30,
    font=("Segoe UI", 12), fg_color="transparent",
    hover_color="#3a3a3a", text_color="#FFFFFF",
    corner_radius=0, command=self._toggle_maximize
)
self.maximize_btn.pack(side="left", padx=2)

# Close
ctk.CTkButton(
    btn_frame, text="✕", width=40, height=30,
    font=("Segoe UI", 12), fg_color="transparent",
    hover_color="#e81123", text_color="#FFFFFF",
    corner_radius=0, command=self._close_window
).pack(side="left", padx=2)

# === ЦЕНТРАЛЬНА ЧАСТИНА: Навігаційні кнопки ===
nav_frame = ctk.CTkFrame(self.title_bar, fg_color="transparent")
nav_frame.pack(side="right", padx=20)


# Add Word Button (+) - Square icon-only
self.add_word_btn = ctk.CTkButton(
    nav_frame,
    text="+",
    width=40,
    height=40,
    font=("Segoe UI", 20, "bold"),
    fg_color="transparent",
    hover_color="#3a3a3a",
    text_color="#FFFFFF",
    corner_radius=6,
    command=self._show_add_word_dialog
)
self.add_word_btn.pack(side="right", padx=5)

# History & Favorites Button (🕐) - Square icon-only
self.history_btn = ctk.CTkButton(
    nav_frame,
    text="🕐",
    width=40,
    height=40,
    font=("Segoe UI", 16),
    fg_color="transparent",
    hover_color="#3a3a3a",
    text_color="#FFFFFF",
    corner_radius=6,
```

```
            command=self._show_history_favorites_popup
        )
        self.history_btn.pack(side="right", padx=5)

        # Help Button - Square icon-only
        ctk.CTkButton(
            nav_frame,
            text="?",
            width=40,
            height=40,
            font=("Segoe UI", 16),
            fg_color="transparent",
            hover_color="#3a3a3a",
            text_color="#FFFFFF",
            corner_radius=6,
            command=self._show_about
        ).pack(side="right", padx=5)

    def _start_drag(self, event):
        """Початок перетягування вікна."""
        self._drag_data["x"] = event.x
        self._drag_data["y"] = event.y

    def _on_drag(self, event):
        """Перетягування вікна."""
        x = self.winfo_x() - self._drag_data["x"] + event.x
        y = self.winfo_y() - self._drag_data["y"] + event.y
        self.geometry(f"+{x}+{y}")

    def _minimize_window(self):
        """Згортання вікна."""
        self.iconify()

    def _toggle_maximize(self):
        """Максимізація/відновлення вікна."""
        if self.state() == 'zoomed':
            self.state('normal')
            self.maximize_btn.configure(text="□")
        else:
            self.state('zoomed')
            self.maximize_btn.configure(text="❐")

    def _close_window(self):
        """Безпечне закриття вікна з очищенням ресурсів."""
        try:
            # Відключаємося від сервера
            if self.network.connected:
                try:
                    self.network.disconnect()
                except Exception as e:
                    logger.warning(f"[UI] Помилка відключення: {e}")

            # Очищаємо базу даних
            if hasattr(self, 'db') and self.db:
                try:
                    self.db.close()
                except Exception as e:
                    logger.warning(f"[UI] Помилка закриття БД: {e}")
```

```
            # Очікуємо завершення активних мережевих потоків (з таймаутом)
            for thread in self._network_threads[:]:  # Копіюємо список
                if thread.is_alive():
                    thread.join(timeout=0.5)  # Максимум 0.5 секунди на потік
                    if thread.is_alive():
                        logger.warning(f"[UI] Мережевий потік не завершився
вчасно")

            logger.info("[UI] Застосунок закривається")
        except Exception as e:
            logger.error(f"[UI] Помилка при закритті: {e}")
        finally:
            # Завжди закриваємо вікно
            try:
                self.destroy()
            except Exception:
                pass

    def _create_layout(self):
        """Створення Single Column layout."""
        self._create_top_bar()

        # Головний контейнер - одна колонка по центру
        self.main_container = ctk.CTkFrame(self, fg_color="transparent")
        self.main_container.pack(fill="both", expand=True, padx=40, pady=(0, 20))

        # Контейнер для контенту (центрований)
        self.content_frame = ctk.CTkFrame(self.main_container,
fg_color="transparent")
        self.content_frame.pack(fill="both", expand=True)

        # Показуємо start screen з Word of the Day
        self._show_start_screen()

    def _create_top_bar(self):
        """Створення компактної верхньої панелі з пошуком."""
        top_bar = ctk.CTkFrame(self, fg_color=COLORS["bg_card"], corner_radius=0,
height=80)
        top_bar.pack(fill="x", padx=0, pady=0)
        top_bar.pack_propagate(False)

        inner = ctk.CTkFrame(top_bar, fg_color="transparent")
        inner.pack(fill="both", expand=True, padx=40, pady=15)

        # === ЛІВА ЧАСТИНА: Пошук ===
        search_frame = ctk.CTkFrame(inner, fg_color="transparent")
        search_frame.pack(side="left", fill="x", expand=True)

        # Search Entry - ЧИСТА ініціалізація без StringVar
        self.search_var = ctk.StringVar(value="")
        self.search_entry = ctk.CTkEntry(
            search_frame,
            height=50,
            width=400,
            font=("Segoe UI", 16),
            placeholder_text="",
            placeholder_text_color=COLORS["text_muted"],
            text_color=COLORS["text_primary"],
            fg_color=COLORS["bg_card"],
```

```
        border_color=COLORS["border"],
        border_width=2,
        corner_radius=12,
        textvariable=self.search_var
        # НЕ ставимо state="disabled" тут - зробимо це окремо
    )
    self.search_entry.pack(side="left", fill="x", expand=True, padx=(0, 15))
    self.search_entry.bind('<Return>', lambda e: self._translate())

    # Блокуємо ПІСЛЯ створення щоб placeholder працював коректно
    self.search_entry.configure(state="disabled")

    # Translate Button - з відступом
    self.search_btn = ctk.CTkButton(
        search_frame,
        text="🔍 Translate",
        width=150,
        height=50,
        font=("Segoe UI", 14, "bold"),
        fg_color=COLORS["text_muted"],  # Сірий коли offline
        hover_color=COLORS["text_muted"],
        corner_radius=12,
        command=self._translate,
        state="disabled"  # Заблоковано до підключення
    )
    self.search_btn.pack(side="left", padx=(0, 20))

