Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Інструмент візуального створення редакторів користувача для рушія Unity

Назва теми

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.2201119.01.24.ПЗ

Виконав студент IV курсу, група ІПЗ-22-1 _____ Роман ЮРЧУК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник канд. пед. наук, доцент _____ Наталія ПРАВОРСЬКА
Науковий ступінь, вчене звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. тех. наук, доцент _____ Оксана ЯШИНА
Науковий ступінь, вчене звання Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

1 січня 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
02.01.2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Юрчуку Роману Володимировичу
Прізвище, ім'я, по батькові студента

1. Тема роботи Інструмент візуального створення редакторів користувача для рушія Unity
Керівник роботи Праворська Наталія Іванівна, канд. пед. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. №7
2. Строк подання студентом роботи на кафедру 01.06.2026 р.
3. Вихідні дані до роботи Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі проєкту, проєктування програмного забезпечення, програмна реалізація та тестування застосунку.
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Три креслення
1. Діаграма класів схем даних та модифікаторів макету
2. Діаграми послідовностей системи віртуалізації
3. Діаграма послідовності формування візуального дерева

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Яшина О. М., доцент	05.05.26	05.05.26
Антиплагіат	Форкун Ю. В., доцент	05.05.26	05.05.26

7. Дата видачі завдання « 02 » січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Ознайомлення з тематикою дипломного проєктування, визначення та узгодження індивідуальної теми кваліфікаційної роботи (КвР)	01.12– 31.12.2025	
2 Збір матеріалу за темою КвР; дослідження предметної області, в якій планується використання програмного забезпечення (ПЗ), визначення задач та вимог, розробка технічного завдання	01.01 – 20.02.2026	
3 Проєктування програмного забезпечення	21.02 – 20.03 2026	
4 Програмна реалізація з використанням відповідних засобів розроблення. Тестування ПЗ	21.03 – 30.04.2026	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2026	
6 Попередній захист КвР	Травень 2026	Згідно графіка
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки.	26.05 – 30.05.2026	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	з 01.06.2026	

Студент _____ Роман ЮРЧУК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Наталія ПРАВОРСЬКА
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Інструмент візуального створення редакторів користувача для рушія Unity».

Автор роботи: Юрчук Роман Володимирович

Керівник роботи: Праворська Наталія Іванівна

Пояснювальна записка: 77 с., 21 рис., 3 табл., 6 дод., 41 джерело.

Графічна частина: 3 креслення.

ІНСТРУМЕНТ, ВІЗУАЛЬНИЙ КОНСТРУКТОР, ПРОГРАМНИЙ ПРОДУКТ, КОРИСТУВАЦЬКИЙ ДОСВІД, UNITY, C#, РЕДАКТОР, NO-CODE

Метою роботи є розроблення інструментального засобу для візуального проєктування користувацьких інтерфейсів редактора у середовищі Unity, що усуває необхідність написання програмного коду та рекомпіляції проєкту.

У кваліфікаційній роботі визначено специфіку розширення базового функціоналу редактора Unity, проведено аналіз наявних засобів для конструювання інтерфейсів, спроєктовано модульну архітектуру системи на основі шаблону MVVM, реалізовано механізми зберігання поліморфних конфігурацій, автоматичної генерації макетів, конвеєра створення представлень із макетів, динамічного перезавантаження при редагуванні, а також набір готових до використання вузлів та модифікаторів.

Для реалізації програмного забезпечення використано ігровий рушій Unity, мову програмування C#, фреймворк UI Toolkit та механізми рефлексії.

В результаті виконання кваліфікаційної роботи здійснена програмна реалізація системи для візуального створення редакторів користувача, проведено функціональне тестування розроблених підсистем, підтверджено стабільність та надійність роботи інструмента та обґрунтовано доцільність його впровадження у процеси створення внутрішнього інструментарію ігрових студій та компаній, що спеціалізуються на розробці інтерактивних застосунків.

27.05.2026
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ.2201119.01.24.ПЗ	Пояснювальна записка	77		
2	А4		Завдання на кваліфіка-	1		
			ційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ.2201119.01.24.Е8	Діаграма класів схем	1		
			даних та модифікаторів			
			макету	1		
5	А3	КвРІПЗ.2201119.01.24.Е8	Діаграми послідовностей			
			системи віртуалізації	1		
6	А3	КвРІПЗ.2201119.01.24.Е8	Діаграма послідовності			
			формування візуального			
			дерева			

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ.2201119.01.24.ВД		
Виконав		Юрчук Р.В.		27.05	Інструмент візуального створення редакторів користувача для рушія Unity Відомість документів	Літ.	Арк. 1 / Аркушів 1
Керівник		Праворська Н. І.		27.05			
Рецензент						ХНУ, ІПЗ-22-1	
Н. контр.		Яшина О. М.		27.05			
Зав. каф.		Бедратюк Л.П.		27.05			

