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-з31 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Курбатова Кіріла Андрійовича

(прізвище, ім’я, по батькові)

1.Тема роботи JSON Tool

2. Строк здачі студентом закінченої роботи_____

3. Вихідні дані до роботи:

_____

_____

_____

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

Застосунок повинен мати можливість завантажувати та зберігати JSON Schema файли, виконувати багаторівневу валідацію (синтаксис JSON, структура схеми, відповідність даних схемі), надавати візуальний редактор властивостей схеми з підтримкою типів, форматів, описів та прикладів, забезпечувати систему відміни/повтору операцій (Undo/Redo) з історією до 50 команд, реалізовувати автоматичне збереження з механізмом debounce, експортувати документацію схеми у формат Markdown, відображати схему у плоскому вигляді (Flat View).

Додатки:

https://github.com/Kurbatishche/JsonTool

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

Visual Studio Code, app.diagrams.net

6. Дата видачі завдання 10.09.2025

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Отримання теми курсової роботи	10.09.2025	
2.	Загальний опис	23.09.2025	
3.	Огляд існуючих рішень	26.09.2025	
4.	Визначення вимог до системи	28.09.2025	
5.	Визначення сценаріїв використання	05.10.2025	
6.	Проєктування архітектури MVVM та вибір технологій	15.10.2025	
7.	Реалізація базової структури проєкту та інтерфейсу	22.20.2025	
8.	Реалізація Strategy Pattern для валідації	29.10.2025	
9.	Реалізація Command Pattern для Undo/Redo	05.11.2025	
10.	Реалізація Observer Pattern для авто-збереження	12.11.2025	
11.	Реалізація Template Method та Flyweight Pattern	19.11.2025	
12.	Оформлення курсової роботи	19.11.2025	
13.	Захист курсової роботи		
14.			
15.			
16.			
17.			
18.			

Студент _____ _____Кіріл_КУРБАТОВ_____
(підпис) (Ім’я ПРІЗВИЩЕ)

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

«____»_____ 2025 р.

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

Тема JSON Tool

Курсова робота
З дисципліни «Технології розроблення програмного забезпечення»

Керівник
доц. Амонс О.А.
«Допущений до захисту»
_____
(Особистий підпис керівника)
« » _____2025р.
Захищений з оцінкою
_____
(оцінка)
Члени комісії:
_____ _____
(особистий підпис) (розшифровка підпису)
_____ _____
(особистий підпис) (розшифровка підпису)

Виконавець
ст. Курбатов К. А.
залікова книжка № ___ – _____
гр. ІА-з31
_____
(особистий підпис виконавця)
« »_____2025р.

Київ – 2025

ЗМІСТ

ВСТУП

У сучасному світі програмної інженерії формат JSON (JavaScript Object Notation) став де-факто стандартом для обміну даними між різними системами та сервісами. За даними дослідження State of API 2023, понад 90% сучасних REST API використовують JSON як основний формат передачі даних. Такі технологічні гіганти як Google, Amazon, Microsoft, Facebook активно використовують JSON у своїх публічних API та внутрішніх системах.

Водночас зростання складності API та необхідність забезпечення якості даних призвели до широкого впровадження JSON Schema — стандарту для опису структури JSON-документів. JSON Schema дозволяє валідувати дані, генерувати документацію та автоматизувати тестування. За статистикою npm, бібліотеки для роботи з JSON Schema завантажуються понад 50 мільйонів разів на тиждень.

Зростання популярності мікросервісної архітектури значно збільшило потребу в якісних інструментах для роботи з JSON Schema. Кожен мікросервіс має свої контракти даних, які потрібно описувати, валідувати та підтримувати. Помилки у схемах даних можуть призводити до критичних збоїв у продакшн-системах, що робить якісний інструментарій для роботи зі схемами особливо важливим.

Існуючі рішення для редагування JSON Schema мають суттєві обмеження. Вбудовані редактори в IDE, такі як VS Code, надають базову підтримку JSON, але не мають спеціалізованого інтерфейсу для роботи з метаданими властивостей. Онлайн-інструменти на кшталт JSONLint або JSON Editor Online вимагають постійного підключення до інтернету та мають обмежені можливості валідації. Комерційні рішення, такі як Altova XMLSpy, мають високу вартість та надлишкову функціональність для типових задач роботи зі схемами.

Особливу цінність для розробки якісного програмного забезпечення мають патерни проєктування — перевірені рішення типових архітектурних проблем. Класифікація патернів, запропонована «бандою чотирьох» (Gang of Four), залишається актуальною понад 30 років. Застосування патернів дозволяє створювати

гнучкий, розширюваний та легкий у підтримці код, що особливо важливо для довгострокових проєктів.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

Visual Studio Code є безкоштовним редактором коду з великою екосистемою розширень, підтримкою IntelliSense, кросплатформністю та інтеграцією з Git, проте має суттєві недоліки: відсутність спеціалізованого інтерфейсу для редагування метаданих, необхідність налаштувань для роботи з JSON Schema, відсутність вбудованого експорту схеми, візуального редактора структури та можливостей порівняння версій.

Postman, представлений як інструмент для тестування API, має зручний інтерфейс, функцію автоматичної генерації схем з прикладів, підтримку колекцій та командної роботи, але орієнтований переважно на тестування API, має обмежену валідацію структури схеми, комерційну ліцензію для розширених функцій, вимагає реєстрації та не пропонує офлайн-режим у безкоштовній версії.

JSON Schema Validator Online — це онлайн-сервіс, який відрізняється простотою використання без встановлення, підтримкою різних версій JSON Schema, миттєвим результатом валідації, можливістю ділитися посиланням та безкоштовністю, однак має суттєві обмеження: залежність від інтернету, відсутність можливості редагування схеми та підсвічування синтаксису, неможливість збереження історії та обмежений розмір вхідних даних.

Oxygen XML Editor — професійний редактор, що пропонує високий рівень підтримки JSON Schema, візуальний редактор структури, генерацію документації, рефакторинг схем та інтеграцію з системами контролю версій, але має високу вартість ліцензії, надлишковий функціонал для простих задач, значні системні вимоги, складний інтерфейс для новачків та зосереджений переважно на XML.

Altova XMLSpy — потужний комерційний редактор, що включає розширений візуальний редактор схем, генерацію коду, інтеграцію з базами даних, підтримку великих схем та розширену діагностику, проте відрізняється дуже високою вартістю,

ресурсомісткістю, складністю освоєння, надлишковістю для типових задач та доступний лише для операційної системи Windows.

JetBrains IDE Plugins представлені плагінами для середовищ розробки, які забезпечують тісну інтеграцію з IDE, якісне автодоповнення, навігацію по структурі схеми, підтримку посилань та безкоштовність у Community Edition, але вимагають встановлення важкого середовища, не є окремим інструментом для схем, мають обмежений експорт, не містять візуального редактора та пропонують повний функціонал лише в платній версії.

1.2. Загальний опис проєкту

Концепція застосунку JSON Schema Tool полягає у наданні розробникам зручного, швидкого та надійного інструменту для створення, редагування та валідації JSON Schema. Це спеціалізований десктопний редактор для операційної системи Windows, розроблений на платформі .NET 8 з використанням технології WPF (Windows Presentation Foundation). На відміну від універсальних редакторів коду, він зосереджений виключно на роботі зі схемами JSON, що дозволяє оптимізувати інтерфейс та функціональність під конкретні задачі. Застосунок поєднує текстовий редактор з підсвічуванням синтаксису та структурований вигляд властивостей схеми, що забезпечує зручність як для досвідчених розробників, так і для новачків. Цільова аудиторія включає backend-розробників, що створюють REST API та потребують валідації контрактів даних, frontend-розробників, DevOps-інженерів, Technical Writers та QA-інженерів. Основні сценарії використання охоплюють створення нових схем з валідацією в реальному часі, документування існуючих схем з подальшим експортом у Markdown, валідацію JSON-даних з детальними повідомленнями про помилки, а також аналіз складних вкладених схем через режим плоского відображення (Flat View).

Функціональні можливості застосунку включають потужну систему валідації, яка підтримує три рівні перевірки: синтаксичну валідацію JSON, валідацію структури схеми на відповідність стандарту та валідацію даних за схемою. Для перевірки JSON-

документів призначена функція “Check Data”, результати якої з можливістю переходу до рядка з помилкою відображаються в спеціальній панелі. Редагування метаданих властивості, таких як тип, формат, опис та приклади, здійснюється через зручну панель “Edit Property”, а зміни застосовуються з використанням Command Pattern, що забезпечує надійний механізм Undo/Redo для відміни та повторення до 50 операцій. Для захисту даних користувача реалізована система авто-збереження, яка працює за принципом debounce, виконуючи запис через 3 секунди після останньої зміни, щоб уникнути надмірного навантаження. Додатково застосунок дозволяє генерувати документацію через експорт схеми у вигляді структурованої Markdown-таблиці та аналізувати складні структури за допомогою режиму Flat View, який перетворює вкладену схему на зручний список повних шляхів до кожної властивості.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

Для функціональних вимог застосунку пріоритетними є наступні можливості високого рівня: завантаження JSON-файлів через діалог відкриття (FR-01), відображення JSON з кольоровим підсвічуванням синтаксису (FR-02) та перевірка його синтаксису в реальному часі (FR-03). Критично важливими також є валідація структури самої JSON Schema (FR-04), можливість перевірки JSON-даних на відповідність схемі (FR-05) та наочне відображення списку властивостей схеми (FR-06). Система повинна дозволяти редагувати метаданти властивостей, такі як тип, формат та опис (FR-07), та надавати надійні інструменти контролю через функції відміни (Undo) (FR-08) та повторення (Redo) операцій (FR-09). Обов'язковою є можливість збереження змін у файл (FR-16), а система повинна наочно показувати помилки валідації з вказівкою номера рядка (FR-14).

До вимог середнього пріоритету належать: підтримка історії до 50 операцій для механізму Undo/Redo (FR-10), система авто-збереження, що активується через 3 секунди після останніх змін (FR-11), та функція експорту схеми у Markdown-таблицю

для документації (FR-12). Користувацький досвід також покращують режим плоского відображення схеми (Flat View) для аналізу складних структур (FR-13), підтримка основних гарячих клавіш для збереження, відкриття та управління історією редагування (FR-15), можливість збереження файлу під новим іменем (Save As) (FR-17), а також відображення нумерації рядків у редакторі (FR-19).

Вимоги низького пріоритету включають відкриття файлів через механізм Drag & Drop (FR-18) та підтримку згортання блоків JSON-коду в редакторі для кращої навігації (FR-20).

