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

Тема: Монітор активності

Курсова робота
З дисципліни «Технології розроблення програмного забезпечення»

Керівник
доц. Амонс О.А.
«Допущений до захисту»
_____
(Особистий підпис керівника)
« » _____2025р.
Захищений з оцінкою
_____
(оцінка)

Виконавець
ст. Лисенко В. Є.
залікова книжка № ___ – _____
гр. ІА-31
_____
(особистий підпис виконавця)
« »_____2025р.

Члени комісії:

_____ _____
(особистий підпис) (розшифровка підпису)
_____ _____
(особистий підпис) (розшифровка підпису)

Київ – 2025

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

Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Дисципліна «Технології розроблення програмного забезпечення»

Курс 3 Група ІА-31 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Лисенка Владислава Євгенійовича

(прізвище, ім’я, по батькові)

1.Тема роботи System activity monitor

2. Строк здачі студентом закінченої роботи

3. Вихідні дані до роботи:

Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці)

1. Аналіз предметної області, загальний опис проєкту та огляд існуючих рішень. 2. Формулювання функціональних та нефункціональних вимог до системи. 3. Розробка сценаріїв використання (Use Case) та концептуальної моделі системи. 4. Проєктування архітектури системи, вибір технологічного стеку та баз даних.

Додатки:

Додаток А – режим доступу до проєкту

Додаток Б - проектування паттернів

5.Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

Рис. 1.1 — Діаграма варіантів використання; Рис. 1.2–1.4 — Діаграми послідовності; Рис. 1.5 — Діаграма класів — Діаграма компонентів; Рис. 1.7 — Діаграма розгортання, Рис. 2.1 – Проектування бази даних, Рис. 2.2 – 2.5 – відображення проектування паттернів

6. Дата видачі завдання 17.09.2025

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Підбір та вивчення літератури	30.09.2025	
2.	Проектування та написання розділу 1	31.10.2025	
3.	Розробка та написання розділу 2	20.11.2025	
4.	Подання курсової роботи на перевірку	25.11.2025	
5.	Захист курсової роботи	08.12.2025	
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			
18.			

Студент _____ _____Владислав ЛИСЕНКО_____
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник_____ __Олександр АМОНС__
(підпис) (Ім’я ПРІЗВИЩЕ)

«____»_____ 20___ р.

ЗМІСТ

ВСТУП

Сучасні операційні системи є складними програмними комплексами, які одночасно керують сотнями процесів та потоків. Для розробників програмного забезпечення та системних адміністраторів критично важливо розуміти принципи розподілу системних ресурсів, таких як процесорний час та оперативна пам'ять.

Актуальність даної роботи полягає у необхідності створення інструментів, які дозволяють не лише моніторити поточний стан системи, але й моделювати (емулювати) поведінку процесів у різних станах. Це дає можливість наочно досліджувати, як додатки впливають на продуктивність комп'ютера, та тестувати механізми керування ними.

Метою даного курсового проєкту є проєктування та розробка програмної системи "System Activity Monitor". Система має виконувати функції диспетчера завдань: відображати список активних процесів, розраховувати їхнє навантаження на CPU та RAM, надавати можливість керування ними (запуск, зупинка), а також збирати аналітичну статистику.

Для досягнення поставленої мети буде розроблено desktop-додаток на платформі .NET (WPF). Особливу увагу приділено архітектурі системи та практичному застосуванню патернів проєктування (GoF). Це дозволить створити гнучку, розширювану систему з чітким розділенням бізнес-логіки та інтерфейсу.

У роботі буде проведено аналіз існуючих аналогів, спроєктовано структуру бази даних для збереження історії активності, розроблено сценарії використання та продемонстровано реалізацію таких патернів, як Command, Observer, State та Visitor.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

Моніторинг активності системи є критично важливим аспектом керування обчислювальними ресурсами. Існуючі програмні рішення дозволяють користувачам та адміністраторам відстежувати стан процесів, споживання оперативної пам'яті (RAM) та навантаження на центральний процесор (CPU). Для проєктування власної системи було проаналізовано найпопулярніші аналоги, що використовуються в середовищі Windows.

1. Windows Task Manager (Диспетчер завдань Windows) Це стандартна утиліта, вбудована в операційні системи сімейства Microsoft Windows.

- Переваги: Доступність "з коробки", інтуїтивно зрозумілий інтерфейс, можливість завершувати процеси, перегляд базової статистики продуктивності.
- Недоліки: Обмежені можливості для глибокого аналізу (наприклад, неможливо переглянути детальну історію дій користувача, таку як кліки миші), відсутність рольової моделі (інтерфейс однаковий для всіх користувачів), неможливість симуляції станів системи для навчальних цілей.

2. Process Explorer (Sysinternals) Просунутий інструмент для моніторингу процесів.

- Переваги: Відображення ієрархічної структури процесів, детальна інформація про дескриптори (handles) та бібліотеки DLL, потужні інструменти діагностики.
- Недоліки: Перевантажений інтерфейс, складний для пересічного користувача, відсутність функції збереження історії сесій у базу даних для подальшого аудиту.

3. Process Hacker Багатофункціональний інструмент з відкритим вихідним кодом.

- Переваги: Повний контроль над службами та мережевою активністю, можливість редагування пам'яті процесів.

- Недоліки: Вимагає встановлення драйверів ядра для повного функціоналу, що створює ризики безпеки; відсутність навчального режиму емуляції.

1.2. Загальний опис проєкту

Проєкт присвячений розробці програмної системи "System Activity Monitor", призначеної для емуляції та моніторингу процесів в операційному середовищі, а також аналізу споживання системних ресурсів. На відміну від стандартних утиліт моніторингу (як Windows Task Manager), дана система проєктується як навчальний емулятор операційної системи, що дозволяє моделювати поведінку програмного забезпечення, створювати штучні критичні стани (зависання, витоки пам'яті) та демонструвати взаємодію компонентів через використання сучасних патернів проєктування (Command, Observer, State, Visitor).

Система базується на подійно-орієнтованій архітектурі (Event-Driven Architecture) з чітким розділенням на рівні логіки та візуалізації.

- Ядро системи (System Core): Реалізоване мовою C# (.NET). Виступає центральним контролером, який керує життєвим циклом процесів, розраховує навантаження на CPU та RAM, обробляє команди користувача та забезпечує логування подій у базу даних (SQLite).
- Інтерфейс користувача (UI): Реалізований на платформі WPF (Windows Presentation Foundation). Забезпечує візуалізацію "Робочого столу", емуляцію вікон прикладних програм (Google Chrome, Visual Studio) та відображення динамічних графіків активності в реальному часі.

Проєкт чітко розмежовує функціонал та права доступу на основі двох ключових ролей:

1. Адміністратор (System Administrator): Володіє повними правами на діагностику та керування стабільністю системи. Його задачі включають:

- Кризовий менеджмент: Примусове завершення ("Kill") або заморожування ("Freeze") процесів для тестування реакції системи на збої.
- Глибока аналітика: Використання патерну *Visitor* для отримання детальних звітів про розподіл ресурсів між типами програм (браузери, IDE, системні служби).
- Аудит: Перегляд історії сесій та архіву системних логів, очищення бази даних.

2. Користувач (User): Емулює поведінку офісного працівника в ізольованому середовищі "Робочого столу". Функціонал Користувача включає:
   - Робота з програмами: Запуск емульованих додатків (Google Chrome, Visual Studio) через інтерактивні іконки.
   - Генерація навантаження: Динамічне створення вкладок у браузері або відкриття проєктів в IDE, що призводить до зростання споживання оперативної пам'яті та процесорного часу в реальному часі.
   - Моніторинг: Перегляд власної статистики активності та керування запущеними вікнами.

Основний сценарій роботи передбачає повний цикл імітації: від авторизації користувача та завантаження "Робочого столу", запуску програм та нарощування навантаження (відкриття нових вкладок), до фіксації дій користувача (імітація вводу клавіатури/миші) та реактивного відображення змін у модулі моніторингу. Система автоматично виявляє стан "простою" (Idle) та захищає критичні системні процеси від випадкового завершення.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

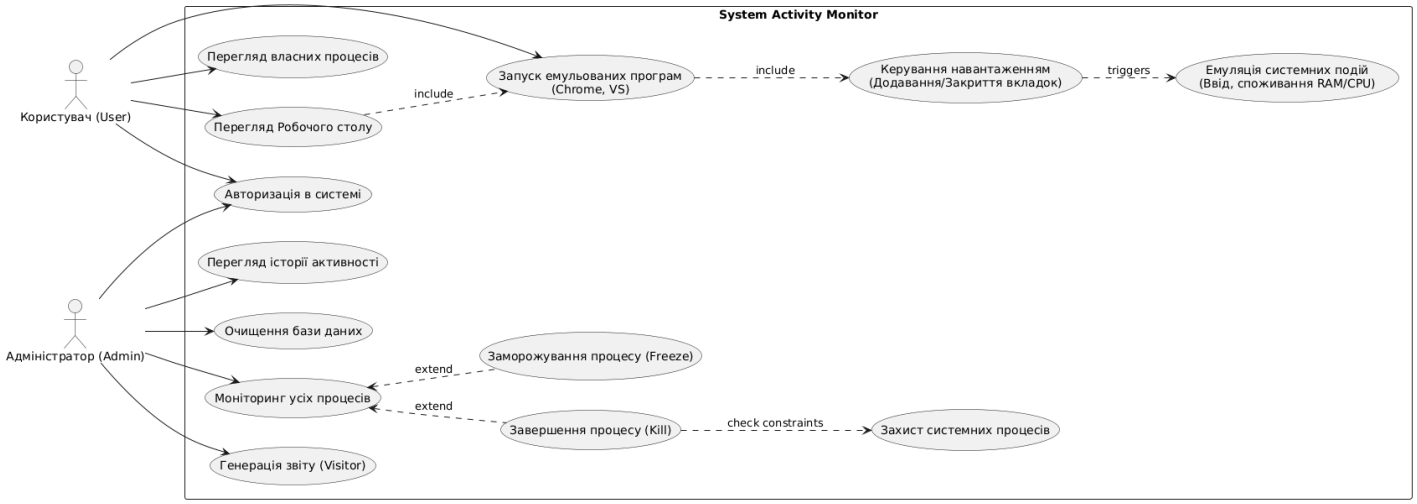

Рис. 1. 1 - Діаграма варіантів використання

Таблиця 1.1. Функціональні вимоги

Підсистема	Вимога
Підсистема авторизації та розмежування прав	Система повинна забезпечувати авторизацію користувачів за логіном та паролем.
	Система повинна підтримувати дві ролі: "Користувач" та "Адміністратор".
	Система повинна автоматично створювати нову сесію (Session ID) при успішному вході.
Підсистема емуляції активності (User Mode)	Система повинна надавати інтерфейс "Робочого столу" з іконками доступних програм.
	Система повинна емулювати запуск додатків (Google Chrome, Visual Studio) у вигляді окремих вікон.
	Система повинна дозволяти динамічно змінювати навантаження на ресурси шляхом додавання та закриття вкладок у емульованих програмах.