    # === ПРАВА ЧАСТИНА: Connection ===
    conn_frame = ctk.CTkFrame(inner, fg_color="transparent")
    conn_frame.pack(side="right")

    self.status_indicator = StatusIndicator(conn_frame)
    self.status_indicator.pack(side="right", padx=(15, 0))

    self.connect_btn = ctk.CTkButton(
        conn_frame,
        text="Connect",
        width=90,
        height=40,
        font=("Segoe UI", 11, "bold"),
        fg_color="#007AFF",
        hover_color="#0056B3",
        corner_radius=8,
        command=self._toggle_connection
    )
    self.connect_btn.pack(side="right", padx=(10, 0))

    self.port_entry = ctk.CTkEntry(
        conn_frame,
        width=55,
        height=40,
        font=("Segoe UI", 11),
        text_color=COLORS["text_primary"],
        fg_color=COLORS["bg_card"],
        border_color=COLORS["border"]
    )
    self.port_entry.insert(0, "8080")
    self.port_entry.pack(side="right", padx=(5, 0))
```

```
        ctk.CTkLabel(conn_frame, text=":", font=("Segoe UI", 11),
text_color=COLORS["text_secondary"]).pack(side="right")

        self.host_entry = ctk.CTkEntry(
            conn_frame,
            width=100,
            height=40,
            font=("Segoe UI", 11),
            text_color=COLORS["text_primary"],
            fg_color=COLORS["bg_card"],
            border_color=COLORS["border"]
        )
        self.host_entry.insert(0, "127.0.0.1")
        self.host_entry.pack(side="right")

    def _show_start_screen(self):
        """Показати стартовий екран з Word of the Day."""
        # Очищаємо контент
        for widget in self.content_frame.winfo_children():
            widget.destroy()

        # Центрований контейнер
        center_frame = ctk.CTkFrame(self.content_frame, fg_color="transparent")
        center_frame.place(relx=0.5, rely=0.4, anchor="center")

        # Привітання
        ctk.CTkLabel(
            center_frame,
            text="👌 Вітаємо в E-Dictionary Pro",
            font=("Segoe UI", 24, "bold"),
            text_color=COLORS["text_primary"]
        ).pack(pady=(80, 10))

        # Підказка залежно від статусу підключення
        if self.network.connected:
            hint_text = "Введіть слово вище для перекладу"
            hint_color = COLORS["text_secondary"]
        else:
            hint_text = "⚠ Натисніть 'Connect' щоб підключитися до сервера"
            hint_color = "#F59E0B"  # Warning yellow

        ctk.CTkLabel(
            center_frame,
            text=hint_text,
            font=("Segoe UI", 14),
            text_color=hint_color
        ).pack(pady=(0, 40))

        # === WORD OF THE DAY CARD ===
        wotd_card = ctk.CTkFrame(
            center_frame,
            fg_color=COLORS["bg_card"],
            corner_radius=16,
            border_width=1,
            border_color=COLORS["border"]
        )
        wotd_card.pack(pady=20, ipadx=40, ipady=20)

        # Header
```

```
wotd_header = ctk.CTkFrame(wotd_card, fg_color="transparent")
wotd_header.pack(fill="x", padx=30, pady=(20, 10))

ctk.CTkLabel(
    wotd_header,
    text="✨ Слово дня",
    font=("Segoe UI", 14, "bold"),
    text_color=COLORS["text_secondary"]
).pack(side="left")

# Copy button
self.wotd_copy_btn = ctk.CTkButton(
    wotd_header,
    text="📋",
    width=30,
    height=30,
    font=("Segoe UI", 14),
    fg_color="transparent",
    hover_color=COLORS["border"],
    text_color=COLORS["text_secondary"],
    corner_radius=6,
    command=self._copy_wotd
)
self.wotd_copy_btn.pack(side="right", padx=(5, 0))

# Refresh button
ctk.CTkButton(
    wotd_header,
    text="🔄",
    width=30,
    height=30,
    font=("Segoe UI", 14),
    fg_color="transparent",
    hover_color=COLORS["border"],
    text_color=COLORS["text_secondary"],
    corner_radius=6,
    command=self._refresh_word_of_the_day
).pack(side="right")

# Word
self.wotd_word_label = ctk.CTkLabel(
    wotd_card,
    text="Hello",
    font=("Segoe UI", 28, "bold"),
    text_color="#10B981"  # Emerald - добре виглядає на білому
)
self.wotd_word_label.pack(padx=30, pady=(10, 5))

# Definition - CTkTextbox для скролінгу довгих визначень
# Wrap in frame for internal padding
textbox_frame = ctk.CTkFrame(wotd_card, fg_color="transparent")
textbox_frame.pack(padx=30, pady=(0, 20), fill="x")

self.wotd_definition_textbox = ctk.CTkTextbox(
    textbox_frame,
    font=("Segoe UI", 14),
    text_color=COLORS["text_primary"],  # Чорний для Light Mode
    fg_color="transparent",
    wrap="word",
```

```
            height=100,  # Збільшено для кращого відображення
            width=450,
            activate_scrollbars=True,
            border_width=0
        )
        self.wotd_definition_textbox.pack(padx=20, pady=20, fill="both",
expand=True)
        self.wotd_definition_textbox.insert("1.0", "Привіт! Вітаємо вас у E-
Dictionary Pro!'")
        self.wotd_definition_textbox.configure(state="disabled")

        # Hint
        ctk.CTkLabel(
            center_frame,
            text="💡 Натисніть 🕒 для перегляду історії пошуку та збережених слів",
            font=("Segoe UI", 11),
            text_color=COLORS["text_muted"]
        ).pack(pady=(30, 0))

        # Спробуємо завантажити слово дня
        self.after(500, self._refresh_word_of_the_day)

    def _show_results_screen(self):
        """Показати екран результатів (SCROLLABLE FIX)."""
        # Очищаємо контент
        for widget in self.content_frame.winfo_children():
            widget.destroy()

        # Results Container
        self.results_container = ctk.CTkFrame(self.content_frame,
fg_color="transparent")
        self.results_container.pack(fill="both", expand=True, padx=20, pady=10)