1.3.2. Нефункціональні вимоги до системи

Нефункціональні вимоги до системи охоплюють шість ключових аспектів: продуктивність, надійність, зручність використання, підтримуваність, портативність та безпеку.

Вимоги до продуктивності визначають, що час завантаження файлів розміром до 1 МБ не повинен перевищувати 2 секунди на типовому обладнанні (наприклад, Intel Core i5, 8 ГБ RAM). Інтерфейс користувача зобов'язаний забезпечувати час відгуку на дії не більше 100 мілісекунд, а всі довготривалі операції, такі як валідація, збереження та експорт, мають виконуватися асинхронно, щоб не блокувати основний потік. Ефективне використання ресурсів гарантується обмеженням споживання оперативної пам'яті на рівні не більше 500 МБ при роботі з файлами до 10 МБ.

Вимоги надійності включають стійку обробку помилок: система має коректно обробляти некоректні JSON-файли без аварійного завершення роботи. Механізм авто-збереження повинен гарантувати відновлення даних у разі непередбачуваного закриття програми, а усі операції редагування мають бути повністю оборотними за допомогою функції Undo. Для діагностики проблем система вестиме логування критичних помилок.

Аспект зручності використання зосереджений на інтуїтивному інтерфейсі, який має бути зрозумілим користувачам з досвідом роботи з JSON без необхідності читання документації. Основні операції повинні бути доступними одночасно через головне меню, панель інструментів та гарячі клавіші. Система надаватиме чіткий зворотний зв'язок через зрозумілі повідомлення про результати дій та детальну інформацію про помилки, підтримуючи високий рівень консистентності в оформленні елементів управління та термінології.

Вимоги підтримуваності спрямовані на забезпечення якості кодової бази. Код має бути організований у логічні модулі з чіткими залежностями, а архітектура — ґрунтуватися на відомих патернах проєктування. Критичні компоненти системи мають бути покриті модульними тестами (unit-тестами), а публічні API класів — супроводжуватися XML-документацією для полегшення розвитку та супроводу.

Вимоги портативності визначають середовище виконання: застосунок повинен стабільно працювати на операційних системах Windows 10 та Windows 11 (64-бітні версії), використовуючи середовище виконання .NET 8.0. Бажаним є підтримка портативного (portable) режиму роботи, що дозволяє користуватися програмою без її формального встановлення.

Нарешті, вимоги безпеки гарантують, що всі дані користувача оброблятимуться виключно локально, без передачі на зовнішні сервери. Застосунок має функціонувати в стандартному режимі, не вимагаючи для своєї роботи адміністративних прав доступу.

1.4. Сценарії використання системи

Головним актором системи є Користувач — розробник програмного забезпечення, який працює з JSON Schema для створення документації API, валідації даних або автоматичної генерації коду. Він має базові знання JSON та розуміє концепцію JSON Schema.

Система підтримує ряд детальних сценаріїв використання. Сценарій UC-01 «Завантаження JSON Schema» описує процес відкриття файлу через меню File → Open або гарячу клавішу Ctrl+O, завантаження його вмісту з валідацією та відображення в редакторі з підсвічуванням синтаксису. Сценарій передбачає альтернативні шляхи при отриманні некоректного JSON або якщо файл не знайдено.

Сценарій UC-02 «Створення нової схеми» ініціюється командою File → New, після чого система, перевіривши наявність незбережених змін, очищує редактор і вставляє шаблон базової JSON Schema. UC-03 «Редагування властивості схеми» дозволяє користувачеві вибрати властивість зі списку, змінити її атрибути через панель Edit Property та застосувати зміни, які автоматично оновлять JSON-документ і будуть записані в історію операцій.

Валідація системи охоплює два ключових сценарії. UC-04 «Валідація структури схеми» запускає асинхронну перевірку самої схеми на відповідність стандарту, а UC-05 «Перевірка JSON документа за схемою» дозволяє завантажити зовнішній JSON-файл і перевірити його на коректність щодо поточної схеми. Обидва сценарії наочно відображають результати, включаючи список помилок.

Для підтримки контролю над діями користувача реалізовано сценарій UC-06 «Відміна операції (Undo)», який дозволяє послідовно скасовувати зміни через команду CommandManager. Сценарій UC-07 «Авто-збереження» описує роботу фонового механізму, який автоматично зберігає файл через 3 секунди після останньої зміни, використовуючи паттерн Спостерігач та debounce-таймер для оптимізації.

Система також надає додаткові інструменти для роботи зі схемами. UC-08 «Експорт у Markdown» генерує з поточної схеми структуровану таблицю у форматі Markdown для документації. UC-09 «Перегляд Flat View» перетворює вкладену

структуру JSON Schema на плоский список шляхів для кращого аналізу. Коли валідація виявляє помилки, сценарій UC-10 «Робота з помилками валідації» надає зручну навігацію, дозволяючи користувачеві клацати по помилці в списку для миттєвого переходу до відповідного рядка в редакторі.

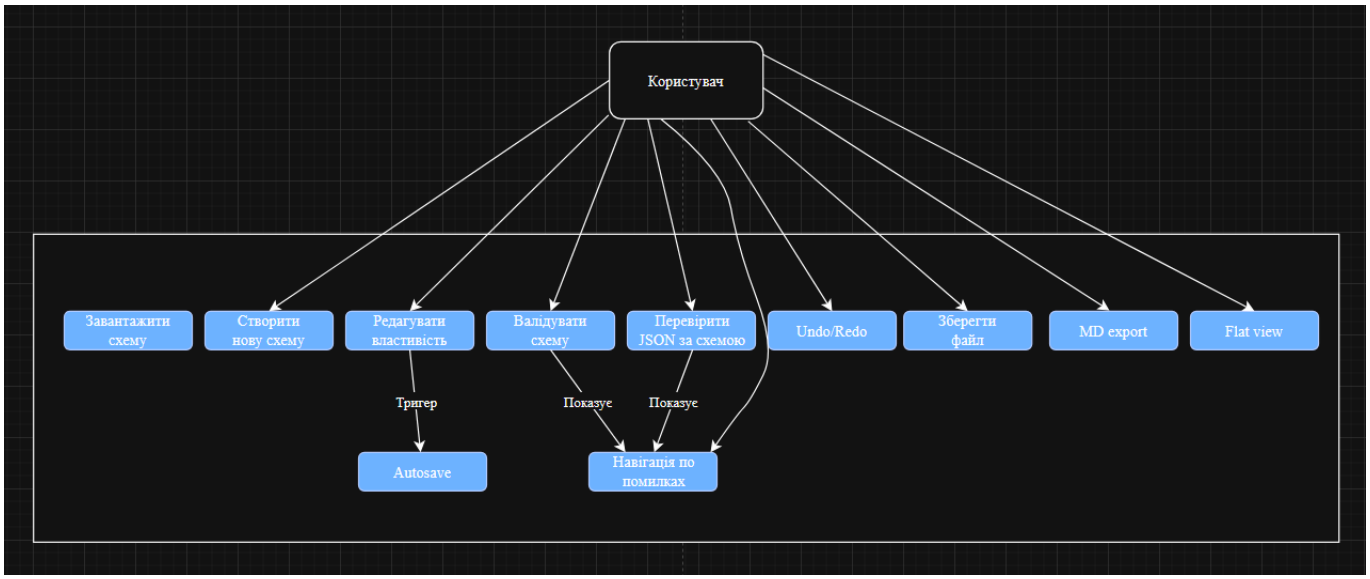


Рис. 1.4.1 — Use Case діаграма системи

1.5. Концептуальна модель системи

Архітектура застосунку базується на патерні MVVM (Model-View-ViewModel), що забезпечує чітке розділення відповідальностей між компонентами. Рівень Model містить бізнес-логіку та дані, незалежні від інтерфейсу користувача. До його основних класів належать JsonSchemaDocument, що представляє документ схеми, JsonPropertyMetadata для метаданих властивості та ValidationResult з результатами валідації. Рівень View реалізований декларативно в XAML-файлах, таких як MainWindow.xaml та FlatViewWindow.xaml, та відповідає виключно за візуальне представлення. Рівень ViewModel, з якого ключовим є MainViewModel, виступає посередником: він надає дані для прив'язки до View та інкапсулює команди для взаємодії користувача, такі як відкриття файлу або валідація, спираючись на сервіси (JsonSchemaService, ValidationContext) та менеджер команд.

Для вирішення конкретних архітектурних задач у Core Layer системи реалізовано кілька патернів проєктування. Strategy Pattern інкапсулює різні алгоритми валідації (синтаксис, структура, дані). Command Pattern дозволяє реалізувати надійний механізм Undo/Redo, інкапсулюючи кожну операцію редагування як окремий об'єкт. Observer Pattern забезпечує слабку зв'язність між компонентами, дозволяючи сповіщувати підписників (наприклад, AutoSaveObserver) про зміни в схемі. Template Method визначає загальний каркас алгоритмів обробки JSON, а Flyweight Pattern оптимізує використання пам'яті за рахунок повторного використання об'єктів для властивостей з однаковими типами та форматами.

Взаємодія компонентів у системі може бути проілюстрована на прикладі ключових процесів. Послідовність дій під час валідації схеми (див. Рис. 1.5.1) починається з команди користувача, передається через MainViewModel до ValidationContext, який, використовуючи відповідну стратегію (наприклад, SyntaxStrategy або SchemaStrategy), виконує перевірку та повертає результат для відображення. Процес редагування властивості з подальшою можливістю скасування (Рис. 1.5.2) включає створення команди EditPropertyCommand, її виконання через CommandManager (що оновлює UndoStack) та сповіщення AutoSaveObserver через SchemaChangeNotifier для запуску таймера авто-збереження.

Загальна діаграма класів (Рис. 1.5.3) демонструє зв'язки між основними компонентами, як-от MainViewModel, SchemaCommandManager, ValidationContext та SchemaChangeNotifier, а також структурні відношення, такі як успадкування (FlatViewProcessor від абстрактного JsonProcessorBase) та асоціації. Життєвий цикл документа описується діаграмою станів (Рис. 1.5.4), яка показує переходи між ключовими станами: від порожнього (Empty) через завантаження (Loading) та редагування (Editing) до збереження (Saving) та валідації (Validating), з обробкою помилок та підтвердженням збереження при закритті.