	Система повинна візуально відображати інтерфейс програм (скріншоти або стилізовані елементи).
Підсистема моніторингу та керування	Система повинна відображати список усіх активних процесів у реальному часі.
	Система повинна розраховувати споживання CPU та RAM для кожного процесу на основі його стану та інтенсивності роботи.
	Система повинна надавати можливість примусового завершення процесів (команда Kill).
	Система повинна блокувати спроби завершення критичних системних процесів (System, Registry).
	Система повинна підтримувати зміну станів процесів (Running, Frozen/Suspended) для тестування.
Підсистема аналітики та логування	Система повинна автоматично фіксувати (емулювати) події вводу (натискання клавіш, кліки миші) та зберігати їх у базі даних.
	Система повинна періодично зберігати зрізи стану ресурсів (Resource Logs).
	Система повинна генерувати зведений звіт про використання ресурсів за допомогою патерну Visitor (розподіл пам'яті за типами ПЗ).

1.3.2. Нефункціональні вимоги до системи

Таблиця 1.2. Нефункціональні вимоги

Назва	Опис

Вимоги до інтерфейсу	• Інтерфейс користувача повинен бути реалізований на технології WPF (Windows Presentation Foundation). • Графічний інтерфейс повинен бути інтуїтивно зрозумілим та наслідувати стиль ОС Windows (System Blue, віконний режим). • Оновлення даних у таблиці процесів має відбуватися плавно, без мерехтіння (використання ObservableCollection).
Вимоги до продуктивності	• Період оновлення даних моніторингу повинен становити 1 секунду (Real-time). • Система повинна підтримувати одночасну роботу не менше 20 емульованих процесів без значних затримок інтерфейсу.
Архітектурні вимоги	• Система повинна бути побудована з використанням об'єктно-орієнтованих патернів проєктування: Command (для керування процесами), Observer (для оновлення UI), State (для станів процесів), Visitor (для звітів). • Архітектура повинна забезпечувати слабку зв'язність (Loose Coupling) між логікою (SystemController) та представленням (UI).
Вимоги до даних	• Всі дані про користувачів, сесії та події повинні зберігатися у локальній реляційній базі даних (SQLite). • Система повинна забезпечувати цілісність даних при аварійному завершенні роботи (транзакційність на рівні EF Core).
Вимоги до надійності	Система не повинна аварійно завершувати роботу при спробі видалення неіснуючого або захищеного процесу (обробка виключень).

1.4. Сценарії використання системи

Таблиця 1.3. Сценарій використання «Емуляція активності користувача»

Поле	Опис
Назва	Запуск програм та емуляція навантаження
Передумови	Користувач успішно авторизувався в системі. Відкрито вікно "Робочий стіл".
Постумови	У системі створено нові процеси, графіки CPU/RAM відображають ріст навантаження.
Взаємодіючі сторони	Користувач, Робочий стіл (Desktop), Ядро системи (Controller).
Короткий опис	Користувач запускає програму (наприклад, Chrome) та збільшує споживання ресурсів, відкриваючи нові вкладки.
Основний потік подій	1. Користувач натискає іконку "Google Chrome" на Робочому столі. 2. Система створює новий процес VirtualProcess та візуальне вікно. 3. Користувач натискає кнопку "+" (New Tab) у вікні програми. 4. Система оновлює стан процесу, збільшуючи споживання RAM. 5. Користувач закриває вікно програми (хрестик). 6. Система завершує процес та звільняє ресурси.
Винятки	Критична помилка емуляції (нестача пам'яті). Система виводить повідомлення, процес не створюється.
Примітки	Використовується патерн Command для запуску та завершення процесів.

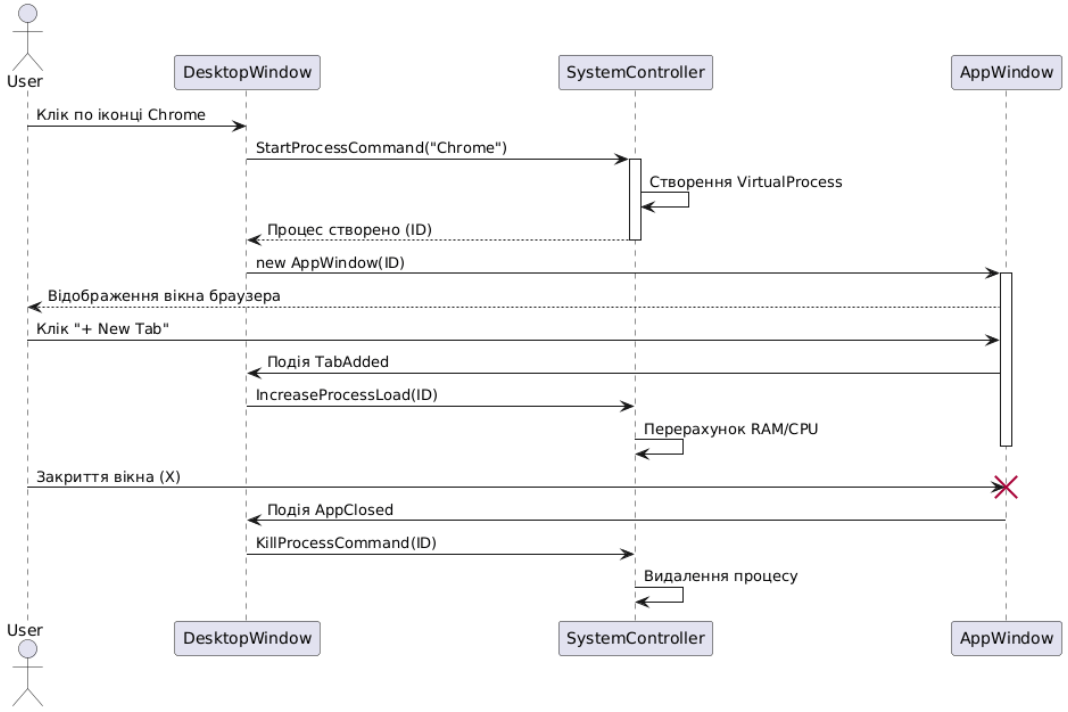

Рис. 1. 2 - Діаграма послідовності для сценарію «Емуляція активності користувача»

Таблиця 1.4. Сценарій використання «Адміністративний контроль»

Поле	Опис
Назва	Примусове завершення процесу та захист системи
Передумови	Адміністратор авторизований. Відкрито вікно моніторингу. У списку є системні процеси.
Постумови	Обраний процес завершено (якщо це дозволено) або виведено повідомлення про заборону.
Взаємодіючі сторони	Адміністратор, Монітор (View), Контролер (Controller).
Короткий опис	Адміністратор намагається завершити процес через список завдань. Система перевіряє права доступу.
Основний потік подій	1. Адміністратор переглядає список активних процесів. 2. Адміністратор натискає кнопку " ✕ Прибрати задачу" навпроти процесу. 3. Система перевіряє, чи є процес захищеним (System, Registry). 4. Якщо процес звичайний — система видаляє його та оновлює список. 5. Якщо процес захищений — система генерує виняток.
Винятки	Спроба видалення системного процесу. Система перехоплює помилку та показує MessageBox: "Доступ заборонено! Процес є критичним".
Примітки	Реалізовано механізм захисту цілісності ядра (Kernel Protection).