        # Results Header
        results_header = ctk.CTkFrame(self.results_container,
fg_color="transparent")
        results_header.pack(fill="x", pady=(0, 10)) # Менший відступ знизу

        ctk.CTkLabel(
            results_header,
            text="📋 Результати",
            font=("Segoe UI", 18, "bold"),
            text_color=COLORS["text_secondary"]
        ).pack(side="left")

        # Back to home button
        ctk.CTkButton(
            results_header,
            text="🏠 На головну",
            width=140,
            height=36,
            font=("Segoe UI", 12),
            fg_color=COLORS["border"],
            hover_color="#4a4a4a",
            text_color=COLORS["text_secondary"],
            corner_radius=8,
            command=self._show_start_screen
        ).pack(side="right", padx=(10, 0))
```

```
        # === CRITICAL FIX: SCROLLABLE FRAME ===
        # Використовуємо CTkScrollableFrame замість звичайного Frame
        self.results_frame = ctk.CTkScrollableFrame(
            self.results_container,
            fg_color="transparent",
            corner_radius=8,
            orientation="vertical" # Вертикальна прокрутка
        )
        self.results_frame.pack(fill="both", expand=True, pady=(10, 0))


    def save_new_word(self, word: str, definition: str, window, error_label=None):
        """
        Зберегти нове слово до словника.

        Args:
            word: Слово для додавання
            definition: Визначення слова
            window: Вікно popup для закриття після збереження
            error_label: Label для відображення помилок (опціонально)
        """
        # Clear any previous error message
        if error_label:
            error_label.configure(text="")

        # Validate input
        if not word or not definition:
            messagebox.showwarning("Validation Error", "Please fill in both
Headword and Definition fields!")
            return

        # Send ADD_WORD command via network
        if self.network.connected:
            response = self.network.send_command(f"ADD_WORD|{word}|{definition}")

            if response and response.startswith("Success"):
                # Show success message
                messagebox.showinfo("Success", f"Word '{word}' has been
successfully added to the dictionary!")
                # Close window and show success log
                window.destroy()
                self._add_to_log_panel(f"✅ Saved: {word}")
                logger.info(f"Додано слово: '{word}'")
                # Immediately display the added word with the same renderer as
searched words
                # First show results screen, then display translation
                self._show_results_screen()
                # Format: "word|definition" for _display_translation
                self._display_translation(word, f"{word}|{definition}")
            elif response and response.startswith("Error"):
                # Handle error response - show red error label and keep popup open
                error_message = "This word already exists!"
                if error_label:
                    error_label.configure(text=error_message)
                else:
                    # Fallback to messagebox if error_label not provided
                    messagebox.showerror("Error", error_message)
                logger.warning(f"Спроба додати існуюче слово: '{word}'")
            else:
```

```
                # Other errors - show messagebox
                messagebox.showerror("Error", f"Failed to add word: {response or
'Unknown error'}")
        else:
            # If offline, show warning
            messagebox.showwarning("Offline",
                "Cannot save word: not connected to server.\n\n"
                "Please connect first.")

    def _show_add_word_dialog(self):
        """
        Показати професійний ропир діалог для додавання нового слова.

        Відкриває CTkToplevel вікно з полями для введення слова
        та його перекладу. Виглядає як нативний діалог.
        """
        # Створюємо ропир вікно
        popup = ctk.CTkToplevel(self)
        popup.title("Add to Dictionary")
        popup.geometry("480x540")  # Збільшено для tag chips
        popup.resizable(False, False)

        # Центруємо відносно головного вікна
        popup.transient(self)
        popup.grab_set()

        # Налаштовуємо колір фону
        popup.configure(fg_color=COLORS["bg_main"])

        # Головний контейнер
        container = ctk.CTkFrame(popup, fg_color="transparent")
        container.pack(fill="both", expand=True, padx=25, pady=25)

        # Заголовок
        ctk.CTkLabel(
            container,
            text="Add to Dictionary",
            font=("Segoe UI", 20, "bold"),
            text_color=COLORS["text_primary"]
        ).pack(pady=(0, 25))

        # Поле: Headword
        headword_label = ctk.CTkLabel(
            container,
            text="Headword:",
            font=("Segoe UI", 13),
            text_color=COLORS["text_secondary"],
            anchor="w"
        )
        headword_label.pack(fill="x", pady=(0, 5))

        word_entry = ctk.CTkEntry(
            container,
            height=38,
            font=("Segoe UI", 14),
            placeholder_text="Enter word...",
            placeholder_text_color=COLORS["text_muted"],
            corner_radius=8,
            border_width=1,
```

```
    border_color=COLORS["border"]
)
word_entry.pack(fill="x", pady=(0, 15))

# Поле: Definition
definition_label = ctk.CTkLabel(
    container,
    text="Definition:",
    font=("Segoe UI", 13),
    text_color=COLORS["text_secondary"],
    anchor="w"
)
definition_label.pack(fill="x", pady=(0, 5))

# Use Textbox for definition (multi-line, scrollable)
definition_textbox = ctk.CTkTextbox(
    container,
    height=90,
    font=("Segoe UI", 13),
    corner_radius=8,
    wrap="word",
    border_width=1,
    border_color=COLORS["border"],
    activate_scrollbars=True
)
definition_textbox.pack(fill="x", pady=(0, 10))

# Error label (initially hidden)
error_label = ctk.CTkLabel(
    container,
    text="",
    font=("Segoe UI", 12),
    text_color=COLORS["danger"],
    anchor="w",
    height=20
)
error_label.pack(fill="x", pady=(0, 5))

# === FORMATTING HELPERS ===
# Instruction label
instruction_label = ctk.CTkLabel(
    container,
    text="Formatting Tips: Use tags to colorize parts of speech.",
    font=("Segoe UI", 11),
    text_color=COLORS["text_muted"],
    anchor="w"
)
instruction_label.pack(fill="x", pady=(0, 8))

# Helper function to insert tag at cursor position
def insert_tag(tag_text: str):
    """Insert tag text at current cursor position in definition textbox."""
    try:
        # Get current cursor position
        cursor_pos = definition_textbox.index("insert")
        # Insert the tag text
        definition_textbox.insert(cursor_pos, tag_text)
        # Move cursor after inserted text
        new_pos =
```