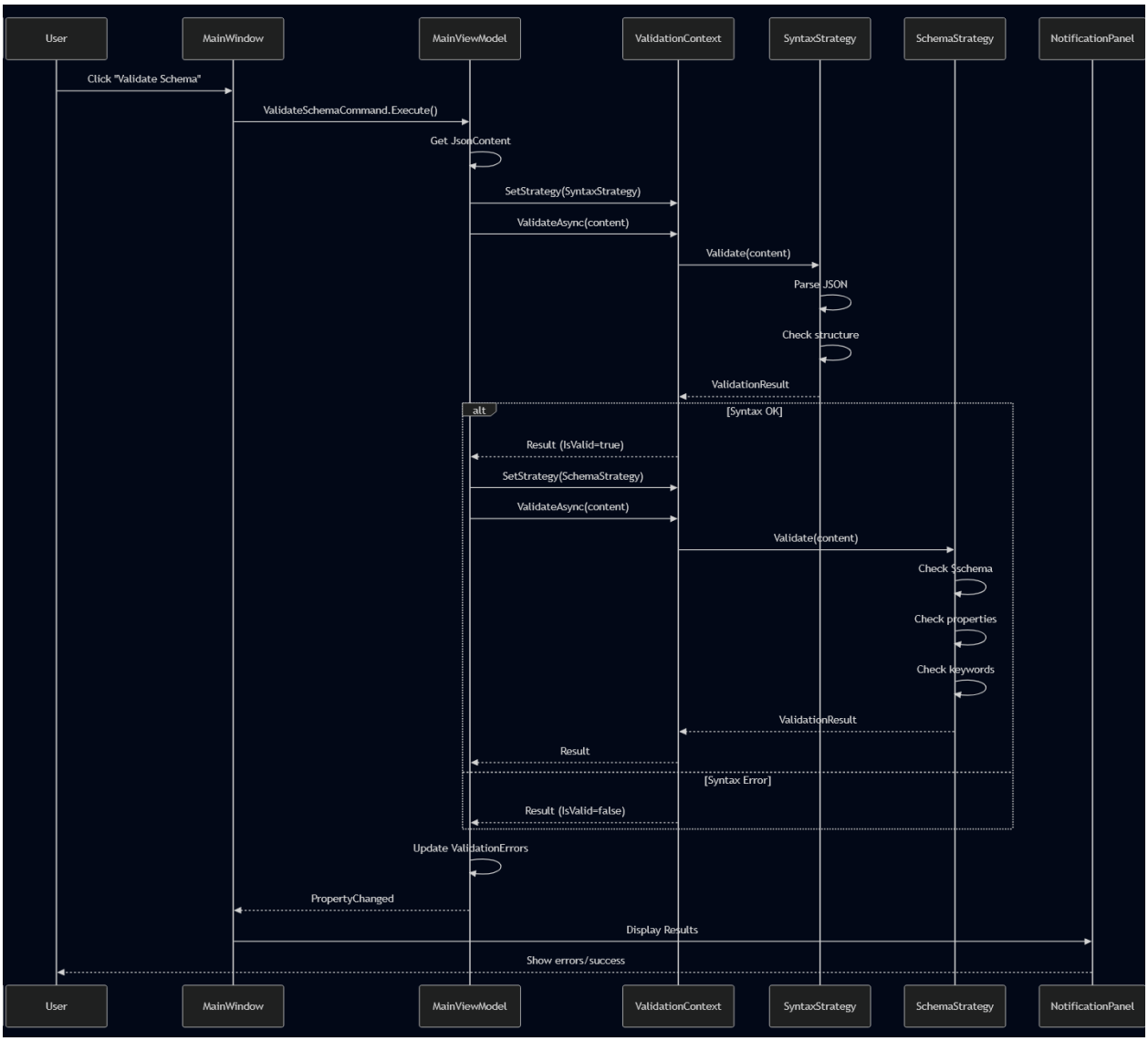


Рис. 1.5.1 — Sequence діаграма процесу валідації

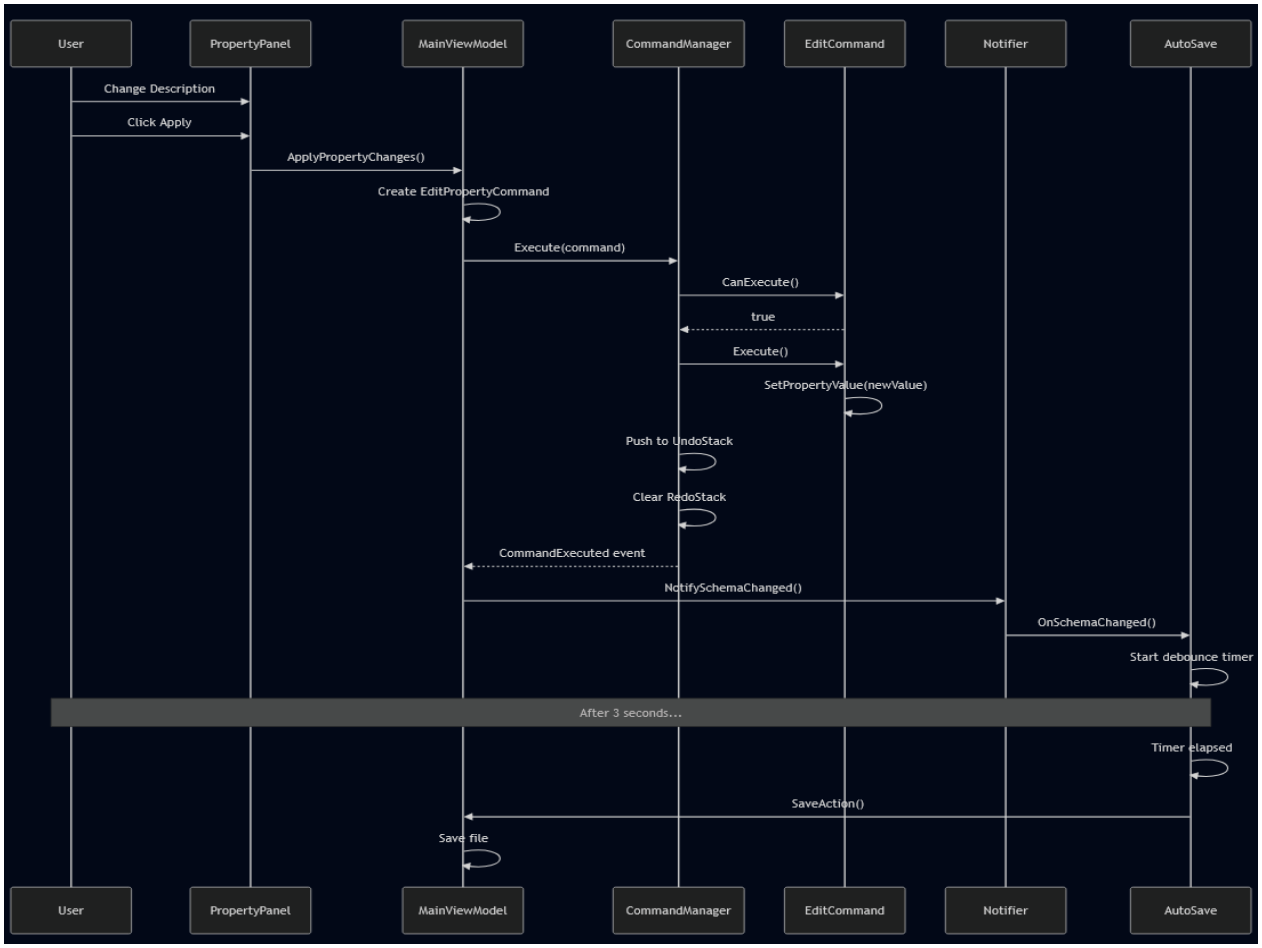


Рис. 1.5.2 — Sequence діаграма процесу валідації

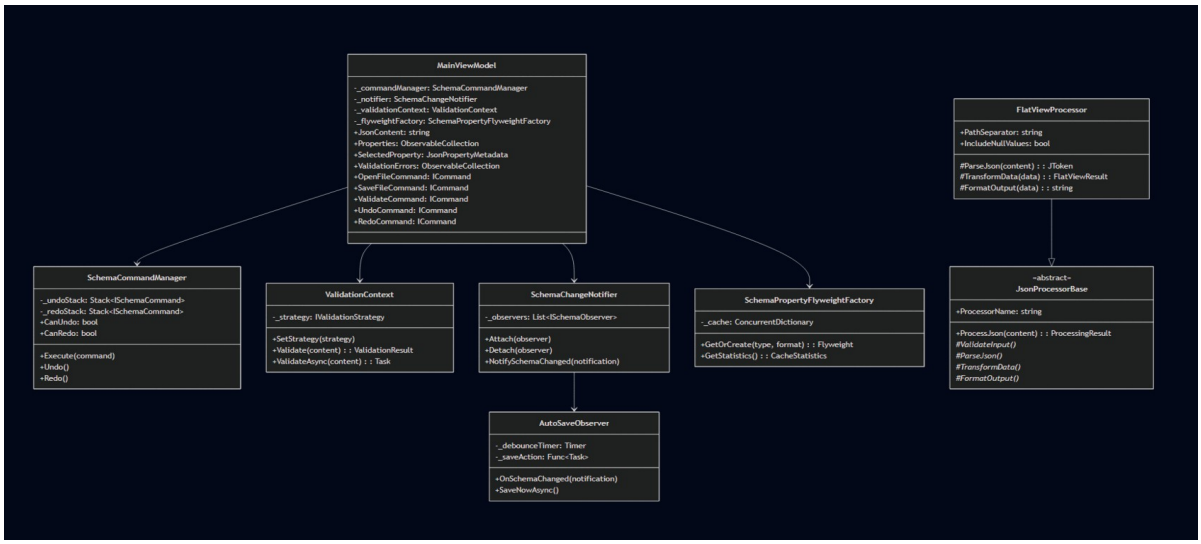


Рис. 1.5.3 — Sequence діаграма процесу валідації

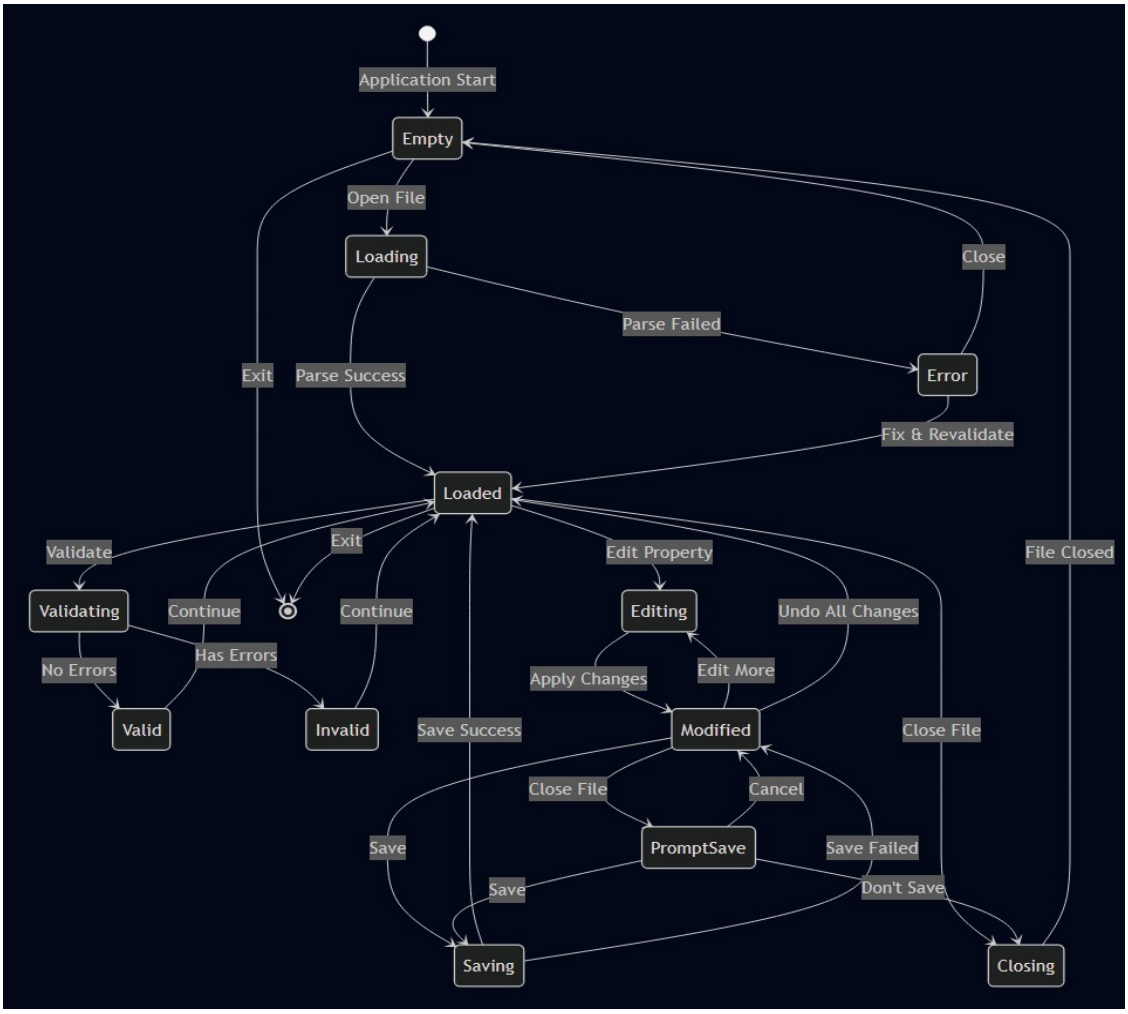


Рис. 1.5.4 — Sequence діаграма процесу валідації

1.6. Вибір мови програмування та середовища розробки

Платформою розробки застосунку обрано C# та .NET 8. .NET 8 є останньою версією з довгостроковою підтримкою (LTS), що гарантує стабільність для корпоративного середовища та сучасні можливості, включаючи оптимізації продуктивності, сучасні можливості C# 12 та покращену роботу з JSON. Статична типізація C#, підтримка Nullable Reference Types та прості шаблони асинхронного програмування роблять його ідеальним вибором для створення надійного десктопного застосунку зі складною логікою.

Для побудови користувацького інтерфейсу вибрано WPF (Windows Presentation Foundation). Це зріла технологія, спеціально розроблена для створення Windows-застосунків з багатими інтерфейсами. Її переваги включають потужну систему прив'язки даних (Data Binding), що фундаментально підтримує архітектуру MVVM, а також декларативний опис інтерфейсу за допомогою XAML, що дозволяє чітко розділити логіку та дизайн. Порівняно з сучасними крос-платформними рішеннями (MAUI), WPF забезпечує більш стабільну та продуктивну роботу на Windows.

Для парсингу та роботи з JSON обрані дві ключові бібліотеки. Newtonsoft.Json (Json.NET) обрано через її надзвичайну поширеність, зрілість, зручний API LINQ to JSON та, що важливо, повну сумісність з бібліотекою схем. Хоча вбудований System.Text.Json пропонує кращу продуктивність, Newtonsoft.Json надає детальнішу інформацію про помилки парсингу, що критично важливо для редактора. NJsonSchema є стандартом де-факто для роботи з JSON Schema в екосистемі .NET. Вона надає повну підтримку актуальних стандартів (Draft-07), надійну валідацію даних за схемою та зручний API для програмної роботи зі схемами.

Для реалізації текстового редактора з підсвічуванням синтаксису обрано компонент AvalonEdit. Це потужний, відкритий редактор, розроблений для інтеграції в WPF-застосунки. Він підтримує всі необхідні функції: підсвічування синтаксису через конфігураційні файли, нумерацію рядків, згортання блоків коду, ефективну роботу з великими файлами та нативну інтеграцію з механізмами WPF, такими як прив'язка даних.

Тестування кодової бази буде забезпечено сучасним стеком бібліотек. xUnit обрано як основний фреймворк для модульного тестування завдяки його чистому синтаксису, підтримці паралельного виконання та активної розробки. Для ізоляції та створення макетів залежностей використовуватиметься Moq, а для написання зрозумілих та читабельних тверджень (assertions) у тестах — бібліотека