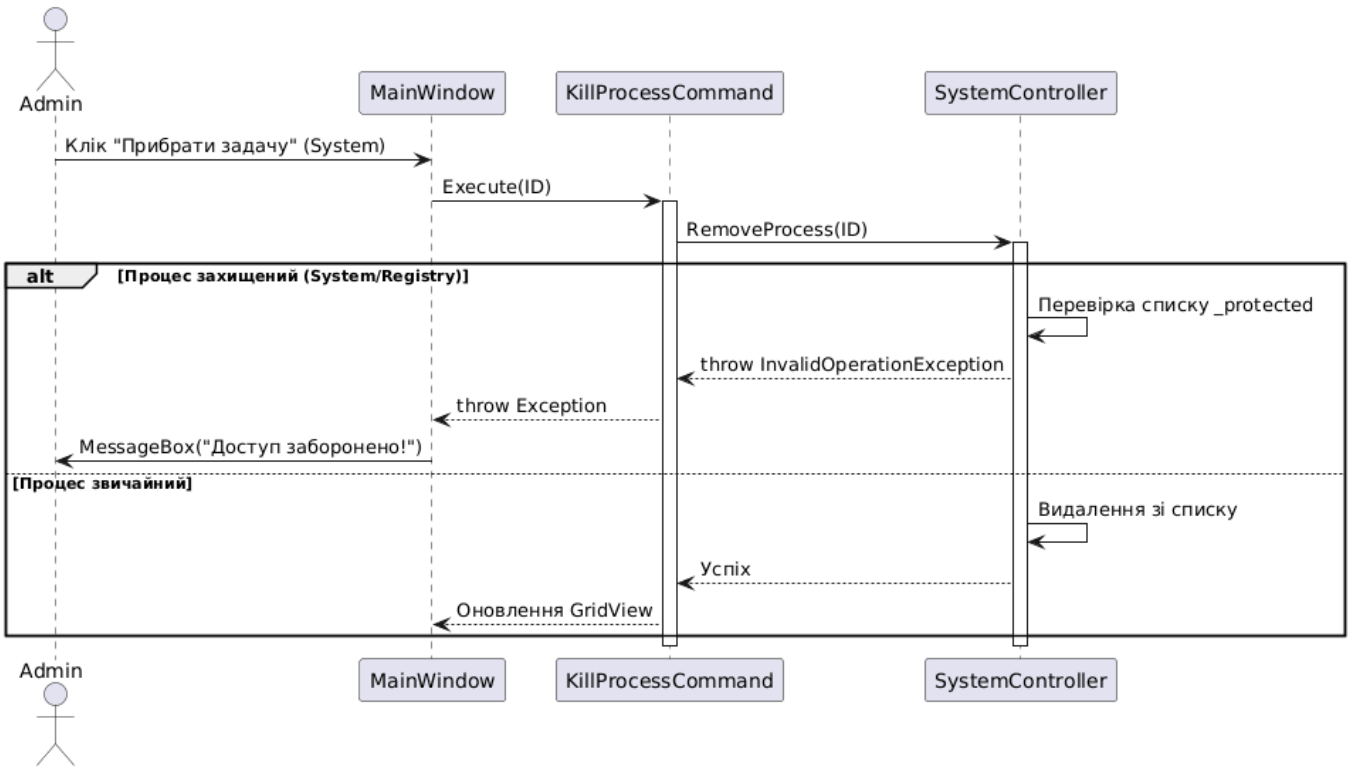

Рис. 1. 3 - Діаграма послідовності «Адміністративний контроль»

Таблиця 1.5. Сценарій використання «Аналітика ресурсів»

Поле	Опис
Назва	Генерація аналітичного звіту про ресурси
Передумови	Користувач або Адміністратор авторизований. Запущено кілька програм різних типів.
Постумови	Користувач отримує зведену статистику споживання пам'яті.
Взаємодіючі сторони	Користувач, Монітор, Відвідувач (Visitor).
Короткий опис	Система обходить усі активні процеси та підраховує сумарне споживання RAM/CPU, групуючи дані.
Основний потік подій	1. Користувач натискає кнопку "Аналіз ресурсів (Visitor)". 2. Система створює об'єкт SystemAnalysisVisitor. 3. Система ітерує список процесів, застосовуючи Visitor до кожного елемента. 4. Visitor накопичує дані про споживання ресурсів. 5. Система формує текстовий звіт і виводить його на екран.
Винятки	Список процесів порожній. Звіт міститиме нульові значення.
Примітки	Демонстрація патерну поведінки Visitor для відділення алгоритму аналізу від об'єктів процесів.

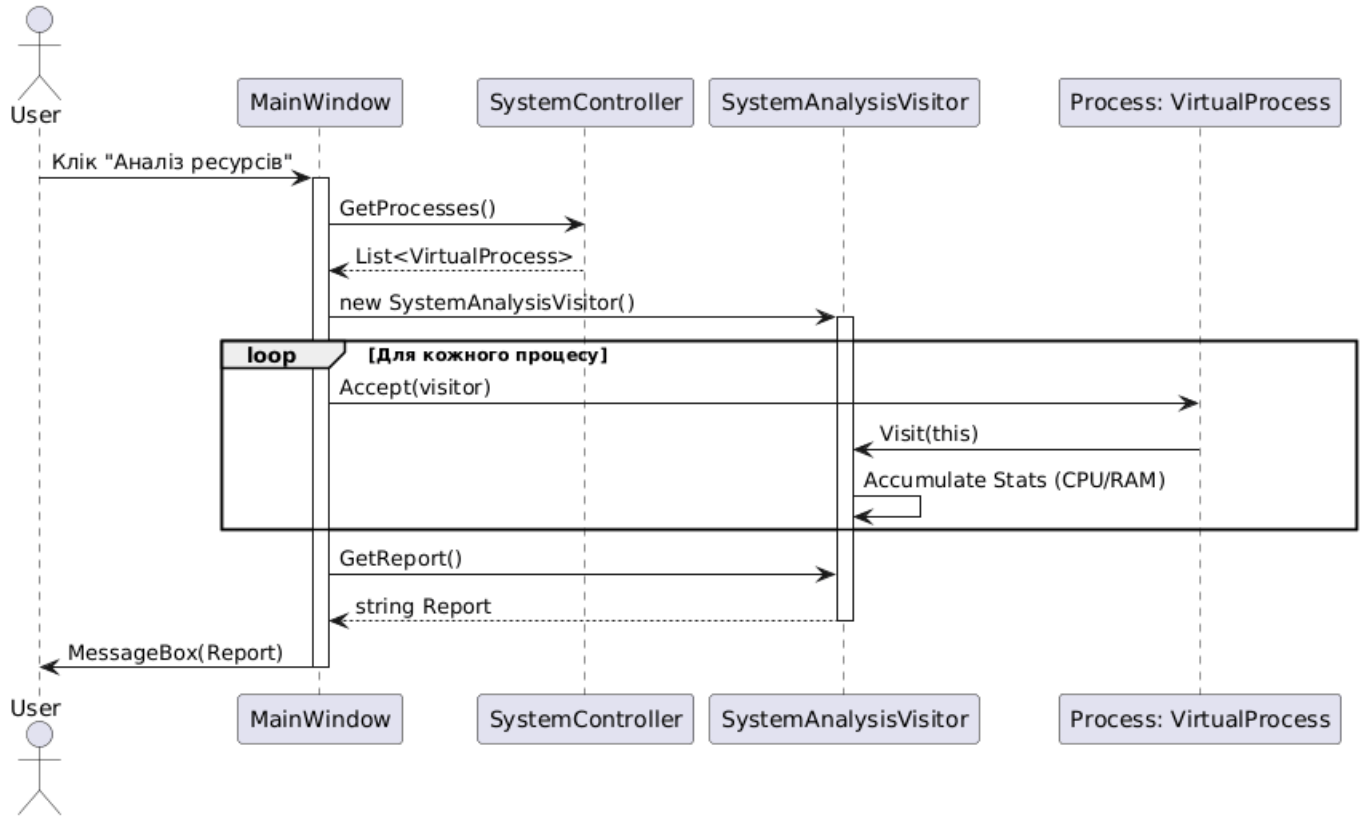

Рис. 1. 4 - Діаграма послідовності «Аналітика ресурсів»