```
definition_textbox.index(f"{cursor_pos}+{len(tag_text)}c")
                definition_textbox.mark_set("insert", new_pos)
                definition_textbox.see("insert")
                # Focus back to textbox
                definition_textbox.focus()
            except Exception as e:
                logger.warning(f"Error inserting tag: {e}")
                # Fallback: append to end
                definition_textbox.insert("end", tag_text)
                definition_textbox.focus()

        # Chips frame for tag buttons - scrollable row
        chips_frame = ctk.CTkFrame(container, fg_color="transparent")
        chips_frame.pack(fill="x", pady=(0, 20))

        # Inner frame for buttons (will be scrollable if needed)
        chips_inner = ctk.CTkFrame(chips_frame, fg_color="transparent")
        chips_inner.pack(fill="x")

        # Tag buttons (chips) - all 7 tags
        tag_buttons = [
            ("[NOUN]", "[NOUN] "),
            ("[VERB]", "[VERB] "),
            ("[ADJ]", "[ADJ] "),
            ("[ADV]", "[ADV] "),
            ("[PHRASE]", "[PHRASE] "),
            ("[IT]", "[IT] "),
            ("[SCI]", "[SCI] ")
        ]

        for tag_label, tag_text in tag_buttons:
            chip_btn = ctk.CTkButton(
                chips_inner,
                text=tag_label,
                width=60,
                height=26,
                font=("Segoe UI", 10),
                fg_color="#87CEEB",  # Light blue (Sky Blue)
                hover_color="#6BB6FF",  # Slightly darker on hover
                text_color="#FFFFFF",
                corner_radius=12,
                command=lambda t=tag_text: insert_tag(t)
            )
            chip_btn.pack(side="left", padx=(0, 5))

        # Кнопки - Frame для правильного розташування
        btn_frame = ctk.CTkFrame(container, fg_color="transparent")
        btn_frame.pack(fill="x", pady=(10, 0))  # Додано явний padding

        def cancel():
            """Закрити popup."""
            popup.destroy()

        # Кнопка Cancel (Grey) - явно упакована
        cancel_btn = ctk.CTkButton(
            btn_frame,
            text="Cancel",
            width=100,
            height=38,
```

```
            font=("Segoe UI", 13),
            fg_color=COLORS["text_muted"],
            hover_color="#5A6268",
            text_color="#FFFFFF",
            corner_radius=8,
            command=cancel
        )
        cancel_btn.pack(side="left", padx=(0, 10))  # Додано явний padding

        # Spacer для відступу між кнопками
        spacer = ctk.CTkFrame(btn_frame, fg_color="transparent", width=20)
        spacer.pack(side="left", fill="x", expand=True)  # Розтягується для
вирівнювання

        # Кнопка Save (Green) - явно упакована з правильними параметрами
        save_btn = ctk.CTkButton(
            btn_frame,
            text="Save",
            width=100,
            height=38,
            font=("Segoe UI", 13, "bold"),
            fg_color=COLORS["success"],
            hover_color="#218838",
            text_color="#FFFFFF",
            corner_radius=8,
            command=lambda: self.save_new_word(
                word_entry.get().strip(),
                definition_textbox.get("1.0", "end-1c").strip(),
                popup,
                error_label
            )
        )
        save_btn.pack(side="right", padx=(10, 0))  # Додано явний padding для
видимості

        # Bind Enter для збереження (Ctrl+Enter for textbox)
        definition_textbox.bind('<Control-Return>',
            lambda e: self.save_new_word(
                word_entry.get().strip(),
                definition_textbox.get("1.0", "end-1c").strip(),
                popup,
                error_label
            )
        )
        word_entry.bind('<Return>',
            lambda e: definition_textbox.focus()
        )

        # Фокус на перше поле
        word_entry.focus()

    def _show_history_favorites_popup(self):
        """Показати професійний попап з історією та улюбленими словами
(CTkTabview)."""
        popup = ctk.CTkToplevel(self)
        popup.title("Saved & History")
        popup.geometry("500x600")
        popup.resizable(False, False)
        popup.transient(self)
```

```
        popup.grab_set()
        popup.configure(fg_color=COLORS["bg_main"])

        # Header
        header = ctk.CTkFrame(popup, fg_color=COLORS["bg_card"], corner_radius=0)
        header.pack(fill="x")

        ctk.CTkLabel(
            header,
            text="🕒 Saved & History",
            font=("Segoe UI", 18, "bold"),
            text_color=COLORS["text_primary"]
        ).pack(side="left", padx=20, pady=15)

        # Close button
        ctk.CTkButton(
            header,
            text="✕",
            width=32,
            height=32,
            font=("Segoe UI", 13),
            fg_color="transparent",
            text_color=COLORS["text_secondary"],
            hover_color=COLORS["border"],
            corner_radius=6,
            command=popup.destroy
        ).pack(side="right", padx=20, pady=15)

        # Tabview with History and Favorites tabs
        tabview = ctk.CTkTabview(popup, fg_color=COLORS["bg_card"])
        tabview.pack(fill="both", expand=True, padx=15, pady=(0, 15))

        # Tab 1: History
        history_tab = tabview.add("History")
        history_tab.configure(fg_color=COLORS["bg_main"])

        # Clear history button in History tab
        history_header = ctk.CTkFrame(history_tab, fg_color="transparent")
        history_header.pack(fill="x", padx=12, pady=(12, 8))

        ctk.CTkButton(
            history_header,
            text="Clear History",
            width=110,
            height=32,
            font=("Segoe UI", 11),
            fg_color="transparent",
            text_color=COLORS["text_secondary"],
            hover_color=COLORS["border"],
            corner_radius=6,
            command=lambda: self._clear_history_and_refresh(history_scroll_frame,
popup)
        ).pack(side="right")

        # Scrollable frame for history
        history_scroll_frame = ctk.CTkScrollableFrame(history_tab,
fg_color="transparent")
        history_scroll_frame.pack(fill="both", expand=True, padx=12, pady=(0, 12))
```

```
        # Tab 2: Favorites
        favorites_tab = tabview.add("Favorites")
        favorites_tab.configure(fg_color=COLORS["bg_main"])

        # Scrollable frame for favorites
        favorites_scroll_frame = ctk.CTkScrollableFrame(favorites_tab,
fg_color="transparent")
        favorites_scroll_frame.pack(fill="both", expand=True, padx=12, pady=12)

        # Function to handle word click (closes popup, sets search, triggers
translation)
        def search_word(word):
            popup.destroy()
            self.search_entry.delete(0, 'end')
            self.search_entry.insert(0, word)
            # Trigger translation if connected
            if self.network.connected:
                self._translate()
            else:
                messagebox.showwarning("No Connection", "Please connect to the
server first!")