FluentAssertions. Цей набір інструментів забезпечує ефективний процес розробки, що підтримує якість коду.

1.7. Проєктування розгортання системи

Застосунок JSON Schema Tool розроблений для локального використання на робочих станціях розробників під керуванням операційних систем Windows. Системні вимоги поділено на мінімальні та рекомендовані. Для мінімальної роботи необхідна 64-бітна Windows 10, процесор рівня Intel Core i3 або AMD Ryzen 3, 4 ГБ оперативної пам'яті та 100 МБ вільного місця на диску. Рекомендовано використовувати Windows 11 з процесором Intel Core i5 / AMD Ryzen 5, 8 ГБ оперативної пам'яті та 200 МБ дискового простору для комфортної роботи з файлами до 10 МБ. Обов'язковою умовою є наявність середовища виконання .NET 8.0 Runtime на цільовій машині, що є єдиною зовнішньою залежністю.

Архітектура розгортання є локальною та автономною. Після встановлення .NET Runtime, застосунок працює як єдиний виконуваний файл JsonTool.exe в контексті середовища виконання .NET. Він завантажує необхідні бібліотеки (Newtonsoft.Json, NJsonSchema, AvalonEdit) з тієї ж директорії.

Процес встановлення спрощений та не вимагає прав адміністратора, реалізуючи модель portable-застосунку. Він починається з перевірки або встановлення .NET 8 Runtime. Користувач завантажує архів застосунку (наприклад, JsonTool-v1.0.0.zip) та розпаковує його в будь-яку зручну директорію. Структура розпакованих файлів містить основний виконуваний файл, бібліотеки залежностей.

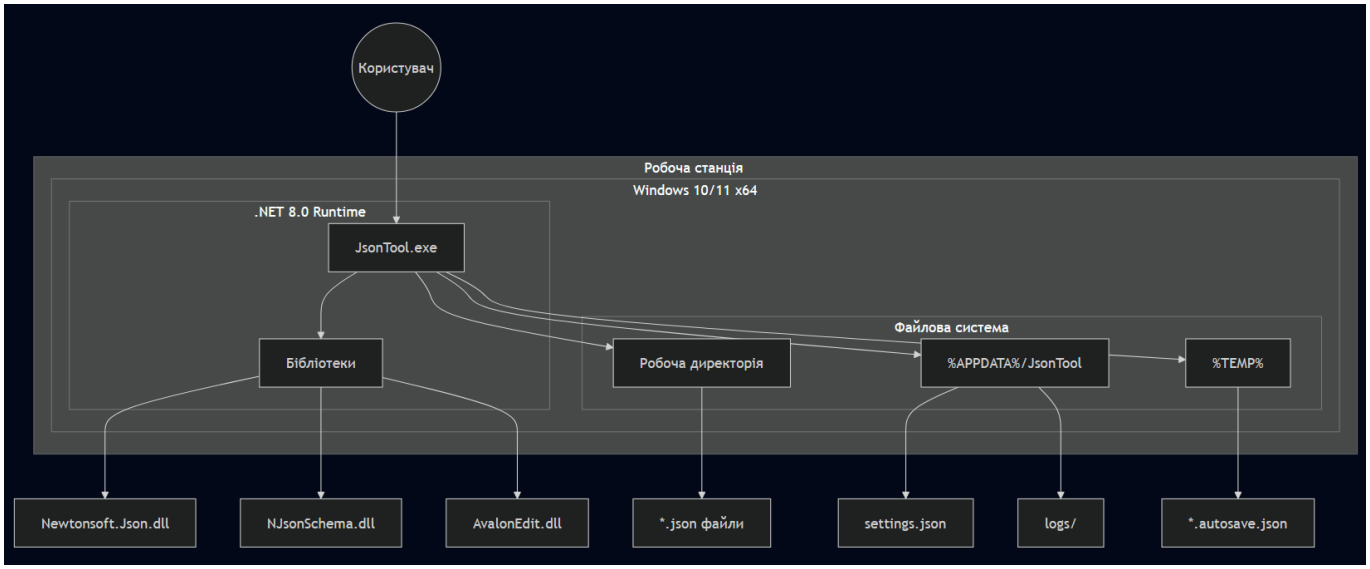


Рис. 1.7.1 — Діаграма розгортання системи

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Архітектура системи

2.1.1. Специфікація системи

Застосунок JSON Schema Tool реалізує комплексний набір функцій, орієнтованих на зручну роботу з файлами JSON Schema. Основний функціонал системи поділено на чотири ключові групи.

Функції роботи з файлами дозволяють повноцінно керувати документами. Користувач може відкривати файли схем через стандартний системний діалог або за допомогою механізму drag-and-drop, зберігати зміни за допомогою гарячої клавіші Ctrl+S та створювати нові документи з базового шаблону. Для захисту даних реалізовано надійний механізм авто-збереження з технікою debounce, який автоматично зберігає файл через 3 секунди після останньої зміни, запобігаючи втраті інформації.

Інструменти редагування забезпечують комфортну роботу з вмістом. Текстовий редактор на базі AvalonEdit надає стандартні можливості, такі як підсвічування синтаксису JSON (з різними кольорами для ключів, рядків, чисел та логічних значень), нумерацію рядків та згортання блоків коду для навігації по великих файлах. Для структурованого редагування властивостей схеми існує окрема панель, яка дозволяє змінювати метадані, такі як тип, формат, текстовий опис та приклади значень. Всі дії користувача захищені механізмом Undo/Redo, який підтримує історію до 50 операцій.