1.5. Концептуальна модель системи

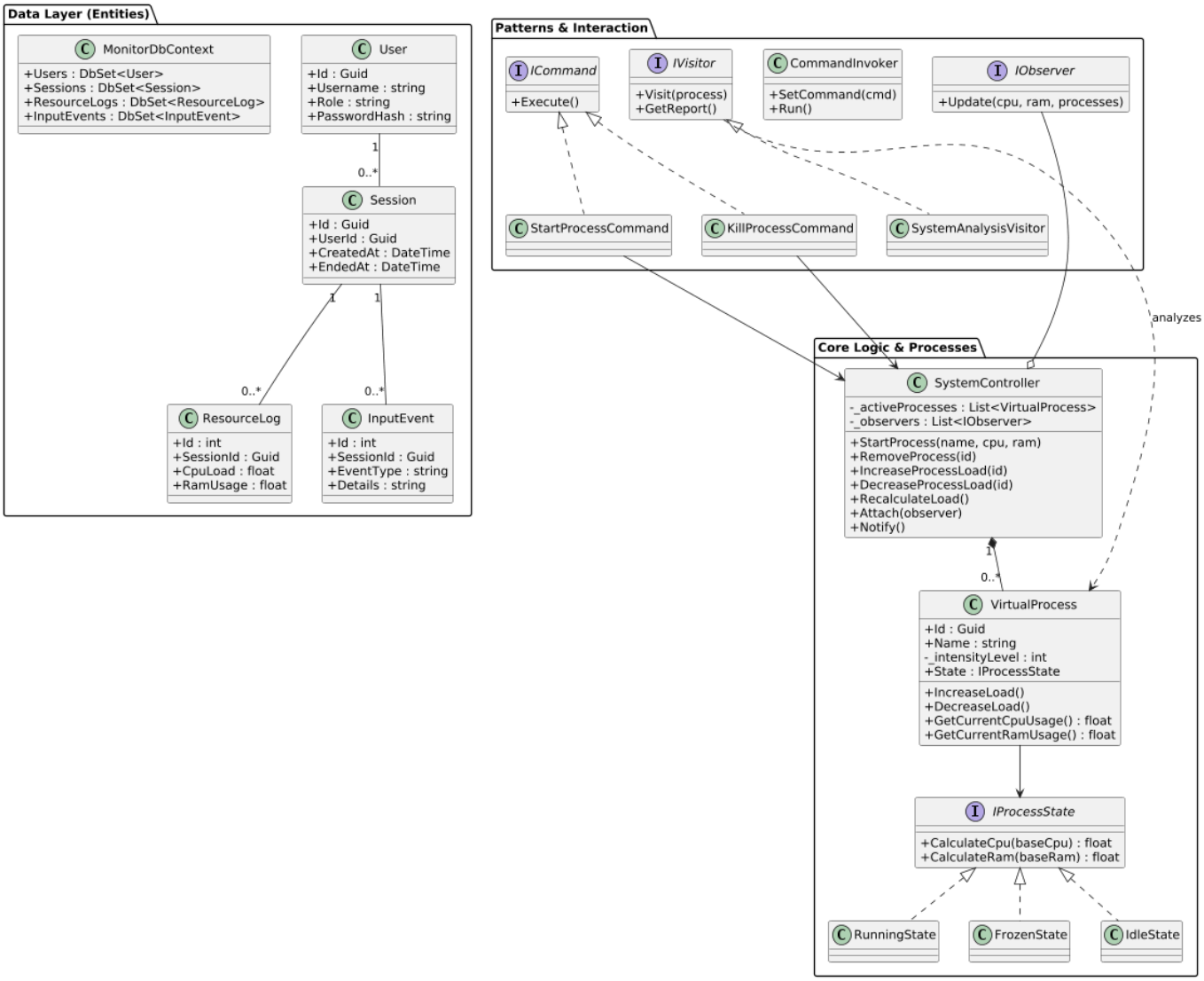

Рис. 1. 5 - Діаграма класів

Концептуальна модель програмної системи System Activity Monitor базується на об'єктно-орієнтованому підході та використанні архітектурних патернів для забезпечення гнучкості, масштабованості та слабкої зв'язності компонентів. Система складається з трьох логічних рівнів: рівень даних (Entities), рівень бізнес-логіки (Core Logic) та рівень представлення (UI), взаємодія між якими реалізована через патерни Command та Observer.

Як зображено на рисунку 1.6, архітектура системи центрується навколо класу SystemController, який виконує роль "ядра" емульованої операційної системи. Розглянемо ключові компоненти моделі детальніше.

1. Рівень даних (Data Access Layer) Цей рівень відповідає за збереження стану системи та історії активності. Він реалізований через ORM Entity Framework Core (клас MonitorDbContext) і включає наступні сутності:

- User: Зберігає облікові дані (логін, хеш пароля) та роль користувача (Admin/User).
- Session: Фіксує факти входу користувача в систему, час початку та завершення роботи, а також метадані про робочу станцію.
- ResourceLog: Зберігає зрізи (snapshots) стану системи (загальне навантаження CPU/RAM) з прив'язкою до сесії.
- InputEvent: Журнал подій вводу, що фіксує емульовані натискання клавіш та кліки миші.

2. Ядро системи (System Core) Центральним елементом є клас SystemController. Він керує списком активних процесів, розраховує навантаження та сповіщає інтерфейс про зміни.

- VirtualProcess: Моделює окрему запущену програму. Містить методи IncreaseLoad() та DecreaseLoad(), які дозволяють динамічно змінювати споживання ресурсів (наприклад, при відкритті нових вкладок у браузері).
- IProcessState (Pattern State): Визначає поведінку процесу в різних станах. Реалізації RunningState, FrozenState та IdleState по-різному розраховують використання CPU/RAM (наприклад, заморожений процес споживає 0% CPU).

3. Керування та взаємодія (Behavioral Patterns) Для взаємодії користувача з ядром використовуються патерни:

- Command (Pattern Command): Інтерфейс ICommand та його реалізації (StartProcessCommand, KillProcessCommand) інкапсулюють запити користувача як об'єкти. Це дозволяє безпечно передавати команди від UI до контролера через CommandInvoker.

- Observer (Pattern Observer): Інтерфейс IObserver дозволяє вікнам (MainWindow, DesktopWindow) автоматично отримувати оновлення від контролера без необхідності постійного опитування (polling).

4. Аналітика (Visitor Pattern) Для генерації звітів використовується патерн Visitor. Клас SystemAnalysisVisitor обходить список процесів і збирає статистику, не порушуючи інкапсуляцію класів процесів.

1.6. Вибір бази даних

Для забезпечення коректного функціонування системи моніторингу та збереження історії активності необхідно організувати надійне сховище даних. У межах проєкту база даних використовується для персистентного зберігання ключових сутностей, зокрема: облікових записів користувачів (Users), історії сесій входу (Sessions), логів споживання ресурсів (ResourceLogs) та журналу подій вводу (InputEvents).

На етапі проєктування архітектури було проведено порівняльний аналіз найпоширеніших реляційних СУБД, сумісних з платформою .NET: Microsoft SQL Server, PostgreSQL та SQLite.

Проте, враховуючи специфіку розроблюваної системи (System Activity Monitor), яка є локальним настільним додатком (Desktop Application, WPF), використання клієнт-серверної СУБД (як MS SQL Server) було визнано архітектурно надлишковим. Вимога до користувача встановлювати та налаштовувати локальний SQL-сервер лише для запуску утиліти моніторингу суттєво погіршила б досвід використання (User Experience).

Тому для фінальної реалізації було обрано вбудовану СУБД SQLite. Це рішення обумовлене такими факторами:

Архітектурна відповідність: SQLite ідеально підходить для локальних додатків, оскільки працює без окремого серверного процесу. Це стандартний підхід для зберігання налаштувань та історії в багатьох desktop-програмах (наприклад, історія браузерів Chrome чи Firefox теж зберігається в SQLite).

Портативність (Zero-configuration): Вся база даних зберігається в одному файлі (system_monitor.db), який автоматично створюється при першому запуску програми. Це дозволяє легко переносити проєкт між комп'ютерами (наприклад, для демонстрації викладачу) без необхідності розгортання серверного оточення.

Повна підтримка Entity Framework Core: Використання підходу Code-First дозволило описати структуру бази даних мовою C# та автоматично генерувати таблиці при старті програми, що спрощує розробку та модифікацію схеми даних.

Продуктивність: Швидкості запису SQLite цілком достатньо для фіксації подій моніторингу в реальному часі (з інтервалом в 1 секунду) без помітного впливу на швидкодію системи.

1.7. Вибір мови програмування та середовища розробки

Для програмної реалізації системи моніторингу та емуляції активності «System Activity Monitor» було обрано мову програмування C# та графічну підсистему Windows Presentation Foundation (WPF) на базі платформи .NET.

Мова програмування C# є потужним інструментом для створення настільних (desktop) додатків у середовищі Windows. Для даного проєкту C# було обрано завдяки наступним факторам:

- Об’єктно-орієнтована природа: C# забезпечує природну реалізацію складних архітектурних патернів, використаних у проєкті (Command, Observer, State, Visitor).
- Подійно-орієнтована модель: Вбудована підтримка подій (events) та делегатів дозволяє легко налаштувати взаємодію між компонентами системи (наприклад, реакція контролера на дії у вікні програми).
- Робота з пам'яттю: Механізм автоматичного керування пам'яттю (Garbage Collection) спрощує розробку, водночас дозволяючи емулювати витоки пам'яті та навантаження через створення великої кількості об'єктів.

Технологія WPF (Windows Presentation Foundation) використовується для побудови користувацького інтерфейсу. Її ключові переваги для даного проєкту:

- Мова розмітки XAML: Дозволяє декларативно описувати зовнішній вигляд вікон ("Робочий стіл", "Монітор"), відокремлюючи дизайн від програмної логіки (code-behind). Це значно спрощує підтримку та модифікацію інтерфейсу.
- Потужна система прив'язки даних (Data Binding): Забезпечує автоматичну синхронізацію між візуальними елементами (графіками, таблицями процесів) та джерелами даних у реальному часі без необхідності писати громіздкий код оновлення UI.
- Кастомізація та стилізація: WPF дозволяє створювати складні графічні інтерфейси, що виходять за межі стандартних елементів управління Windows, що було критично важливо для створення реалістичної імітації "браузера" з вкладками та сучасного вигляду панелі моніторингу.

Як середовище розробки було обрано Microsoft Visual Studio. Це повнофункціональне інтегроване середовище (IDE), яке є стандартом де-факто для .NET розробки. Вибір обґрунтований наступними можливостями:

- Інтегрований дизайнер XAML: Дозволяє візуально проєктувати інтерфейс та миттєво бачити зміни без запуску програми.
- Потужний відлагоджувач (Debugger): Критично важливий для даного проєкту, оскільки дозволяє відстежувати стан потоків, переглядати вміст об'єктів у пам'яті та діагностувати роботу патернів у покроковому режимі.
- Управління пакунками NuGet: Забезпечує легке підключення зовнішніх бібліотек, зокрема Entity Framework Core (для роботи з базою даних SQLite) та System.Drawing (для роботи з графікою).

1.8. Проєктування розгортання системи

Для повного розуміння архітектури системи необхідно розглянути її з двох точок зору: логічної та фізичної. Логічна архітектура визначає, як структуровано програмний код і як взаємодіють його модулі, тоді як фізична архітектура показує, як

скомпільований додаток розгортається на апаратному забезпеченні користувача.