        # Load and display History
        self._load_history_tab(history_scroll_frame, search_word, popup)

        # Load and display Favorites
        self._load_favorites_tab(favorites_scroll_frame, search_word, popup)

    def _load_history_tab(self, frame, search_callback, popup):
        """Завантажити та відобразити історію в frame з кнопками видалення."""
        # Clear existing widgets
        for widget in frame.winfo_children():
            widget.destroy()

        # Отримуємо історію з бази даних
        history_words = self.db.get_history_words(limit=50)

        if not history_words:
            ctk.CTkLabel(
                frame,
                text="No recent searches",
                font=("Segoe UI", 13),
                text_color=COLORS["text_muted"]
            ).pack(pady=50)
        else:
            for word in history_words:
                # Item frame with word button and delete button
                item_frame = ctk.CTkFrame(frame, fg_color="transparent")
                item_frame.pack(fill="x", pady=3)

                # Word button (clickable)
                word_btn = ctk.CTkButton(
                    item_frame,
                    text=f"🕒 {word}",
                    font=("Segoe UI", 13),
                    fg_color="transparent",
                    text_color=COLORS["text_primary"],
                    hover_color=COLORS["border"],
                    anchor="w",
```

```
                    height=38,
                    corner_radius=8,
                    command=lambda w=word: search_callback(w)
                )
                word_btn.pack(side="left", fill="x", expand=True, padx=(0, 5))

                # Delete button (X icon)
                delete_btn = ctk.CTkButton(
                    item_frame,
                    text="✕",
                    width=36,
                    height=38,
                    font=("Segoe UI", 12),
                    fg_color="transparent",
                    text_color=COLORS["text_muted"],
                    hover_color=COLORS["danger"],
                    corner_radius=8,
                    command=lambda w=word: self._delete_history_word(w, frame,
search_callback, popup)
                )
                delete_btn.pack(side="right")

    def _load_favorites_tab(self, frame, search_callback, popup):
        """Завантажити та відобразити улюблені слова в frame з кнопками
видалення."""
        # Clear existing widgets
        for widget in frame.winfo_children():
            widget.destroy()

        # Отримуємо улюблені з бази даних
        favorites = self.db.get_favorites()

        if not favorites:
            ctk.CTkLabel(
                frame,
                text="No favorite words",
                font=("Segoe UI", 13),
                text_color=COLORS["text_muted"]
            ).pack(pady=50)
        else:
            for word, translation in favorites:
                # Item frame with word button and delete button
                item_frame = ctk.CTkFrame(frame, fg_color="transparent")
                item_frame.pack(fill="x", pady=3)

                # Word button (clickable)
                word_btn = ctk.CTkButton(
                    item_frame,
                    text=f"☆ {word}",
                    font=("Segoe UI", 13),
                    fg_color="transparent",
                    text_color=COLORS["text_primary"],
                    hover_color=COLORS["border"],
                    anchor="w",
                    height=38,
                    corner_radius=8,
                    command=lambda w=word: search_callback(w)
                )
                word_btn.pack(side="left", fill="x", expand=True, padx=(0, 5))
```

```
                # Delete button (X icon)
                delete_btn = ctk.CTkButton(
                    item_frame,
                    text="✕",
                    width=36,
                    height=38,
                    font=("Segoe UI", 12),
                    fg_color="transparent",
                    text_color=COLORS["text_muted"],
                    hover_color=COLORS["danger"],
                    corner_radius=8,
                    command=lambda w=word: self._delete_favorite_word(w, frame,
search_callback, popup)
                )
                delete_btn.pack(side="right")

    def _delete_history_word(self, word: str, frame, search_callback, popup):
        """Видалити конкретне слово з історії."""
        success = self.db.remove_from_history(word)
        if success:
            # Reload the tab
            self._load_history_tab(frame, search_callback, popup)
            self._add_to_log_panel(f"🗑️ Removed '{word}' from history")
            logger.info(f"Видалено з історії: '{word}'")
        else:
            logger.warning(f"Не вдалося видалити з історії: '{word}'")

    def _delete_favorite_word(self, word: str, frame, search_callback, popup):
        """Видалити слово з улюблених."""
        success = self.db.remove_favorite(word)
        if success:
            # Reload the tab
            self._load_favorites_tab(frame, search_callback, popup)
            self._add_to_log_panel(f"🗑️ Removed '{word}' from favorites")
            logger.info(f"Видалено з улюблених: '{word}'")
        else:
            logger.warning(f"Не вдалося видалити з улюблених: '{word}'")

    def _clear_history_and_refresh(self, frame, popup):
        """Очистити історію та оновити відображення."""
        self._clear_history()
        # Reload history tab
        def search_callback(word):
            popup.destroy()
            self.search_entry.delete(0, 'end')
            self.search_entry.insert(0, word)
            if self.network.connected:
                self._translate()
        self._load_history_tab(frame, search_callback, popup)

    def _bind_shortcuts(self):
        """Прив'язка клавіатурних скорочень."""
        self.bind('<Control-h>', lambda e: self._show_about())
        self.bind('<Control-H>', lambda e: self._show_about())
        self.bind('<Escape>', lambda e: self._focus_search())

    def _focus_search(self):
        """Встановлення фокусу на поле пошуку."""
```

```
        try:
            if hasattr(self, 'search_entry') and self.search_entry.winfo_exists():
                self.search_entry.focus()
        except Exception:
            pass

    # === FUNCTIONALITY ===

    def _show_connection_warning(self):
        """Показати попередження про відсутність з'єднання."""
        # Створюємо модальне вікно попередження
        warning = ctk.CTkToplevel(self)
        warning.title("⚠ Сервер недоступний")
        warning.geometry("450x280")
        warning.resizable(False, False)
        warning.transient(self)
        warning.grab_set()
        warning.configure(fg_color=COLORS["bg_main"])