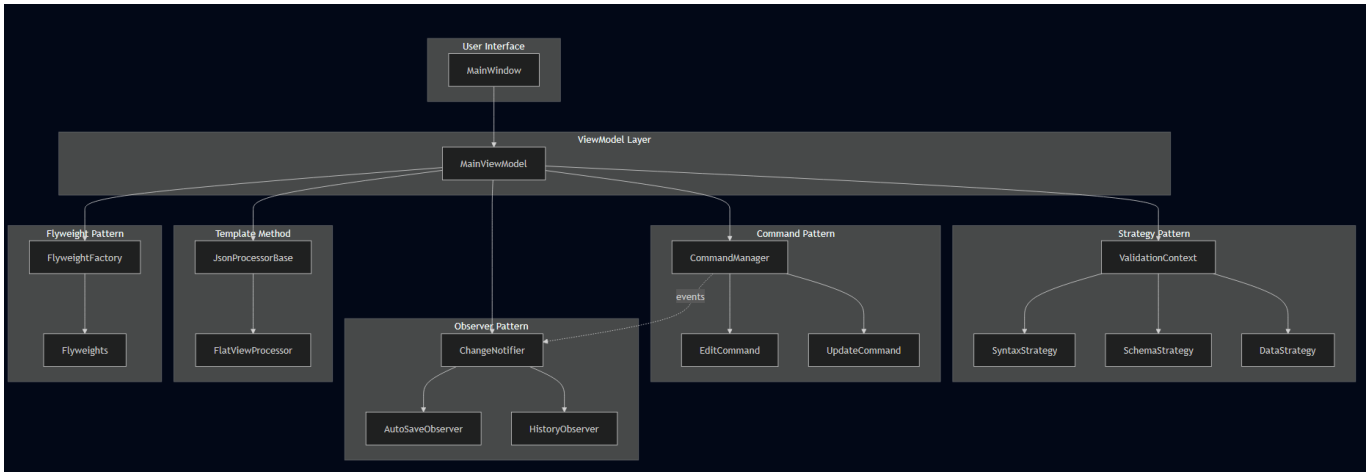
Система валідації є центральною частиною застосунку та працює на трьох рівнях. По-перше, здійснюється швидка перевірка синтаксису JSON у реальному часі.

По-друге, система валідує структуру документа на відповідність стандарту JSON Schema Draft-07, виявляючи логічні помилки. По-третє, існує можливість перевірити будь-який зовнішній JSON-документ на відповідність поточній схемі. Результати перевірок наочно відображаються з детальними повідомленнями про помилки, включаючи номер рядка та шлях у документі, що спрощує їх виправлення.

Додаткові функції експорту та візуалізації розширюють застосування застосунку. Користувач може експортувати структуру поточної схеми у вигляді акуратної Markdown-таблиці, що ідеально підходить для включення в технічну документацію проекту. Для аналізу складних структур реалізовано спеціальний режим Flat View, який трансформує вкладену ієрархію JSON Schema у плоский список всіх властивостей з повними шляхами, що значно полегшує розуміння та навігацію.

2.1.2. Вибір та обґрунтування патернів реалізації

Архітектура JSON Schema Tool ґрунтується на п'яти ключових патернах проєктування, які інтегровані між собою для створення гнучкої, ефективної та легко підтримуваної системи. Взаємодія цих патернів візуалізована на діаграмі.



2.1 — Діаграма взаємодії всіх патернів

Першим фундаментальним патерном є Strategy Pattern (Стратегія валідації), який вирішує проблему необхідності підтримувати різні типи перевірки JSON — синтаксичну, структурну схеми та валідацію даних за схемою. Замість використання

розгалужених умовних конструкцій, система визначає спільний інтерфейс IValidationStrategy з методом Validate(). Конкретні реалізації цього інтерфейсу — SyntaxValidationStrategy, SchemaValidationStrategy та JsonBySchemaValidationStrategy — інкапсулюють логіку кожної перевірки. Клас ValidationContext управляє поточною стратегією та делегує їй виконання. Як показано на Рис. 2.2 — Діаграма класів Strategy Pattern, це дозволяє динамічно змінювати алгоритм валідації. Наприклад, у MainViewModel процес перевірки починається з синтаксису, а потім, за його успіху, переходить до валідації структури схеми:

```
_validationContext.SetStrategy(new SyntaxValidationStrategy());
var syntaxResult = await _validationContext.ValidateAsync(JsonContent);
if (syntaxResult.IsValid) {
    _validationContext.SetStrategy(new SchemaValidationStrategy());
    await _validationContext.ValidateAsync(JsonContent);
}
```

Такий підхід дотримується принципу Open/Closed, дозволяє легко додавати нові типи валідації та ізолює тестування кожного алгоритму.

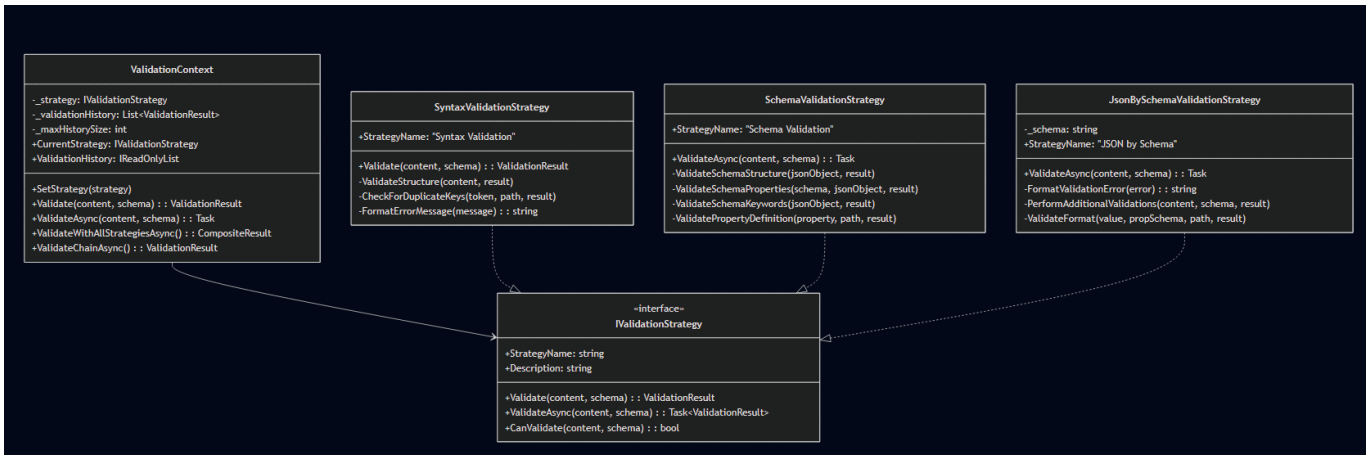


Рис. 2.2 — Діаграма класів Strategy Pattern

Для забезпечення надійного контролю над діями користувача реалізовано Command Pattern (Команди редагування), що надає повноцінну підтримку Undo/Redo. Кожна операція зміни, наприклад, редагування опису властивості, інкапсулюється в

об'єкт команди, який реалізує інтерфейс ISchemaCommand з методами Execute() та Undo(). Як ілюструє Рис. 2.3 — Діаграма класів Command Pattern, центральний SchemaCommandManager керує двома стеками: _undoStack для виконаних команд та _redoStack для скасованих. Коли користувач змінює властивість, створюється команда, яка зберігає старі та нові дані:

var command = new EditPropertyCommand(property, "Description", oldValue, newValue);

_commandManager.Execute(command);

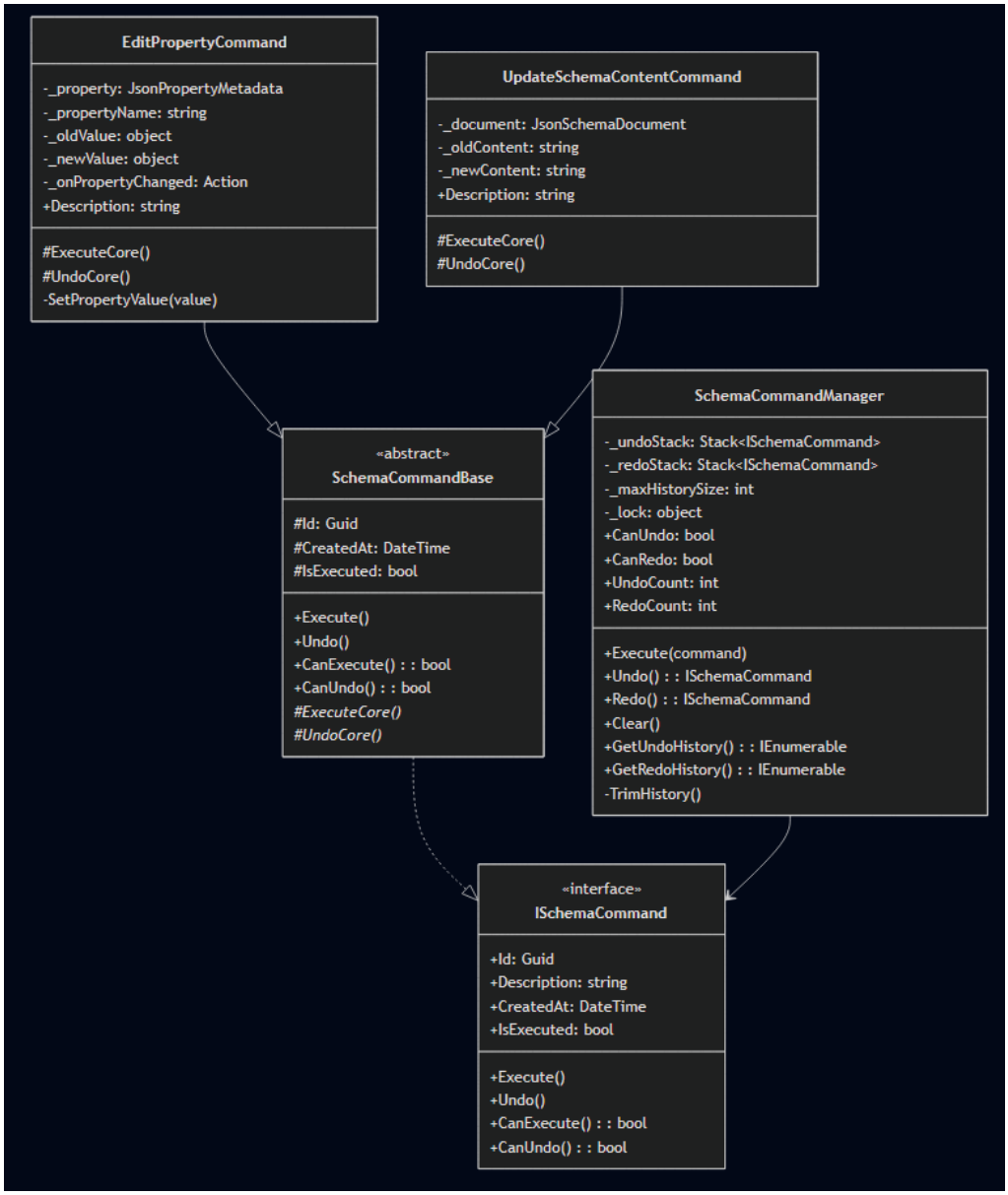


Рис. 2.3 — Діаграма класів Command Pattern

При виклику Undo() команда витягується зі стеку undo, виконує свою логіку відкату та поміщається в стек redo. Менеджер також автоматично обмежує розмір історії та очищує стек redo при виконанні нової команди, що гарантує консистентність стану. Це забезпечує повну оборотність дій та спрощує додавання нових типів операцій.