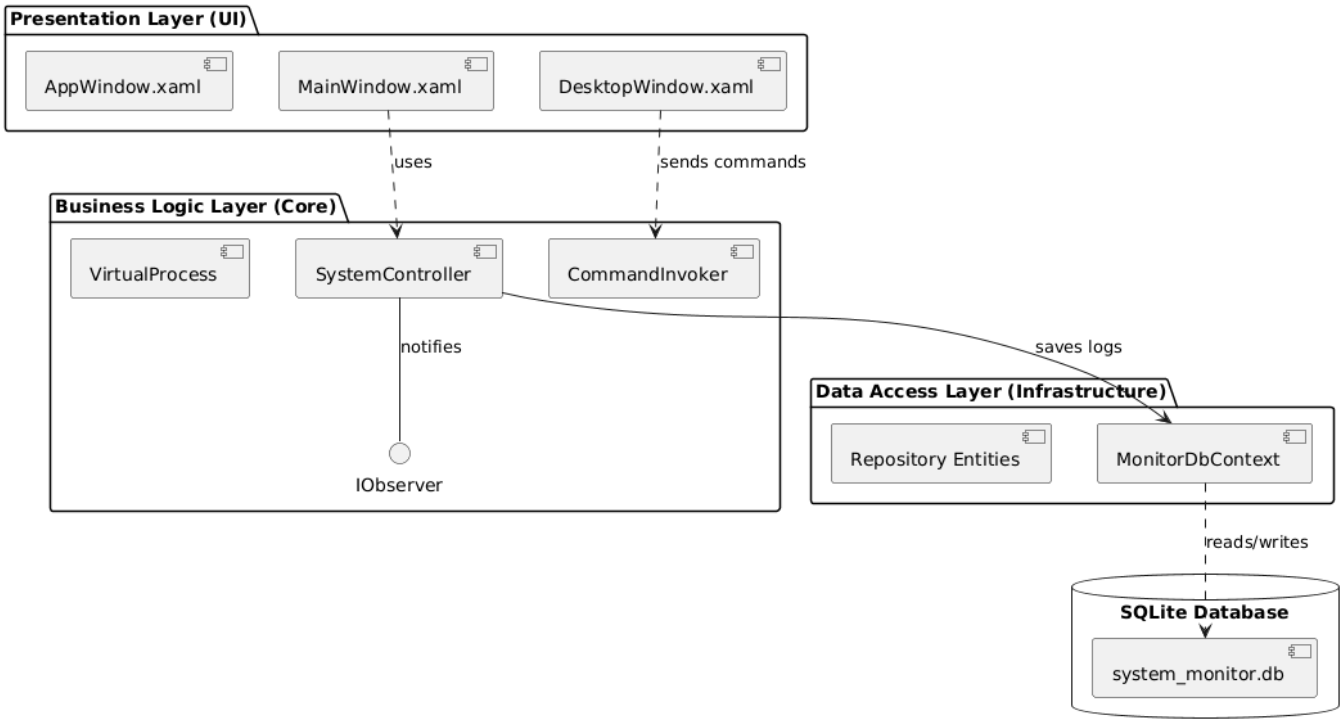

Рис. 1. 6 - Діаграма компонентів

На діаграмі компонентів (рис. 1.6) зображено логічну структуру системи System Activity Monitor, яка побудована за багатошаровим принципом (N-Tier Architecture) для забезпечення слабкої зв'язності та зручності тестування.

Система поділена на три основні логічні шари:

- Шар представлення (Presentation Layer): Відповідає за взаємодію з користувачем. Реалізований на технології WPF (XAML). Включає компоненти візуалізації: DesktopWindow (емуляція робочого столу), AppWindow (вікна програм) та MainWindow (панель моніторингу). Цей шар лише відображає дані та передає команди користувача до ядра системи.
- Шар бізнес-логіки (Core / Business Logic): Тут зосереджена основна функціональність емулятора. Головним компонентом є SystemController, який керує життєвим циклом процесів (VirtualProcess) та обробляє команди через CommandInvoker. Саме тут реалізовані алгоритми розрахунку навантаження CPU/RAM та патерни Observer/Visitor.

- Шар доступу до даних (Data Access Layer): Забезпечує абстракцію над базою даних. Використовує ORM Entity Framework Core (MonitorDbContext) для перетворення об'єктів C# у записи реляційної бази даних. Це дозволяє бізнес-логіці працювати з даними, не знаючи деталей SQL-запитів.

Компонент SQLite Database зображено окремо як файл system_monitor.db, з яким безпосередньо взаємодіє шар доступу до даних.

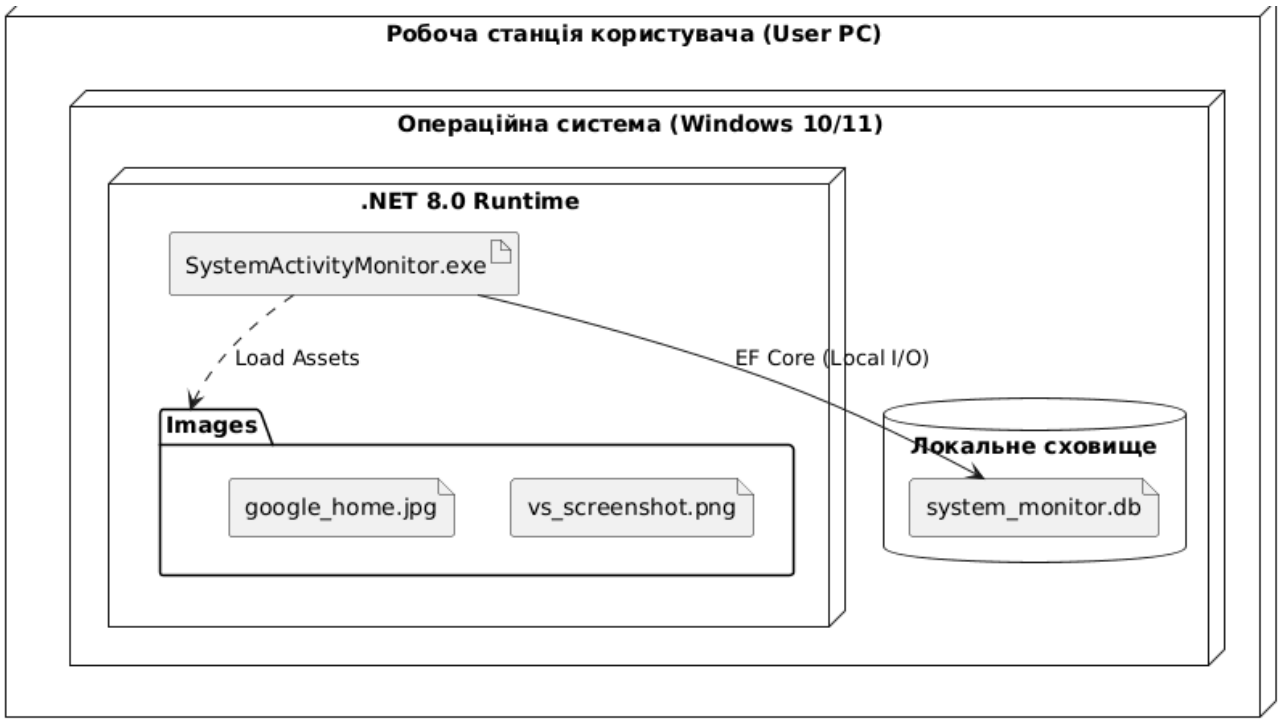

Рис. 1. 7 - Діаграма розгортання

На діаграмі розгортання (рис. 1.7) зображено фізичну архітектуру системи. Оскільки System Activity Monitor є локальним настільним додатком (Desktop Application), використовується модель автономного розгортання (Standalone Deployment).

Система розгортається на одному фізичному вузлі:

- Робоча станція користувача (User PC): Персональний комп'ютер під керуванням ОС Windows 10 або новішої версії. Він виступає одночасно і клієнтом, і сервером, і сховищем даних.

- Середовище виконання (.NET Runtime): Для роботи додатку необхідна наявність встановленої платформи .NET 8.0 (або новішої), яка забезпечує виконання керованого коду SystemActivityMonitor.exe, керування пам'яттю та обробку виключень.
- Файлова система (Local Storage): Використовується для зберігання файлу бази даних system_monitor.db (який створюється автоматично в папці з програмою) та папки Images з графічними ресурсами для емуляції інтерфейсів програм.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура бази даних

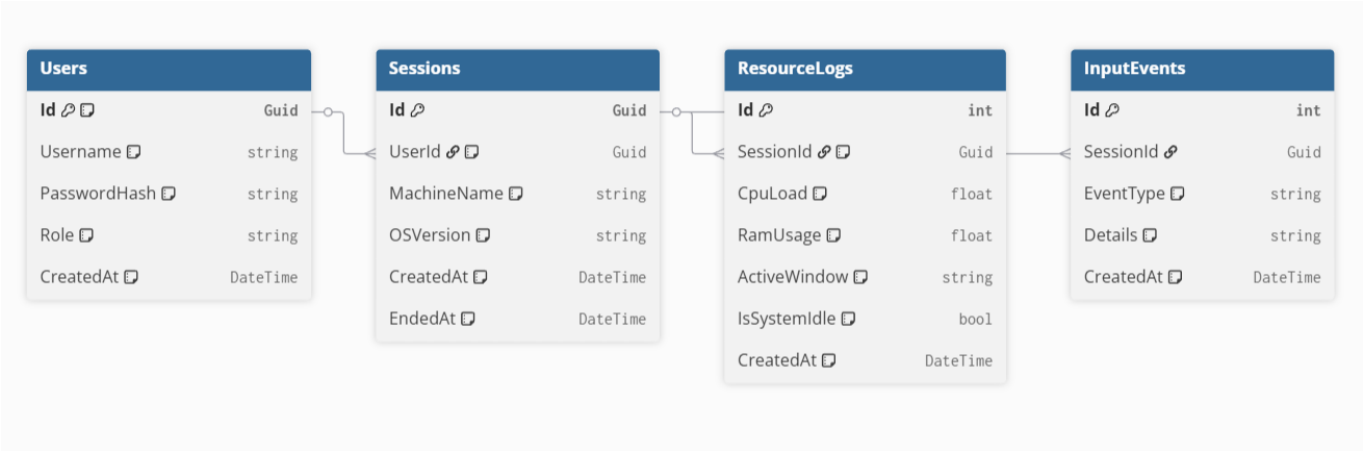

Рис. 2.1 – Проектування бази даних

Для забезпечення персистентності даних розроблено реляційну схему бази даних, що складається з чотирьох взаємопов’язаних сутностей. Структура таблиць та зв'язків між ними наведена на рис. 2.1.