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧОК

API	–	Application User Interface (Прикладний програмний інтерфейс)
UI	–	User Interface (Користувацький інтерфейс)
IMGUI	–	Immediate Mode GUI (старіша система інтерфейсу користувача в Unity, яка повністю керується кодом)
UI Toolkit	–	сучасна система інтерфейсу користувача в Unity зі збереженням стану, яка наслідує підхід вебтехнологій
low-code	–	підхід до розробки програмного забезпечення, що базується на візуальному проєктуванні з мінімальним написанням ручного коду
no-code	–	підхід до розробки програмного забезпечення, що базується на візуальному проєктуванні без написанням жодного коду
visual scripting	–	метод створення логіки програмного забезпечення за допомогою графічного інтерфейсу, що базується на вузлах та зв'язках між ними, замість написання текстового коду
геймдизайнер	–	спеціаліст, який розробляє концепцію, правила, механіки та ігровий процес, керуючи створенням гри від ідеї до релізу
відмальовувач	–	частина програмного коду, що бере на себе відповідальність щодо створення користувацького інтерфейсу для роботи з певним типом даних
інспектор	–	вікно редактора Unity, призначене для перегляду та маніпуляції компонентами або об’єктами даних

ВСТУП

Сучасна індустрія розробки програмного забезпечення, зокрема сфера створення комп'ютерних ігор та інтерактивних застосунків, характеризується стрімким зростанням складності проєктів та підвищенням вимог до швидкості розробки. Одним із ключових факторів успішної реалізації таких проєктів є використання просунутих інструментальних засобів – ігрових рушіїв, серед яких провідне місце посідає Unity. Його популярність зумовлена гнучкістю, кросплатформеністю та розгалуженою екосистемою, що дозволяє вирішувати широкий спектр завдань інженерії програмного забезпечення.

Проте, разом із розширенням можливостей рушія, зростає і потреба у створенні спеціалізованих внутрішніх інструментів розробки – так званих редакторів користувача (Custom Editors). Такі інструменти дозволяють геймдизайнерам та програмістам ефективніше керувати ігровими даними, налаштовувати складні компоненти та автоматизувати рутинні процеси. Оцінка сучасного стану розробки таких засобів у середовищі Unity вказує на наявність певних технологічних бар'єрів, що сповільнюють загальний темп створення програмних продуктів.

Традиційний підхід до створення розширень редактора Unity передбачає написання значної кількості коду з використанням API IMGUI (Immediate Mode GUI) або сучаснішого UI Toolkit. Основним недоліком такого підходу є тривалий ітераційний цикл: будь-яка зміна в структурі інтерфейсу чи логіці інструмента вимагає рекомпіляції всього проєкту. Це не лише призводить до нераціональних витрат часу розробників, а й створює високий поріг входження для фахівців, які не мають глибоких знань API редактора Unity.

Актуальність теми кваліфікаційної роботи визначається об’єктивною потребою галузі в модернізації процесів розробки інструментарію. Створення програмного засобу для візуального конструювання редакторів дозволить усунути необхідність постійної рекомпіляції при налаштуванні інтерфейсів та

забезпечить наочність процесу проєктування шляхом додавання інших функцій, орієнтованих на покращення досвіду розробників. Такий підхід відповідає сучасним тенденціям розвитку no-code (low-code) та visual scripting у інструментальних рішеннях, що спрямовані на підвищення ефективності процесів життєвого циклу програмного забезпечення.

Обґрунтованість необхідності розв'язання цієї задачі також підкріплюється комерційною привабливістю скорочення витрат на розробку внутрішнього інструментарію. Впровадження інструменту візуального створення редакторів дозволить спеціалістам фокусуватися на логіці функціонування продукту, а не на технічних деталях відмальовування інтерфейсу.

Метою кваліфікаційної роботи є розроблення інструментального засобу для візуального створення користувацьких редакторів у середовищі Unity, що дозволяє підвищити ефективність вказаного процесу шляхом автоматизації генерації інтерфейсів та усунення необхідності рекомпіляції коду під час їхнього налаштування, а також знижує поріг необхідних знань для початку роботи.

Для досягнення поставленої мети та отримання функціонально придатного програмного продукту було визначено наступні завдання:

- виконати аналіз предметної області, вивчити сучасні стандарти, тенденції та методики створення розширень редактора Unity;
- проаналізувати наявні програмні аналоги та інструментальні рішення, виявити їхні переваги та недоліки в контексті швидкості ітераційної розробки;
- встановити особливості та сформулювати перелік функціональних і нефункціональних вимог до програмного забезпечення, що забезпечить візуальну збірку інтерфейсів;
- розробити архітектуру та структуру інструменту, що дозволить інтегрувати систему візуального конструювання в середовище Unity;

- розробити моделі та алгоритми функціонування візуального конструктора, включаючи механізми миттєвого відображення змін без рекомпіляції проєкту;
- виконати програмну реалізацію розробленого засобу з використанням мови програмування C# та API Unity;
- провести тестування розробленого програмного продукту для підтвердження його працездатності та відповідності описаним вимогам.