Третім критичним патерном є Observer Pattern (Спостерігач за змінами), який забезпечує слабку зв'язність між компонентами, що мають реагувати на модифікації схеми. Як видно з Рис. 2.4 — Діаграма класів Observer Pattern, клас SchemaChangeNotifier виступає в ролі суб'єкта, який веде список підписаних спостерігачів, що реалізують інтерфейс ISchemaObserver. Найважливішим спостерігачем є AutoSaveObserver. Коли відбувається зміна в документі, MainViewModel викликає _notifier.NotifySchemaChanged(), що сповіщає всіх підписників. AutoSaveObserver у відповідь запускає debounce-таймер на 3 секунди. Якщо протягом цього часу не надходять нові зміни, таймер спрацьовує та викликає функцію збереження. Цей механізм ефективно запобігає надмірному навантаженню диска при частому редагуванні. Патерн дозволяє динамічно додавати нові спостерігачі (наприклад, для логування або відправки аналітики) без жодних змін у логіці редагування.

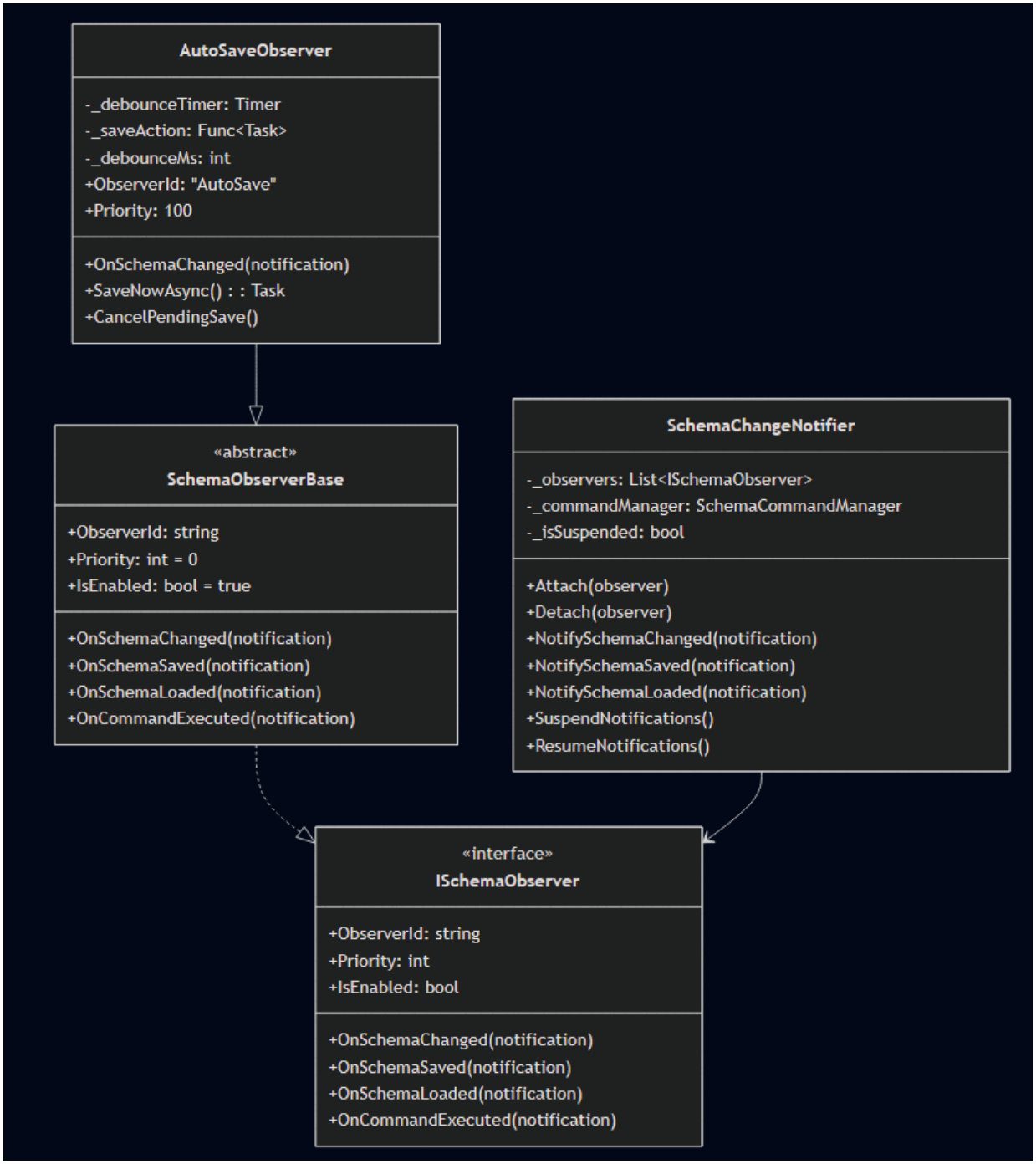


Рис. 2.4 — Діаграма класів Observer Pattern

Для стандартизації алгоритмів обробки JSON використано Template Method Pattern (Шаблонний метод обробки). Абстрактний клас JsonProcessorBase визначає каркас алгоритму у методі ProcessJson(), який задає незмінну послідовність кроків: валідація вхідних даних, парсинг JSON, трансформація даних та форматування виводу. Конкретні процесори, такі як FlatViewProcessor (який перетворює вкладену

структуру на плоский список), перевизначають ці етапи, зберігаючи загальну структуру. Це добре відображено на Рис. 2.5 — Діаграма класів Template Method Pattern. Наприклад, основний метод базового класу виглядає так:

```
public ProcessingResult ProcessJson(string content) {
    if (!ValidateInput(content)) { /*...*/ }
    var parsedData = ParseJson(content);
    var transformedData = TransformData(parsedData);
    var output = FormatOutput(transformedData);
    return new ProcessingResult { Output = output };
}
```

Такий підхід забезпечує повторне використання коду, контроль над структурою алгоритму та надає hook-методи для додаткової логіки.

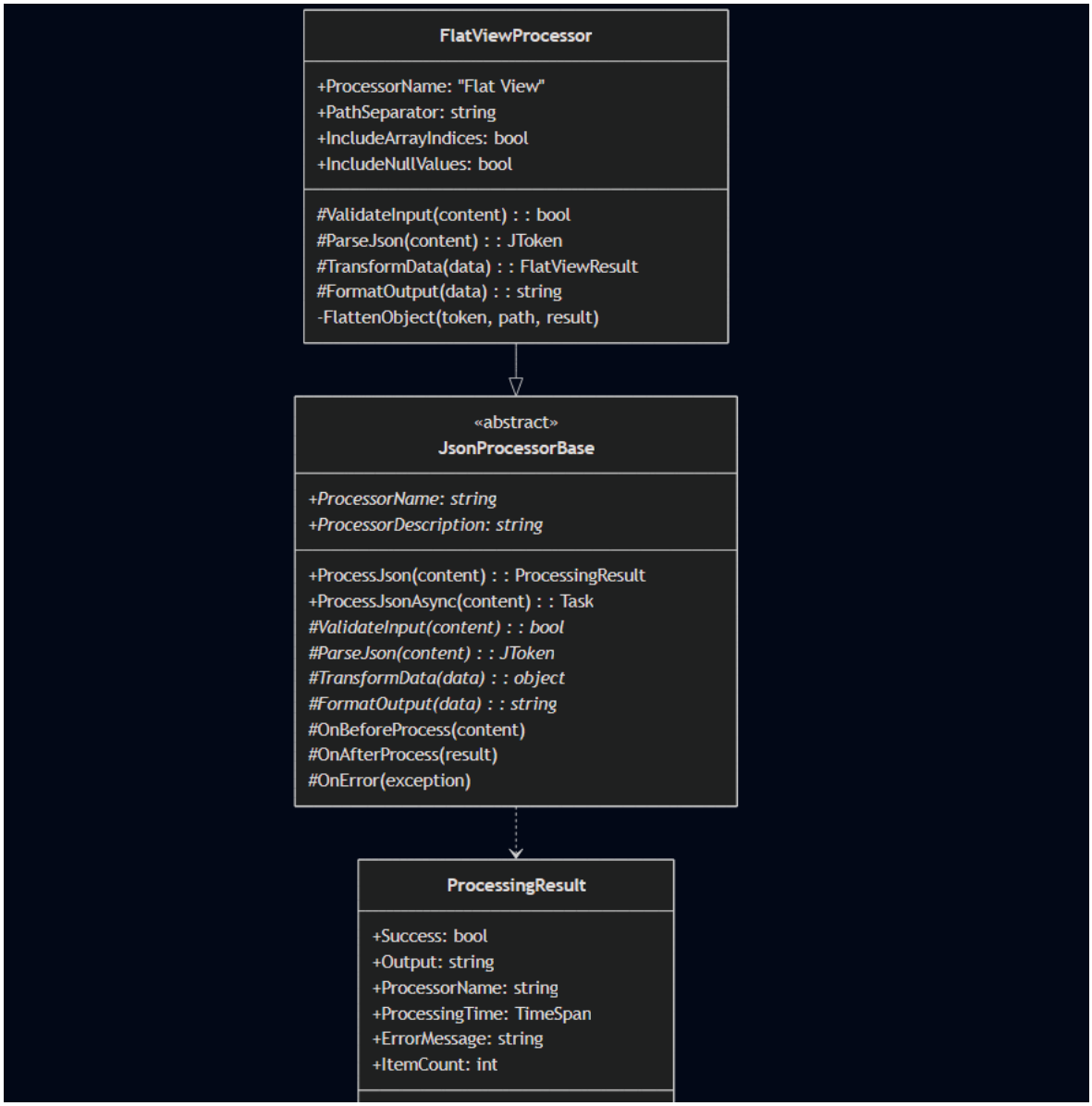


Рис. 2.5 — Діаграма класів Template Method Pattern

Останнім оптимізаційним патерном є Flyweight Pattern (Легковаговик для економії пам'яті), спрямований на ефективну роботу з великими схемами, де багато властивостей мають однакові тип та формат. Замість зберігання цих повторюваних даних окремо для кожної властивості, система використовує фабрику SchemaPropertyFlyweightFactory, яка створює та кешує об'єкти SchemaPropertyFlyweight. Ці об'єкти містять внутрішній незмінний стан —

комбінацію type, format, pattern. При створенні нової властивості система запитує фабрику: var flyweight = _flyweightFactory.GetOrCreate("string", "email");

Якщо легковаговик з такою комбінацією вже існує в кеші, він повертається, а його лічильник використання збільшується. Інші, унікальні дані властивості (ім'я, опис) зберігаються окремо. Фабрика також надає статистику ефективності кешу. Структура цього патерну представлена на Рис. 2.6 — Діаграма класів Flyweight Pattern. Це дозволяє значно зменшити споживання пам'яті в схемах з сотнями або тисячами однотипних полів.