1. User (Користувачі) Зберігає облікові дані всіх осіб, що мають доступ до системи моніторингу.

- Id (PK, Guid): Унікальний ідентифікатор користувача.
- Username (string): Логін для авторизації в системі.
- PasswordHash (string): Хеш пароля (забезпечує безпеку зберігання).
- Role (string): Рівень доступу користувача (Admin — повний доступ, User — обмежений доступ до власного робочого столу).

2. Session (Сесії активності) Фіксує факти входу користувачів у систему, дозволяючи відстежувати тривалість роботи та прив'язку до обладнання.

- Id (PK, Guid): Унікальний ідентифікатор сесії.
- UserId (FK, Guid): Посилання на користувача, який ініціював сесію.
- MachineName (string): Мережеве ім'я комп'ютера, на якому запущено клієнт.
- OSVersion (string): Версія операційної системи.
- CreatedAt (DateTime): Час початку сесії (авторизації).

- EndedAt (DateTime, Nullable): Час завершення сесії (виходу з системи). Якщо NULL — сесія ще активна.

3. ResourceLog (Логи ресурсів) Зберігає періодичні зрізи (snapshots) стану системи для побудови графіків та аналітики.

- Id (PK, int): Унікальний ідентифікатор запису логу.
- SessionId (FK, Guid): Посилання на активну сесію.
- CpuLoad (float): Поточне завантаження центрального процесора (у відсотках).
- RamUsage (float): Обсяг використаної оперативної пам'яті (у мегабайтах).
- ActiveWindow (string): Назва активного вікна або процесу в момент запису.
- IsSystemIdle (bool): Прапорець, що вказує на стан простою системи.
- CreatedAt (DateTime): Точний час запису показників.

4. InputEvent (Події вводу) Журнал емульованих дій користувача (натискання клавіш та миші) для аудиту активності.

- Id (PK, int): Унікальний ідентифікатор події.
- SessionId (FK, Guid): Посилання на сесію, в рамках якої відбулася подія.
- EventType (string): Тип події ("Mouse" або "Keyboard").
- Details (string): Деталі події (наприклад, "Left Click", "Ctrl+C", "Key Press").
- CreatedAt (DateTime): Час виникнення події.

2.2. Архітектура системи

2.2.1. Специфікація системи

Розроблена система "System Activity Monitor" є настільним (desktop) програмним комплексом, побудованим на базі платформи .NET (версія 9.0) з використанням технології Windows Presentation Foundation (WPF). Архітектура системи спроєктована за принципом подійно-орієнтованої архітектури (Event-Driven Architecture), що дозволяє реалізувати реактивну взаємодію між ядром емулятора та інтерфейсом користувача.

Таблиця 2.1. Технологічний стек

Платформа	.NET 9.
Мова програмування	C# 12.
Графічний інтерфейс	WPF (Windows Presentation Foundation) з використанням XAML
Доступ до даних (ORM)	Entity Framework Core.
База даних	SQLite (вбудована локальна база даних).
Системна взаємодія	System.Diagnostics.Process API (для керування процесами).
Клієнтська частина	Desktop UI (WPF Windows & UserControls)
Середовище розробки	Microsoft Visual Studio.

Структура рішення

Програмний код організовано у вигляді рішення (Solution), що реалізує модульну архітектуру з чітким відокремленням бізнес-логіки від інтерфейсу користувача. Рішення складається з двох основних проєктів:

1. SystemActivityMonitor.Data (Ядро системи та Бізнес-логіка): Це бібліотека класів (.NET Class Library), яка виконує роль Backend-у для настільного додатку. Вона об'єднує в собі функціонал, який у веб-додатках зазвичай розноситься на шари Domain, Data та Services:

- Domain Layer (Доменна область):
    - Визначає сутності (User, Session, ResourceLog), що формують модель даних системи.
    - Містить абстракції процесів (VirtualProcess) та їх станів (IProcessState, RunningState, FrozenState), реалізуючи патерн State.
- Data Access Layer (Шар даних):
    - Інкапсулює роботу з базою даних через MonitorDbContext.
    - Використовує Entity Framework Core для виконання міграцій та транзакцій (Code-First підхід).
- Service Layer (Сервісна логіка):

- Містить центральний контролер SystemController, який виступає сервісом керування ОС.
- Реалізує патерн Command (ICommand, CommandInvoker) для інкапсуляції запитів користувача.
- Реалізує патерн Visitor (SystemAnalysisVisitor) для бізнес-аналітики ресурсів.

2. SystemActivityMonitor.UI (Шар представлення / Presentation Layer): Це виконуваний проєкт (WPF Application), що відповідає виключно за Frontend системи:

- View (Представлення):
  - Набір вікон XAML (DesktopWindow, AppWindow, MainWindow), що формують візуальний інтерфейс.
- ViewModel & Interaction:
  - Реалізує логіку прив'язки даних (Data Binding) для відображення графіків та списків у реальному часі.
  - Реалізує патерн Observer (IObserver), підписуючись на події ядра для реактивного оновлення UI без необхідності прямого опитування контролера.
- Assets:
  - Зберігає графічні ресурси та стилі для емуляції інтерфейсів сторонніх програм.

2.2.2. Вибір та обґрунтування патернів реалізації

У процесі розробки системи System Activity Monitor було використано ряд поведінкових патернів проєктування для забезпечення гнучкості архітектури, слабкої зв'язності компонентів та розширюваності системи.

- Command (Команда)

Проблема: Необхідність відокремлення об'єктів, що ініціюють запити (кнопки інтерфейсу WPF), від об'єктів, що їх виконують (SystemController). Прямий виклик

методів контролера з обробників подій UI призводить до сильної зв'язності та ускладнює логування або скасування операцій.

Рішення: Всі операції керування процесами інкапсульовано в окремі об'єкти-команди, структура яких наведена на рис. 2.2.

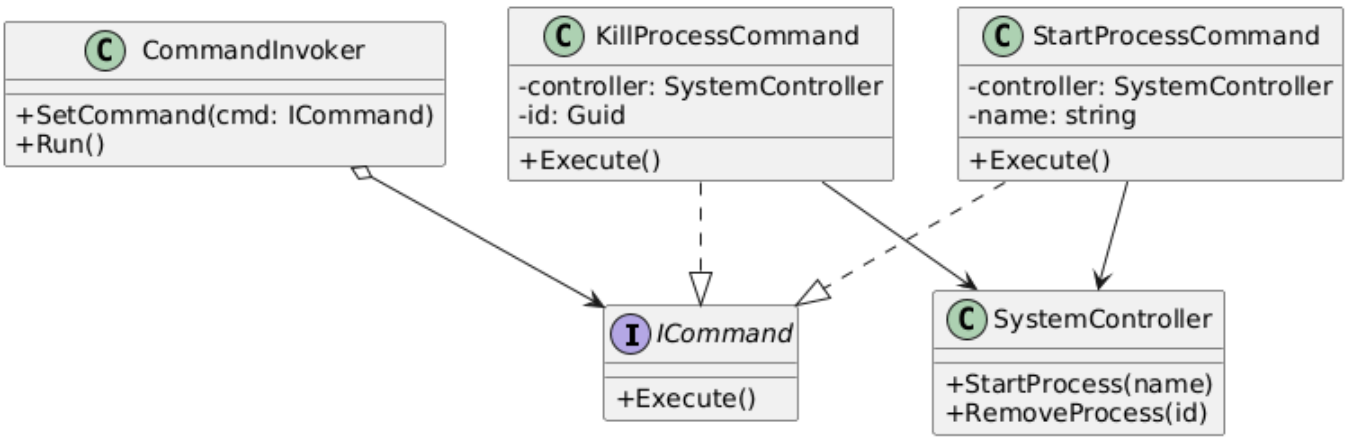

Рис. 2.2 - Структура патерну Command

ICommand — загальний інтерфейс, що декларує метод Execute(). StartProcessCommand / KillProcessCommand — конкретні реалізації команд. Вони зберігають параметри запиту (наприклад, ProcessId для видалення або назву програми для запуску) та посилання на отримувача. SystemController — виступає у ролі Отримувача (Receiver), який містить реальну бізнес-логіку виконання операцій. CommandInvoker — ініціатор (Invoker), який отримує команду від інтерфейсу та запускає її виконання. Такий підхід дозволяє додавати нові типи дій (наприклад, "PauseProcess") без зміни коду кнопок інтерфейсу.

- Observer (Спостерігач)

Проблема: Система працює в реальному часі, де параметри процесів (CPU, RAM) змінюються щосекунди. Інтерфейс користувача (MainWindow) повинен миттєво відображати ці зміни. Використання постійного опитування (polling) контролера таймером з боку UI є неефективним та ресурсомістким. Реалізовано патерн Observer для реактивного оновлення інтерфейсу при зміні стану моделі (рис. 2.3).