        # Контейнер
        container = ctk.CTkFrame(warning, fg_color="transparent")
        container.pack(fill="both", expand=True, padx=30, pady=30)

        # Іконка та заголовок
        ctk.CTkLabel(
            container,
            text="🟢",
            font=("Segoe UI", 48)
        ).pack(pady=(0, 10))

        ctk.CTkLabel(
            container,
            text="Сервер недоступний",
            font=("Segoe UI", 18, "bold"),
            text_color=COLORS["text_primary"]
        ).pack()

        ctk.CTkLabel(
            container,
            text=f"Не вдалося підключитися до
{self.network.host}:{self.network.port}\n\n"
                 "Переконайтесь, що сервер запущено,\n"
                 "або натисніть 'Connect' для повторної спроби.",
            font=("Segoe UI", 12),
            text_color=COLORS["text_secondary"],
            justify="center"
        ).pack(pady=15)

        # Кнопки
        btn_frame = ctk.CTkFrame(container, fg_color="transparent")
        btn_frame.pack(fill="x", pady=(10, 0))

        ctk.CTkButton(
            btn_frame,
            text="OK",
            width=120,
            height=40,
            font=("Segoe UI", 12, "bold"),
            fg_color="#007AFF",
```

```
            hover_color="#0056B3",
            corner_radius=8,
            command=warning.destroy
        ).pack()

    def _update_ui_state(self, connected: bool):
        """
        Оновлює стан UI елементів залежно від статусу з'єднання.

        Args:
            connected: True якщо підключено, False якщо ні.
        """
        if connected:
            # Розблоковуємо UI
            self.search_entry.configure(
                state="normal",
                placeholder_text=""
            )
            self.search_btn.configure(
                state="normal",
                fg_color="#10B981",
                hover_color="#059669"
            )
            # Ставимо фокус на пошуковий рядок після підключення
            self.after(100, lambda: self.search_entry.focus_set())
            self._add_to_log_panel("✅ UI розблоковано")
        else:
            # Блокуємо UI
            self.search_entry.configure(
                state="disabled",
                placeholder_text=""
            )
            self.search_btn.configure(
                state="disabled",
                fg_color="#6B7280",
                hover_color="#6B7280"
            )

    def _toggle_connection(self):
        """Перемикання з'єднання з сервером (відключення синхронне, підключення в
фоновому потоці)."""
        if self.network.connected:
            # Відключення - синхронне, оскільки швидке
            try:
                self.network.disconnect()
                self.status_indicator.set_offline()
                self.connect_btn.configure(text="Connect", fg_color="#007AFF")
                self._update_ui_state(False)
                self._add_to_log_panel("Від'єднано від сервера")
            except Exception as e:
                logger.error(f"[UI] Помилка відключення: {e}")
                self.status_indicator.set_offline()
                self.connect_btn.configure(text="Connect", fg_color="#007AFF")
                self._update_ui_state(False)
        else:
            self.status_indicator.set_connecting()
            self.connect_btn.configure(text="...", state="disabled")
            self.update()
```

```
            host = self.host_entry.get().strip() or "127.0.0.1"
            try:
                port = int(self.port_entry.get().strip() or "8080")
            except ValueError:
                port = 8080

            self.network.host = host
            self.network.port = port

            # Запускаємо підключення в фоновому потоці
            def connect_thread():
                try:
                    connected = self.network.connect()
                    # Оновлюємо UI в головному потоці
                    self.after(0, lambda: self._on_connect_result(connected, host,
port))
                except Exception as e:
                    logger.error(f"[UI] Помилка підключення: {e}")
                    self.after(0, lambda: self._on_connect_result(False, host,
port))

            thread = threading.Thread(target=connect_thread, daemon=True)
            thread.start()
            self._network_threads.append(thread)

    def _on_connect_result(self, connected: bool, host: str, port: int):
        """Обробка результату підключення (викликається в UI потоці)."""
        if connected:
            self.status_indicator.set_online()
            self.connect_btn.configure(text="Disconnect", state="normal",
fg_color="#EF4444", hover_color="#DC2626")
            self._update_ui_state(True)
            self._add_to_log_panel(f"Підключено до {host}:{port}")
            # Оновлюємо Word of the Day
            self.after(200, self._refresh_word_of_the_day)
        else:
            self.status_indicator.set_offline()
            self.connect_btn.configure(text="Connect", state="normal",
fg_color="#007AFF")
            self._update_ui_state(False)
            self._add_to_log_panel(f"Помилка підключення: {host}:{port}")
            messagebox.showerror("Помилка з'єднання",
                f"Не вдалося підключитися до сервера!\n\n"
                f"Адреса: {host}:{port}\n\n"
                f"Переконайтесь, що сервер запущено."
            )

    def _translate(self):
        """Виконання перекладу слова (в фоновому потоці)."""
        # Empty input protection - do nothing if input is empty
        search_term = self.search_entry.get().strip()
        if not search_term:
            return

        word = search_term.lower()

        if not self.network.connected:
            messagebox.showwarning("Немає з'єднання", "Спочатку підключіться до
сервера!")
```

```
            return

        self.search_btn.configure(text="🔍 Думаю...", state="disabled")
        self.update()

        # Показуємо results screen одразу
        self._show_results_screen()

        # Запускаємо переклад в фоновому потоці
        def translate_thread():
            try:
                response = self.network.translate(word)
                # Оновлюємо UI в головному потоці
                self.after(0, lambda: self._on_translate_result(word, response))
            except Exception as e:
                logger.error(f"[UI] Помилка перекладу: {e}")
                self.after(0, lambda: self._on_translate_result(word, None))

        thread = threading.Thread(target=translate_thread, daemon=True)
        thread.start()
        self._network_threads.append(thread)

    def _on_translate_result(self, word: str, response: str | None):
        """Обробка результату перекладу (викликається в UI потоці)."""
        self.search_btn.configure(text="🔍 Translate", state="normal")

        if response is None or response == "":
            self.status_indicator.set_offline()
            self.connect_btn.configure(text="Connect")
            self._add_to_log_panel(f"Втрачено з'єднання: '{word}'")
            messagebox.showerror("Помилка", "З'єднання з сервером втрачено!")
        elif response == "NOT_FOUND" or (isinstance(response, str) and
response.strip().lower() in ["not found", "error", "notfound"]):
            self._show_not_found(word)
            self._add_to_log_panel(f"Не знайдено: '{word}'")
            # DO NOT save "Not found" to History!
        else:
            self._display_translation(word, response)
            self._add_to_log_panel(f"{word} → ...")
            # Note: History is now saved in _display_translation() immediately
after parsing

    def _add_word(self, ukr: str, eng: str) -> bool:
        """
        Додавання нового слова до словника (в фоновому потоці).