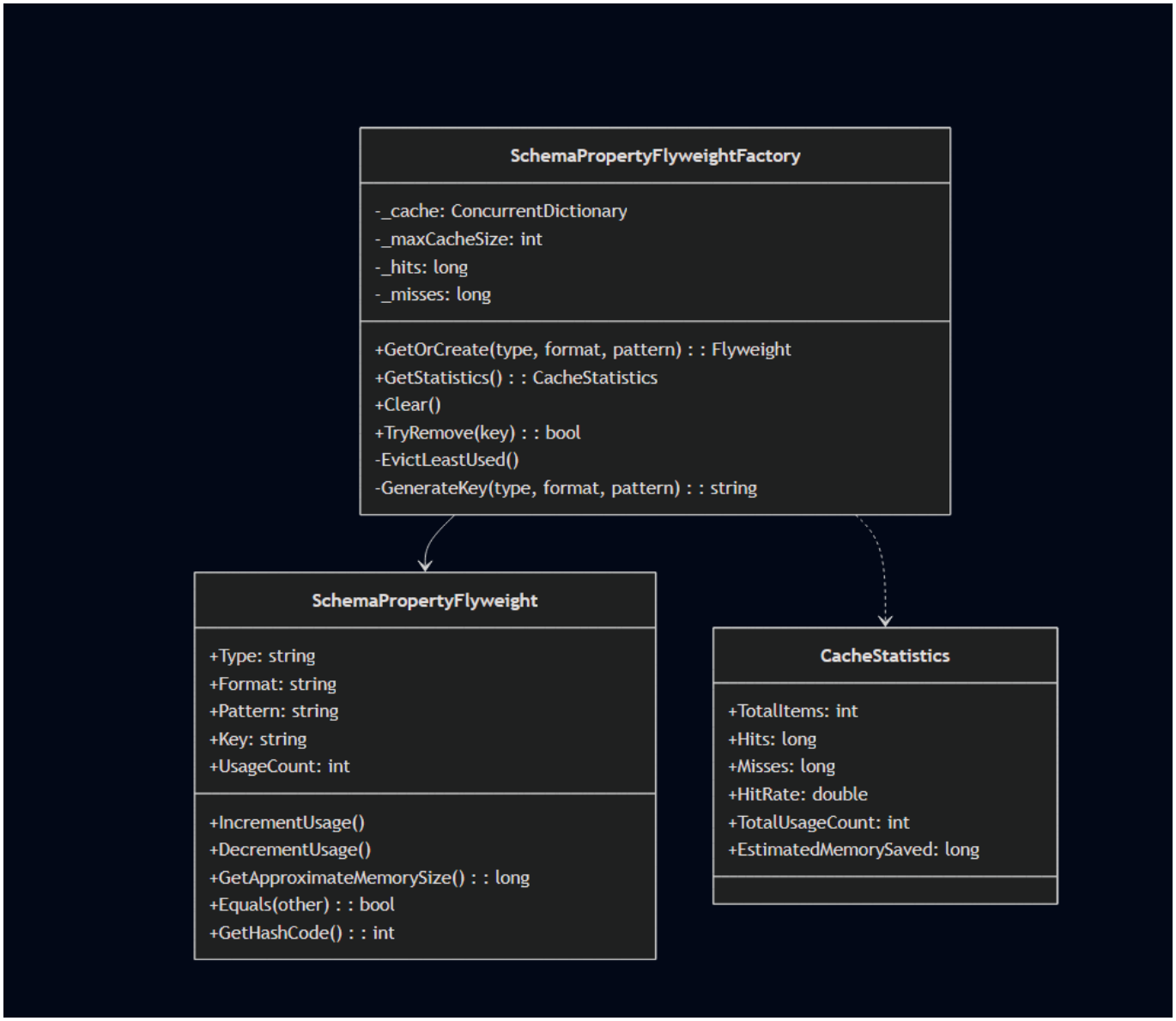


Рис. 2.6 — Діаграма класів Flyweight Pattern

Усі ці патерни тісно інтегровані в архітектурі застосунку. Клас MainViewModel виступає як центральний координатор, який ініціалізує та використовує всі компоненти: він делегує валідацію ValidationContext, керує історією операцій через SchemaCommandManager, сповіщає про зміни через SchemaChangeNotifier, використовує процесори для трансформації даних та застосовує FlyweightFactory для створення властивостей. Така модульна, заснована на патернах архітектура забезпечує високу гнучкість, легкість тестування окремих компонентів та спрощує майбутнє розширення функціоналу.

2.2. Інструкція користувача

Для початку роботи з програмою JSON Schema Tool відкрийте папку, куди ви її встановили, та запустіть файл JsonTool.exe. Після запуску ви побачите головне вікно з текстовим редактором, списком властивостей та панеллю інструментів. Щоб почати роботу з новим документом, виберіть у меню пункт File та команду New, після чого в редакторі з'явиться готовий шаблон базової JSON Schema, який ви можете відредагувати за своїми потребами.

Якщо вам потрібно відкрити вже існуючий файл схеми, скористайтеся командою Open у тому ж меню File. У відкритому діалозі виберіть потрібний файл з розширенням .json та натисніть кнопку відкриття. Для ще більш швидкого доступу ви можете просто перетягнути потрібний файл з Провідника прямо у вікно програми. Зберігати зроблені зміни можна за допомогою команди Save у меню File. Програма також має зручну функцію авто-збереження, яка автоматично зберігає ваш документ через три секунди після того, як ви припините редагування, що допомагає уникнути втрати даних у разі непередбачених обставин.

Основним інструментом для роботи зі схемою є панель редагування властивостей. У лівій частині вікна розташований список всіх властивостей вашої схеми. Коли ви вибираєте будь-яку властивість у цьому списку, праворuch з'являється панель Edit Property. Тут ви можете змінити різні параметри: Type визначає тип даних (наприклад, текст, число), Format задає спеціальний формат значення (такий як електронна пошта або посилання), поле Description призначене для текстового опису властивості, а Examples — для прикладів коректних значень. Після внесення змін натисніть кнопку Apply, щоб вони вступили в силу, або Cancel, якщо ви передумали.

У процесі роботи неминуче виникають помилки або необхідність повернутися до попереднього варіанту. Для цього в програмі реалізовані звичні функції скасування та повтору дій. Ви можете скористатися кнопками зі стрілочками на панелі інструментів або стандартними комбінаціями клавіш, щоб відмінити останню дію або повернути щойно скасовану зміну. Це особливо зручно при активному редагуванні складних структур.

Однією з ключових можливостей програми є перевірка коректності. Кнопка Validate Schema запускає повну перевірку поточної схеми. Результати перевірки, включаючи всі знайдені помилки та попередження, з'являються в спеціальній панелі внизу вікна. Якщо ви клацнете на конкретній помилці в цьому списку, курсор у текстовому редакторі автоматично перестрибне на відповідний рядок, де ця проблема виникла. Крім перевірки самої схеми, програма дозволяє перевіряти реальні JSON-дані на відповідність завантаженій схемі. Для цього достатньо натиснути кнопку Check Data, вибрати файл з даними у форматі JSON, і система миттєво покаже, чи відповідають ці дані вимогам вашої схеми.

Для документування та аналізу програма пропонує два додаткових інструменти. Перший — експорт у формат Markdown. Натиснувши відповідну кнопку, ви можете зберегти структуру вашої схеми у вигляді акуратної таблиці,

готової для вставки в технічну документацію. Другий інструмент — Flat View — призначений для роботи зі складними, багаторівневими схемами. Він відкриває окреме вікно, де вся ієрархія властивостей представлена у вигляді зручного плоского списку, де кожен рядок містить повний шлях до властивості та її значення. Для прискорення роботи запам'ятайте основні гарячі клавіші: комбінація для відкриття файлу, для збереження поточних змін, а також клавіші для швидкого скасування та повтору останніх дій.

ВИСНОВКИ

У результаті виконання курсової роботи було розроблено настільний застосунок JSON Schema Tool — повнофункціональний інструмент для редагування, валідації та документування JSON Schema. Застосунок реалізує графічний інтерфейс на базі технології WPF та надає користувачеві широкий набір функцій. Він включає текстовий редактор з підсвічуванням синтаксису JSON, панель для візуального редагування метаданих властивостей схеми, потужну систему багаторівневої валідації (перевірку синтаксису, структури самої схеми та відповідності даних схемі), а також функцію експорту документації у зручному форматі Markdown. Для підвищення зручності та надійності роботи впроваджено систему операцій Undo/Redo з історією до 50 дій та механізм автоматичного збереження з використанням техніки debounce, що запобігає втраті даних.

Архітектура застосунку ґрунтується на системному застосуванні п'яти класичних патернів проєктування (GoF), що забезпечило гнучкість та якість кодової бази. Strategy Pattern був використаний для реалізації різних стратегій валідації. Це дозволило інкапсулювати алгоритми перевірки синтаксису, структури схеми та даних у окремі класи (SyntaxValidationStrategy, SchemaValidationStrategy, JsonBySchemaValidationStrategy), що реалізують спільний інтерфейс IValidationStrategy. Такий підхід дав можливість динамічно змінювати тип перевірки та легко додавати нові, не модифікуючи існуючий код. Command Pattern став основою для надійної системи Undo/Redo. Кожна операція редагування, наприклад зміна опису властивості, інкапсулюється в об'єкт-команду, що реалізує інтерфейс ISchemaCommand та зберігає стан до і після зміни. Центральний SchemaCommandManager керує стеками виконаних та скасованих операцій, забезпечуючи повну оборотність дій користувача.