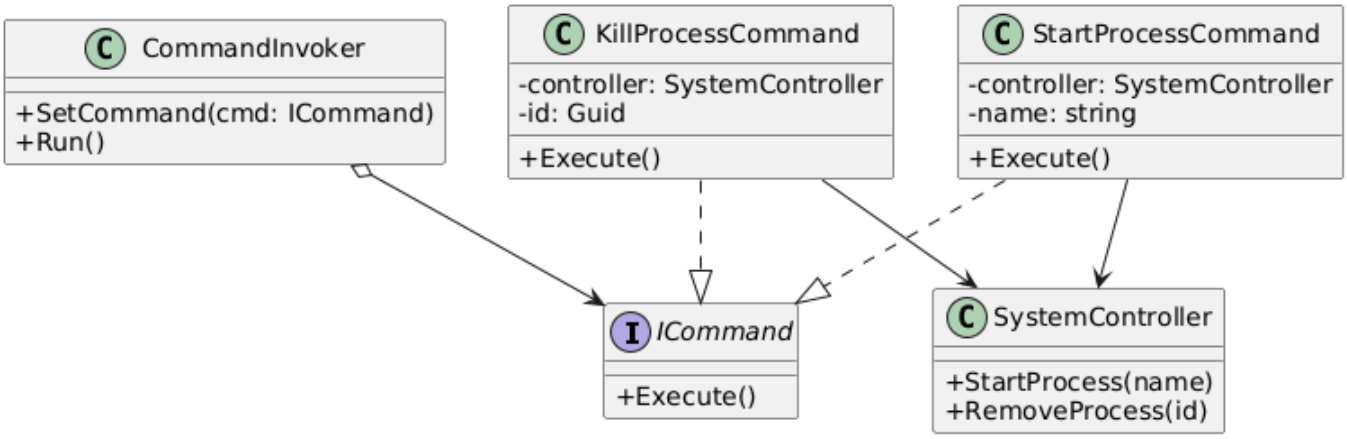

Рис. 2.3 - Структура патерну Observer

IObserver — інтерфейс підписника, що визначає метод Update(float cpu, float ram, List<Process> list). SystemController — виступає як Суб'єкт (Subject). Він веде список підписників та автоматично сповіщає їх викликаючи метод Notify() кожного разу, коли перераховує навантаження. MainWindow / DesktopWindow — конкретні спостерігачі, які реалізують інтерфейс IObserver. При отриманні сповіщення вони оновлюють графіки та таблиці даними, що прийшли від контролера. Перевага: Модуль бізнес-логіки не залежить від конкретних вікон GUI. Ми можемо додати нові вікна (наприклад, віджет для трею), просто підписавши їх на оновлення.

- State (Стан)

Емульовані процеси можуть перебувати в різних станах: "Запущений" (активно споживає ресурси), "Заморожений" (споживає RAM, але не CPU), "Очікування" (знижене споживання). Використання великих умовних конструкцій (if-else або switch) у класі процесу для розрахунку ресурсів ускладнює підтримку коду.
Рішення: Поведінку розрахунку ресурсів винесено в окремі класи станів (рис. 2.4).

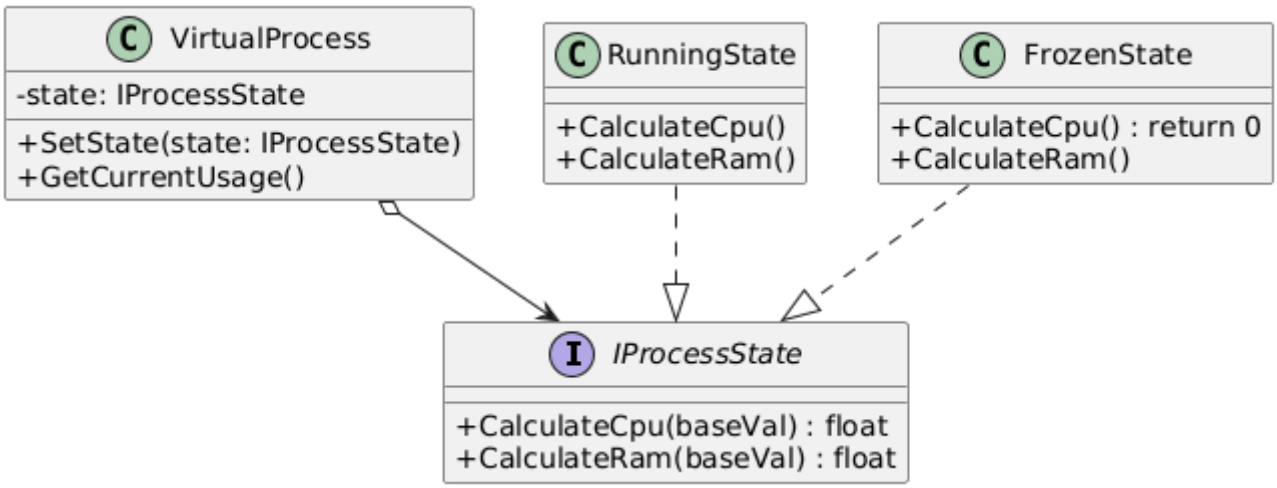

Рис. 2.4 - Структура патерну State

IProcessState — інтерфейс стану, що визначає методи CalculateCpu() та CalculateRam(). RunningState / FrozenState / IdleState — конкретні стани. Наприклад, FrozenState у методі розрахунку CPU завжди повертає 0, тоді як RunningState повертає значення з урахуванням навантаження. VirtualProcess — контекст, що зберігає посилання на поточний об'єкт стану. При запиті статистики він делегує обчислення активному стану. Локалізація логіки поведінки. Додавання нового стану (наприклад, "Zombie Process") не вимагає зміни класу самого процесу.

- Visitor (Відвідувач)

Необхідність отримання зведеного аналітичного звіту (наприклад, сумарне споживання пам'яті окремо браузерами та IDE), не змінюючи класи самих процесів. Додавання методів типу GenerateReport() у клас VirtualProcess порушило б принцип єдиної відповідальності (SRP). Використано патерн Visitor для відділення алгоритму аналізу від структури об'єктів (рис. 2.5).

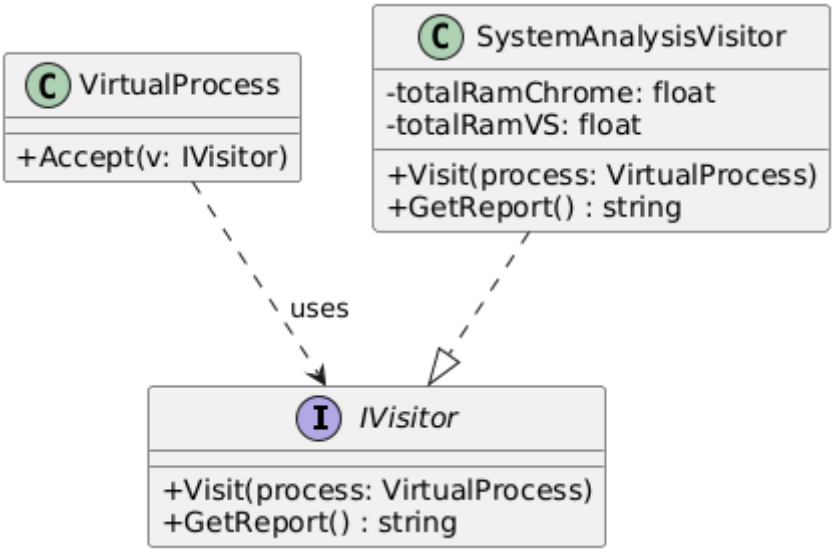

Рис. 2.5 - Структура патерну State

IVisitor — інтерфейс, що декларує метод Visit(VirtualProcess process). SystemAnalysisVisitor — конкретний відвідувач. Він проходить по списку процесів, накопичує статистику та групує дані за типами додатків. VirtualProcess — елемент структури, що має метод Accept(IVisitor visitor), який дозволяє відвідувачу "зайти" до об'єкта. Дозволяє створювати нові типи звітів (наприклад, експорт у XML або пошук витоків пам'яті), створюючи нові класи Visitor, без жодних змін у класах процесів.

2.3. Інструкція користувача

Початок роботи (для всіх користувачів)

1. Запуск програми: Після запуску виконуваного файлу SystemActivityMonitor.exe відкривається вікно авторизації ("Login Window").
2. Авторизація:
   - Введіть свій Логін та Пароль у відповідні поля.
   - Натисніть кнопку "Login" (Вхід).
   - Система автоматично перевіряє наявність користувача в базі даних system_monitor.db та звіряє хеш пароля.
   - У разі успішної авторизації створюється нова сесія (Session ID), і відкривається відповідний інтерфейс залежно від вашої ролі.

- У разі помилки система виведе повідомлення "Invalid credentials".

Інструкція Користувача (User Mode)

Основна мета звичайного користувача — імітація робочої діяльності для створення навантаження на систему.

1. Робочий стіл (Desktop Window): Після входу користувач потрапляє на емульований "Робочий стіл". Тут розташовані іконки доступних програм: Google Chrome та Visual Studio.
2. Запуск програм:
   - Двічі клікніть (або натисніть кнопку "Open") на іконці програми.
   - Система створить новий віртуальний процес, і на екрані з'явиться вікно відповідної програми.
3. Робота з Google Chrome (Емуляція RAM):
   - Відкрите вікно браузера відображає домашню сторінку.
   - Додавання вкладок: Натисніть кнопку "+" у верхній панелі вікна. Кожна нова вкладка ("New Tab") збільшує споживання оперативної пам'яті (RAM) процесом.
   - Закриття вкладок: Натисніть хрестик "x" на конкретній вкладці, щоб закрити її. Це звільнить частину ресурсів.
4. Робота з Visual Studio (Емуляція CPU):
   - Відкриття IDE імітує завантаження важкого проєкту, що призводить до підвищеного навантаження на центральний процесор (CPU).
5. Завершення роботи:
   - Закриття вікна програми (червоний хрестик) автоматично завершує процес і видаляє його зі списку активних задач.
   - Натисніть "Log Out" на робочому столі, щоб завершити сесію і повернутися до вікна входу.

Інструкція Адміністратора (Admin Mode)

Адміністратор має повний контроль над стабільністю системи та інструменти для аналізу ресурсів.