        Returns:
            bool: False одразу (результат буде показано через callback)
        """
        if not ukr or not eng:
            messagebox.showwarning("Відсутні дані", "Введіть обидва слова!")
            return False

        if not self.network.connected:
            messagebox.showwarning("Немає з'єднання", "Спочатку підключіться!")
            return False

        # Запускаємо додавання в фоновому потоці
        def add_word_thread():
```

```
            try:
                response = self.network.add_word(ukr, eng)
                # Оновлюємо UI в головному потоці
                self.after(0, lambda: self._on_add_word_result(ukr, eng, response))
            except Exception as e:
                logger.error(f"[UI] Помилка додавання слова: {e}")
                self.after(0, lambda: self._on_add_word_result(ukr, eng, None))

        thread = threading.Thread(target=add_word_thread, daemon=True)
        thread.start()
        self._network_threads.append(thread)
        return False  # Результат буде показано через callback

    def _on_add_word_result(self, ukr: str, eng: str, response: str | None):
        """Обробка результату додавання слова (викликається в UI потоці)."""
        if response == "ADDED":
            self._add_to_log_panel(f"Додано: {ukr} = {eng}")
            messagebox.showinfo("Успішно", f"Слово '{ukr}' додано!")
        elif response == "EXIST":
            self._add_to_log_panel(f"Дублікат: '{ukr}'")
            messagebox.showwarning("Дублікат", f"Слово '{ukr}' вже існує!")
        else:
            self._add_to_log_panel(f"Помилка: '{ukr}'")
            messagebox.showerror("Помилка", "Не вдалося додати слово.")

    def _display_translation(self, search_query, raw_response):
        """
        Відображення результатів перекладу (English → Ukrainian).

        Формат відповіді сервера: "Word|Definition"

        Args:
            search_query: Запит користувача (англійське слово)
            raw_response: Відповідь сервера
        """
        logger.info(f"Переклад: '{search_query}'")

        # === CRASH PROTECTION: Handle plain text responses without "|" separator
===
        if "|" in raw_response:
            # Normal response format: "Word|Definition"
            headword, definition_body = raw_response.split("|", 1)
            headword = headword.strip()
            definition_body = definition_body.strip()

            # === CRITICAL: Force save to history immediately after successful
parsing ===
            clean_headword = headword.strip()
            if clean_headword and definition_body:
                self.db.add_to_history(clean_headword, definition_body)

            # Store current word for favorites toggle
            self.current_headword = clean_headword
            self.current_definition = definition_body

            # Check if word is favorite
            is_favorite = self.db.is_favorite(clean_headword)
        else:
            # Handle plain text response (e.g. "Not found", "Error")
```

```
            # Use the user's search query as headword instead of "System Message"
            headword = self.search_var.get().strip() if self.search_var.get() else
search_query.strip()
            definition_body = raw_response.strip()
            clean_headword = headword

            # Store current word for favorites toggle (even if it's an error)
            self.current_headword = clean_headword
            self.current_definition = definition_body

            is_favorite = False

            # Check if it's an error/not found message
            response_lower = raw_response.strip().lower()
            if response_lower in ["not found", "error", "notfound"]:
                # Don't save error messages to history
                logger.warning(f"Server returned error message: '{raw_response}'")
            else:
                # If it's some other plain text (unexpected), save to history if
valid
                if clean_headword and definition_body:
                    self.db.add_to_history(clean_headword, definition_body)
                logger.warning(f"Unexpected response format: '{raw_response}'")

        # === Форматуємо визначення ===
        formatted_definition = format_and_display(definition_body,
headword=headword)

        # === Створюємо картку результату ===
        result_card = ResultCard(
            self.results_frame,
            headword=headword,
            definition=formatted_definition,
            favorite_callback=self._handle_favorite_toggle,
            is_favorite=is_favorite
        )
        result_card.pack(fill="x", pady=10)

    def _show_not_found(self, word):
        """Показати повідомлення про ненайдене слово."""
        frame = ctk.CTkFrame(self.results_frame, fg_color=COLORS["bg_card"],
corner_radius=12)
        frame.pack(fill="x", pady=10)

        ctk.CTkLabel(frame, text="☹", font=("Segoe UI Emoji", 40)).pack(pady=(20,
10))
        ctk.CTkLabel(
            frame,
            text=f"'{word}' не знайдено",
            font=("Segoe UI", 16, "bold"),
            text_color=COLORS["warning"]
        ).pack()
        ctk.CTkLabel(
            frame,
            text="Спробуйте інше слово або додайте до словника.",
            font=("Segoe UI", 12),
            text_color=COLORS["text_secondary"]
        ).pack(pady=(5, 20))
```

```
    def _show_placeholder(self, message):
        """Показати placeholder в області результатів."""
        ctk.CTkLabel(
            self.results_frame,
            text="📖",
            font=("Segoe UI Emoji", 48),
            text_color=COLORS["text_muted"]
        ).pack(pady=(50, 10))

        ctk.CTkLabel(
            self.results_frame,
            text=message,
            font=("Segoe UI", 14),
            text_color=COLORS["text_muted"]
        ).pack()

    def _clear_results(self):
        """Очищення області результатів."""
        for widget in self.results_frame.winfo_children():
            widget.destroy()

    def _clear_history(self):
        """Очищення історії пошуку."""
        self.db.clear_history()
        self._add_to_log_panel("Історію очищено")

    def _add_to_log_panel(self, message):
        """Виводить повідомлення в консоль та логер."""
        timestamp = datetime.now().strftime("%H:%M:%S")
        print(f"[{timestamp}] {message}")
        logger.info(message)

    def _copy_wotd(self):
        """Копіювання Word of the Day в буфер обміну (тільки переклад)."""
        try:
            word = self.wotd_word_label.cget("text")
            # Отримуємо текст з textbox (не label)
            raw_definition = self.wotd_definition_textbox.get("1.0", "end").strip()