Для організації слабко зв'язаної взаємодії між компонентами було застосовано Observer Pattern. Клас SchemaChangeNotifier виступає в ролі суб'єкта, який веде список спостерігачів, таких як AutoSaveObserver. Коли схема змінюється, сповіщення розсилається всім підписаним спостерігачам, які можуть відреагувати своїм чином (наприклад, запустити таймер авто-збереження), причому основний код редагування не залежить від їх конкретних реалізацій. Для стандартизації алгоритмів обробки JSON, як-от перетворення вкладеної структури у плоский вигляд (Flat View), використано Template Method Pattern. Абстрактний клас JsonProcessorBase визначає каркас алгоритму в методі ProcessJson(), що задає послідовність кроків: валідація вхідних даних, парсинг JSON, трансформація даних та форматування виводу. Конкретні процесори перевизначають окремі кроки, зберігаючи загальну структуру. Для оптимізації використання пам'яті при роботі з великими схемами, де багато властивостей мають однакові тип та формат, застосовано Flyweight Pattern. SchemaPropertyFlyweightFactory створює та кешує об'єкти SchemaPropertyFlyweight, що містять спільний незмінний стан. Замість дублювання цих даних для кожної властивості, система повторно використовує один і той же легковаговик, економлячи ресурси.

Забезпечено високу якість коду через комплексний підхід. Ключова функціональність, особливо реалізація патернів, покрита модульними тестами (unit-тестами) з використанням фреймворка xUnit, що підтверджує коректність роботи. У процесі розробки застосовано сучасні можливості мови C#, такі як Nullable Reference Types для попередження помилок зі значеннями null та асинхронне програмування через async/await для відзивчивого інтерфейсу. Архітектура застосунку дотримується принципів SOLID, що сприяє чіткому розділенню відповідальностей та спрощує підтримку. Публічні API основних класів супроводжуються детальними XML-коментарями, що покращує розуміння коду та підтримує автоматичну генерацію документації.

Систематичне використання патернів проєктування забезпечило ряд ключових переваг у розробці. Розширюваність архітектури досягнута завдяки здатності безболісно додавати новий функціонал: нові стратегії валідації, типи команд, спостерігачі або процесори обробки можуть бути впроваджені шляхом реалізації відповідних інтерфейсів або розширення базових класів, без необхідності переписувати існуючу логіку. Покращена тестованість є прямим наслідком слабкої зв'язаності компонентів, реалізованої через патерни Observer та Strategy. Кожен модуль, як-от окрема стратегія валідації або спостерігач, може бути протестований ізольовано від решти системи, що підвищує надійність тестів та спрощує виявлення дефектів. Підтримуваність кодової бази значно покращена через чітке розмежування обов'язків між класами та модулями. Розробнику легше зрозуміти логіку, знайти потрібний компонент для змін та оцінити наслідки цих змін, оскільки архітектурні межі чітко визначені патернами. Нарешті, можливість повторного використання базових компонентів, таких як інтерфейси стратегій, базова логіка команд або каркас процесора, відкриває шлях для їх інтеграції в майбутні проєкти або створення бібліотек, що значно прискорює розробку.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1994. — 395 p.
2. Freeman E., Robson E. Head First Design Patterns. 2nd Edition. — O'Reilly Media, 2020. — 672 p.
3. Microsoft .NET Documentation. Windows Presentation Foundation. — URL: https://docs.microsoft.com/dotnet/desktop/wpf/ (дата звернення: 23.12.2025).
4. JSON Schema Specification. Draft-07. — URL: https://json-schema.org/specification-links.html#draft-7 (дата звернення: 23.12.2025).
5. Newtonsoft.Json Documentation. — URL: https://www.newtonsoft.com/json/help/html/Introduction.htm (дата звернення: 23.12.2025).
6. NJsonSchema Documentation. — URL: https://github.com/RicoSuter/NJsonSchema (дата звернення: 23.12.2025).
7. AvalonEdit Text Editor. — URL: http://avalonedit.net/ (дата звернення: 23.12.2025).
8. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Prentice Hall, 2017. — 432 p.
9. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley, 2002. — 560 p.
10. Smith J. WPF Apps With The Model-View-ViewModel Design Pattern. — MSDN Magazine, February 2009.
11. xUnit.net Documentation. — URL: https://xunit.net/docs/getting-started/netcore/cmdline (дата звернення: 23.12.2025).
12. FluentAssertions Documentation. — URL: https://fluentassertions.com/introduction (дата звернення: 23.12.2025).

ДОДАТКИ

1) Приклад JSON Schema

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "$id": "https://example.com/user.schema.json",
 "title": "User",
 "description": "A user account in the system",
 "type": "object",
 "properties": {
  "id": {
   "type": "integer",
   "description": "Unique user identifier",
   "minimum": 1
  },
  "username": {
   "type": "string",
   "description": "User's login name",
   "minLength": 3,
   "maxLength": 50,
   "pattern": "^[a-zA-Z0-9_]+$"
  },
  "email": {
   "type": "string",
   "description": "User's email address",
   "format": "email"
  },
  "fullName": {
```

```
  "type": "string",
  "description": "User's full name"
},
"age": {
  "type": "integer",
  "description": "User's age in years",
  "minimum": 0,
  "maximum": 150
},
"roles": {
  "type": "array",
  "description": "User's assigned roles",
  "items": {
    "type": "string",
    "enum": ["admin", "user", "moderator", "guest"]
  },
  "uniqueItems": true
},
"settings": {
  "type": "object",
  "description": "User preferences",
  "properties": {
    "theme": {
      "type": "string",
      "enum": ["light", "dark", "system"]
    },
    "notifications": {
      "type": "boolean"
```

```
      }
    }
  },
  "createdAt": {
    "type": "string",
    "description": "Account creation timestamp",
    "format": "date-time"
  }
 },
 "required": ["id", "username", "email"],
 "additionalProperties": false
}
```

2) Структура проєкту

```
JsonTool/
├── JsonTool.csproj
├── App.xaml
├── App.xaml.cs
├── README.md
│
├── Core/                          # Патерни проєктування
│   ├── Command/                       # Command Pattern
│   │   ├── ICommand.cs
│   │   ├── ISchemaCommand.cs
│   │   ├── SchemaCommandBase.cs
│   │   ├── CommandManager.cs
│   │   ├── CommandInvoker.cs
│   │   ├── AddPropertyCommand.cs
```

```
│   │   ├── DeletePropertyCommand.cs
│   │   ├── EditPropertyCommand.cs
│   │   ├── UpdatePropertyCommand.cs
│   │   ├── UpdateMetadataCommand.cs
│   │   └── UpdateSchemaContentCommand.cs
│   │
│   ├── Observer/                    # Observer Pattern
│   │   ├── IDocumentObserver.cs
│   │   ├── IDocumentSubject.cs
│   │   ├── ISchemaObserver.cs
│   │   ├── ISchemaSubject.cs
│   │   ├── SchemaObserverBase.cs
│   │   ├── SchemaChangeNotifier.cs
│   │   ├── DocumentSubject.cs
│   │   ├── AutoSaveObserver.cs
│   │   ├── HistoryObserver.cs
│   │   ├── UIUpdateObserver.cs
│   │   └── ObserverIntegration.cs
│   │
│   ├── Strategy/                    # Strategy Pattern
│   │   ├── IExportStrategy.cs
│   │   ├── IJsonViewStrategy.cs
│   │   ├── ExportContext.cs
│   │   ├── JsonExportStrategy.cs
│   │   ├── MarkdownExportStrategy.cs
│   │   ├── FlatViewStrategy.cs
│   │   ├── TreeViewStrategy.cs
│   │   └── Validation/
```

```
│   │       ├── IValidationStrategy.cs
│   │       ├── ValidationContext.cs
│   │       ├── ValidationStrategyFactory.cs
│   │       ├── SyntaxValidationStrategy.cs
│   │       ├── SchemaValidationStrategy.cs
│   │       └── JsonBySchemaValidationStrategy.cs
│   │
│   ├── TemplateMethod/                 # Template Method Pattern
│   │   ├── JsonProcessorBase.cs
│   │   ├── JsonValidatorBase.cs
│   │   ├── FlatViewProcessor.cs
│   │   ├── SimpleFlatViewProcessor.cs
│   │   ├── SchemaProcessor.cs
│   │   ├── JsonSchemaValidator.cs
│   │   ├── JsonDataValidator.cs
│   │   ├── MarkdownExporter.cs
│   │   └── MarkdownTableExporter.cs
│   │
│   ├── Flyweight/                      # Flyweight Pattern
│   │   ├── ITextFormat.cs
│   │   ├── TextFormat.cs
│   │   ├── TextFormatFactory.cs
│   │   ├── SchemaPropertyFlyweight.cs
│   │   ├── SchemaPropertyFlyweightFactory.cs
│   │   ├── SchemaPropertyContext.cs
│   │   ├── SchemaFlyweightParser.cs
│   │   └── SyntaxHighlighter.cs
│   │
```

```
│   └── Models/                    # Моделі даних
│       ├── JsonSchemaDocument.cs
│       ├── JsonPropertyMetadata.cs
│       ├── SchemaProperty.cs
│       ├── ValidationResult.cs
│       └── ValidationError.cs
│
├── Services/                      # Сервіси
│   ├── IJsonSchemaService.cs
│   ├── JsonSchemaService.cs
│   ├── AutoSaveService.cs
│   └── AutoSaveSettings.cs
│
├── ViewModels/                    # MVVM ViewModel
│   ├── ViewModelBase.cs
│   ├── RelayCommand.cs
│   ├── MainViewModel.cs
│   ├── MainViewModelNew.cs
│   ├── MainViewModelComplete.cs
│   ├── MainViewModelDI.cs
│   └── PropertyEditorViewModel.cs
│
├── Views/                         # MVVM View
│   ├── MainWindow.xaml
│   ├── MainWindow.xaml.cs
│   ├── FlatViewWindow.xaml
│   └── FlatViewWindow.xaml.cs
│
```

```
├── Converters/                    # WPF конвертери
│   └── BoolToVisibilityConverter.cs
│
├── Highlighting/                  # Підсвічування синтаксису
│   ├── JsonHighlightingLoader.cs
│   └── JsonSyntaxHighlighter.cs
│
├── Resources/                     # Ресурси
│   └── JsonHighlighting.xshd
│
└── Samples/
```