1. Головна панель моніторингу (Main Monitor): Після входу відкривається вікно диспетчера завдань. Основну частину екрана займає таблиця процесів, яка оновлюється в реальному часі (щосекунди).
    - Стовпці: ID процесу, Назва (Name), Статус (Running/Frozen), CPU (%), RAM (MB).
2. Керування процесами:
    - Завершення процесу (Kill): Якщо процес споживає занадто багато ресурсів, знайдіть його в списку та натисніть кнопку "Прибрати задачу" у колонці "Дії".
    - Захист системи: Спроба завершити критичні системні процеси (наприклад, System або Registry) буде заблокована, а на екран виведеться повідомлення про помилку доступу ("Access Denied").
3. Аналітика та Звіти (Visitor Pattern):
    - Для отримання зведеної статистики натисніть кнопку "Аналіз ресурсів (Visitor)" на панелі керування.
    - Система згенерує текстовий звіт, у якому буде підраховано сумарне споживання пам'яті за категоріями (наприклад: "Браузери: 1200 MB", "IDE: 800 MB", "Системні: 200 MB").
4. Історія та Аудит:
    - Всі дії користувачів (вхід, запуск програм, закриття вікон) автоматично фіксуються в базі даних. Адміністратор може переглядати логи попередніх сесій (за наявності відповідного інтерфейсу перегляду історії).

ВИСНОВКИ

У ході виконання курсової роботи було успішно спроєктовано та розроблено програмну систему «System Activity Monitor» — настільний (desktop) додаток для емуляції роботи операційної системи, моніторингу процесів та аналізу споживання системних ресурсів. На основі аналізу існуючих диспетчерів завдань було обрано архітектуру, що поєднує можливості реального моніторингу з навчальною симуляцією. Реалізація клієнтської частини на базі технології WPF (.NET 8) забезпечила створення сучасного, чутливого інтерфейсу користувача, а використання Entity Framework Core у зв’язці з SQLite дозволило організувати надійне локальне збереження історії сесій та логів активності без необхідності розгортання складного серверного оточення.

Ключовим досягненням роботи стала практична реалізація та інтеграція класичних патернів проєктування (GoF), що дозволило вирішити складні архітектурні задачі та забезпечити слабку зв'язність компонентів. Зокрема, патерн Command забезпечив інкапсуляцію дій користувача (запуск, завершення процесів), відокремивши графічний інтерфейс від бізнес-логіки ядра. Використання патерну Observer дозволило реалізувати реактивну модель оновлення даних, завдяки чому графіки та таблиці моніторингу відображають зміни в реальному часі без затримок. Патерн State надав гнучкий механізм для емуляції поведінки процесів у різних станах (активний, заморожений, очікування), коректно розраховуючи їх вплив на CPU та RAM. Для генерації зведених аналітичних звітів було застосовано патерн Visitor, що дозволило розширити функціонал аналізу без втручання в структуру класів процесів.

Розроблена система демонструє високий рівень інтерактивності та реалізму. Користувачі отримали повноцінну емуляцію «Робочого столу» з можливістю запуску віртуальних додатків (Google Chrome, Visual Studio). Унікальною особливістю стала реалізація динамічного навантаження: користувач може відкривати та закривати візуальні вкладки в браузері, спостерігаючи за миттєвою реакцією монітора ресурсів. Адміністратори системи забезпечені потужним інструментарієм для діагностики,

включаючи функції примусового завершення процесів («Kill») із вбудованим захистом від видалення критичних системних служб.

Особливу увагу в роботі було приділено стабільності та аудиту: реалізовано механізм збереження історії сесій, логування дій користувача (кліки, натискання клавіш) та захист від виключних ситуацій. Таким чином, мета курсової роботи досягнута в повному обсязі. Отриманий програмний продукт є архітектурно довершеною системою, яка може слугувати як ефективним навчальним тренажером для вивчення принципів роботи ОС, так і базою для подальшого розширення функціоналу.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Desktop Guide (WPF .NET). Documentation for Windows Presentation Foundation [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/.
2) C# documentation. Official documentation for C# language [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/.
3) Visual Studio 2022 Documentation. IDE Documentation [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/visualstudio/ide/.
4) Process Class (System.Diagnostics). .NET API Browser [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process.
5) Entity Framework Core Documentation. Documentation for Entity Framework Core [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/ef/core/.
6) SQLite Documentation. Official Documentation for SQLite [Електронний ресурс]. – Режим доступу: https://www.sqlite.org/docs.html.
7) Troelsen A., Japikse P. Pro C# 10 with .NET 6. – Apress, 2022. – 1650 p. [Електронний ресурс]. – Режим доступу: https://link.springer.com/book/10.1007/978-1-4842-7869-7.
8) Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley Professional, 1994. – 395 p. [Електронний ресурс]. – Режим доступу: https://www.javier8a.com/itc/bd1/articulo.pdf.
9) Refactoring.Guru. Design Patterns: Command, Observer, Visitor, State [Електронний ресурс]. – Режим доступу: https://refactoring.guru/uk/design-patterns.

10) Richter J. CLR via C#. – 4th Edition. – Microsoft Press, 2012. – 896 p. [Електронний ресурс]. – Режим доступу: https://www.microsoft.com/learning.

11) Albahari J. C# 12 in a Nutshell: The Definitive Reference. – O'Reilly Media, 2023. – 1060 p.

ДОДАТКИ

Додаток А

Проект завантажено на github-репозиторій, доступний за посиланням: https://github.com/Balance4490/SystemActivityMonitorPrj/commits/master/

Додаток Б

Проектування паттернів

- **Програмна реалізація патерну Command**

Інтерфейс ICommand.cs:

```
namespace SystemActivityMonitor.Data.Patterns.Command
{
    public interface ICommand
    {
        void Execute();
    }
}
```

Конкретна команда завершення процесу із захистом: KillProcessCommand.cs

```
namespace SystemActivityMonitor.Data.Patterns.Command
{
    public class KillProcessCommand : ICommand
    {
        private SystemController _controller;
        private Guid _processId;

        public KillProcessCommand(SystemController controller, Guid processId)
        {
            _controller = controller;
            _processId = processId;
        }

        public void Execute()
        {
            _controller.KillProcess(_processId);
        }
    }
}
```

Виклик команди з UI: MainWindow.xaml.cs:

```
private void BtnKillRow_Click(object sender, RoutedEventArgs e)
```

```
        {
            var button = (System.Windows.Controls.Button)sender;
            if (button.Tag is Guid processId)
            {
                var cmd = new KillProcessCommand(_controller, processId);
                _invoker.SetCommand(cmd);

                try
                {
                    _invoker.Run();
                }
                catch (InvalidOperationException ex)
                {
                    MessageBox.Show(ex.Message, "Системна помилка",
MessageBoxButton.OK, MessageBoxImage.Error);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Не вдалося завершити процес: " + ex.Message);
                }
            }
        }
```

- **Програмна реалізація патерну Observer (Спостерігач)**

Інтерфейс IObserver.cs:

```
namespace SystemActivityMonitor.Data.Patterns.Observer
{
    public interface IObserver
    {
        void Update(float totalCpu, float totalRam, List<VirtualProcess> processes);
    }
}
```

Суб'єкт спостереження: KillProcessCommand.cs

```
public class SystemController
{
    private List<IObserver> _observers = new List<IObserver>();

    public void Attach(IObserver observer)
    {
        _observers.Add(observer);
    }

    public void Notify()
    {
        float totalCpu = _activeProcesses.Sum(p => p.GetCurrentCpuUsage());
        float totalRam = _activeProcesses.Sum(p => p.GetCurrentRamUsage());
```

```
        foreach (var observer in _observers)
        {
            observer.Update(totalCpu, totalRam, new
List<VirtualProcess>(_activeProcesses));
        }
    }
}
```

Реалізація спостерігача у вікні: MainWindow.xaml.cs:

```
public partial class MainWindow : Window, IObserver
{
    public void Update(float totalCpu, float totalRam, List<VirtualProcess>
processes)
    {
        Dispatcher.Invoke(() =>
        {
            ProcessList.ItemsSource = processes;
            CpuBar.Value = totalCpu;
            RamBar.Value = totalRam;
            lblCpu.Text = $"{totalCpu:F1}%";
            lblRam.Text = $"{totalRam:F0} MB";
        });
    }
}
```

- **Програмна реалізація патерну State (Стан)**

Інтерфейс IProcessState.cs:

```
public interface IProcessState
{
    float CalculateCpu(float baseCpu);
    float CalculateRam(float baseRam);
    string Name { get; }
}
```

Стан "Заморожений": FrozenState.cs

```
public class FrozenState : IProcessState
{
    public string Name => "Frozen";

    public float CalculateCpu(float baseCpu)
    {
        return 0f;
    }
```

```
    public float CalculateRam(float baseRam)
    {
        return baseRam;
    }
}
```

Контекст – Процес: VirtualProcess.cs:

```
public class VirtualProcess
{
    public IProcessState State { get; set; }

    public float GetCurrentCpuUsage()
    {
        float multiplier = 1.0f + (_intensityLevel - 1) * 0.1f;
        return State.CalculateCpu(_baseCpu) * multiplier;
    }

    public void Freeze()
    {
        State = new FrozenState();
    }
}
```

- **Програмна реалізація патерну Visitor (Відвідувач)**

Інтерфейс відвідувача IVisitor.cs:

```
public interface IVisitor
{
    void Visit(VirtualProcess process);
}
```

Метод Accept у процесі: VirtualProcess.cs

```
public class VirtualProcess
{
    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

Аналітик ресурсів: SystemAnalysisVisitor.cs

```
public class SystemAnalysisVisitor : IVisitor
{
    private float _chromeRam = 0;
    private float _vsRam = 0;
    private float _systemRam = 0;
```

```
    public void Visit(VirtualProcess process)
    {
        if (process.Name.Contains("Chrome"))
        {
            _chromeRam += process.GetCurrentRamUsage();
        }
        else if (process.Name.Contains("Visual Studio"))
        {
            _vsRam += process.GetCurrentRamUsage();
        }
        else
        {
            _systemRam += process.GetCurrentRamUsage();
        }
    }

    public string GetReport()
    {
        return $"Звіт по пам'яті:\nChrome: {_chromeRam} MB\nVisual Studio: {_vsRam}
MB\nSystem: {_systemRam} MB";
    }
}
```