            # Очищаємо від заголовків POS
            clean_definition = self._clean_definition_for_copy(raw_definition,
word)

            self.clipboard_clear()
            self.clipboard_append(clean_definition)

            # Показуємо галочку на 1.5 сек
            self.wotd_copy_btn.configure(text="✓")
            self.after(1500, lambda: self.wotd_copy_btn.configure(text="📋"))

            logger.info(f"Скопійовано Word of the Day (переклад)")
        except Exception as e:
            logger.error(f"Помилка копіювання WOTD: {e}")

    def _clean_definition_for_copy(self, text: str, headword: str = None) -> str:
        """
        Очистити текст визначення для копіювання.
        Видаляє заголовки POS, headword та зайве форматування.
```

```
        Args:
            text: Сирий текст визначення
            headword: Слово-заголовок для видалення

        Returns:
            Чистий текст перекладу
        """
        # Видаляємо заголовки частин мови типу [ NOUN ], [ VERB ] тощо
        text =
re.sub(r'\[\s*(NOUN|VERB|ADJECTIVE|ADVERB|PREPOSITION|CONJUNCTION|PRONOUN|INTERJECT
ION|NUMERAL|PHRASAL VERB)\s*\]', '', text)

        # Видаляємо headword якщо він на початку
        if headword:
            pattern = rf'^\s*{re.escape(headword)}\s*\n?'
            text = re.sub(pattern, '', text, flags=re.IGNORECASE)

        # Очищаємо зайві переноси рядків та пробіли
        text = re.sub(r'\n{3,}', '\n\n', text)
        text = re.sub(r'^\s+', '', text, flags=re.MULTILINE)
        text = text.strip()

        return text

    def _refresh_word_of_the_day(self):
        """Оновити слово дня з сервера."""
        if not self.network.connected:
            self.wotd_word_label.configure(text="Offline")
            self._update_wotd_textbox("Підключіться для слова дня")
            return

        try:
            # Запитуємо випадкове слово через network
            response = self.network.send_command("GET_RANDOM|")

            if response and response != "NOT_FOUND" and '|' in response:
                parts = response.split('|', 1)
                word = parts[0].strip()
                definition = parts[1].strip() if len(parts) > 1 else ""

                # Форматуємо визначення через наш Human-Readable Formatter
                formatted_definition = format_and_display(definition,
headword=word)

                self.wotd_word_label.configure(text=word.title())
                self._update_wotd_textbox(formatted_definition)
                logger.info(f"Слово дня: {word}")
            else:
                # Якщо сервер не підтримує GET_RANDOM - показуємо заглушку
                self.wotd_word_label.configure(text="Hello")
                self._update_wotd_textbox("Привіт! Вітання, формальне або
неформальне.")
        except Exception as e:
            logger.error(f"Помилка отримання слова дня: {e}")
            self.wotd_word_label.configure(text="—")
            self._update_wotd_textbox("Недоступно")

    def _update_wotd_textbox(self, text: str):
        """Оновити текстове поле слова дня з форматуванням та кольоровими
```

```
тегами."""
        try:
            if hasattr(self, 'wotd_definition_textbox') and
self.wotd_definition_textbox.winfo_exists():
                self.wotd_definition_textbox.configure(state="normal")
                # Використовуємо Deep Sky Blue для тегів у WotD (щоб відрізнятися
від зеленого headword)
                insert_formatted_text(self.wotd_definition_textbox, text,
tag_color="#00BFFF")
                self.wotd_definition_textbox.configure(state="disabled")
        except Exception as e:
            logger.error(f"Помилка оновлення WOTD textbox: {e}")

    def _safe_focus_search(self):
        """Безпечне встановлення фокусу з захистом від TclError."""
        import tkinter as tk
        try:
            if hasattr(self, 'search_entry') and self.search_entry.winfo_exists():
                self.search_entry.focus_set()
        except (tk.TclError, AttributeError):
            pass  # Ігноруємо помилки фокусу

    def _handle_favorite_toggle(self, word: str, definition: str, is_favorite:
bool):
        """
        Обробка toggle улюблених слів.

        Оновлює базу даних та забезпечує синхронізацію стану зірки.
        """
        # Use stored current values if available, otherwise use passed parameters
        clean_word = (self.current_headword or word).strip()
        clean_definition = (self.current_definition or definition).strip()

        if is_favorite:
            # Додаємо до улюблених
            success = self.db.add_favorite(clean_word, clean_definition)
            if success:
                logger.info(f"Додано до улюблених: '{clean_word}'")
                self._add_to_log_panel(f"☆ Додано до улюблених: '{clean_word}'")
            else:
                # Word might already be in favorites - verify and sync
                if self.db.is_favorite(clean_word):
                    logger.info(f"Слово вже в улюблених: '{clean_word}'")
                else:
                    logger.warning(f"Не вдалося додати до улюблених:
'{clean_word}'")
        else:
            # Видаляємо з улюблених
            success = self.db.remove_favorite(clean_word)
            if success:
                logger.info(f"Видалено з улюблених: '{clean_word}'")
                self._add_to_log_panel(f"☆ Видалено з улюблених: '{clean_word}'")
            else:
                # Word might not be in favorites - verify and sync
                if not self.db.is_favorite(clean_word):
                    logger.info(f"Слово не в улюблених: '{clean_word}'")
                else:
                    logger.warning(f"Не вдалося видалити з улюблених:
'{clean_word}'")
```

```
def _show_about(self):
    """Показати діалог 'Про програму'."""
    messagebox.showinfo(
        "Про E-Dictionary Pro",
        "🪶 Електронний словник v2.0\n"
        "━━━━━━━━━━━━━━━━━━━━━━━━\n\n"
        "Курсова робота 2025\n\n"
        "Розробник: Дмитро Петрунів\n"
        "Бекенд: C++ (Winsock2)\n"
        "Фронтенд: Python (CustomTkinter)\n\n"
        "━━━━━━━━━━━━━━━━━━━━━━━━\n"
        "Натисніть Ctrl+H щоб побачити це."
    )
```