Об’єктом дослідження є процеси розширення функціональних можливостей редактора Unity та розробляння внутрішнього інструментарію. Предметом дослідження є методи та засоби візуальної генерації користувацьких інтерфейсів для компонентів Unity без написання коду, no-code підхід.

Практична значущість отриманих результатів полягає у створенні завершеного програмного продукту, який дозволив би геймдизайнерам та розробникам інструментів створювати складні редактори Unity в інтуїтивно зрозумілому візуальному середовищі. Впровадження цього інструменту забезпечить значне скорочення ітераційного циклу розробки, знизить ресурсомісткість процесу та дозволить уникати типових помилок, що виникають при ручному програмуванні інтерфейсів.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ПРОЄКТУ

1.1 Аналіз процесів розширення функціональних можливостей редактора Unity та специфіки створення користувацьких інструментів

Предметна область дослідження охоплює розробку допоміжного інструментарію в середовищі ігрового рушія Unity для оптимізації робочих процесів. Сучасна інженерія програмного забезпечення в індустрії розробки ігор демонструє, що ефективність створення продукту напряму залежить від якості внутрішніх інструментів, які дозволяють автоматизувати рутинні операції та візуалізувати складні структури даних. Основними об'єктами маніпуляції в редакторі Unity є класи MonoBehaviour для логіки об'єктів та ScriptableObject для збереження даних, які відображаються через механізм, побудований на основі серіалізації Unity [1].

Для редагування цих об'єктів Unity використовує вікно інспектора, яке автоматично генерує графічний інтерфейс на основі серіалізованих полів класу. Проте стандартний механізм має обмеження: він не підтримує динамічну зміну інтерфейсу залежно від логічних умов та не дозволяє хоч якось змінювати візуалізацію без написання додаткового коду.

На рисунку 1.1 зображено стандартний цикл серіалізації даних, де критичною ланкою є жорстка прив'язка графічного представлення до структури коду. Будь-яка зміна у відображенні вимагає безпосереднього втручання в архітектуру скрипта, що ускладнює підтримку великих проєктів.

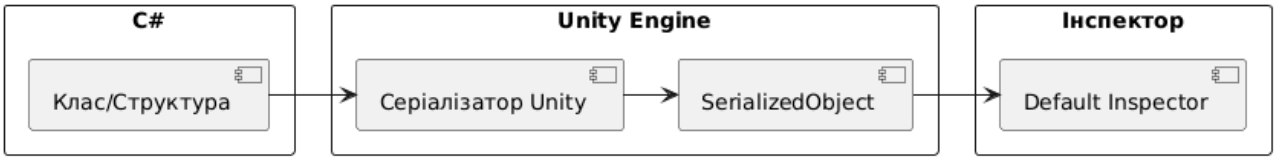


Рисунок 1.1 - Схема стандартного процесу серіалізації

Згідно з аналізом 2024 року, за останні роки акцент досліджень у галузі розробки ігор змістився із специфікацій (Requirements/Specifications) та інструментів для коду (Coding Tools and Techniques) до інструментів та технік дизайну (Design Tools and Techniques), які до 2009 взагалі не бралися до уваги [2]. Це підкреслює важливість вдосконалення взаємодії розробника із ігровим рушієм, в тому числі вдосконалення інтерфейсів редагування даних. Впровадження візуальних середовищ для конструювання таких інтерфейсів є частиною загального тренду розробки no-code (low-code) та visual scripting, що дозволяє фахівцям без глибоких знань програмування самостійно налаштовувати свій робочий простір.

Функціональна еволюція засобів побудови інтерфейсів у Unity призвела до співіснування трьох парадигм: імперативної (IMGUI), об'єктно-орієнтованої (uGUI) та декларативної (UI Toolkit) [3]. Парадигма uGUI не може бути використана для реалізації інтерфейсів редактора, оскільки вона базується на ієрархії ігрових об'єктів, тому вона не братиметься до уваги у цьому дослідженні як варіант вирішення проблеми. Система IMGUI, хоча й залишається стандартом для швидкого прототипування, через свою процедурну природу, створює значне навантаження на центральний процесор, оскільки потребує повного перемальовування інтерфейсу кожного кадру [4]. Натомість сучасний фреймворк UI Toolkit базується на збереженні структури та станів у вигляді об'єктів і використовує стандарти UXML та USS, що дозволяє відокремити структуру від стилістики [5].

Згідно з порівняльними дослідженнями 2025 року, використання UI Toolkit для складних систем забезпечує до 9 разів меншу кількість викликів відмальовування та у 3 рази швидшу обробку логіки на боці процесора порівняно з традиційними методам uGUI, який також використовує зберігання стану [6]. Окрім цього UI Toolkit має оптимізовані рішення для відмальовки списків та переліків. Попри ці переваги, як UI Toolkit, так і IMGUI обмежені архітектурною особливістю рушія – необхідністю повного перезавантаження при будь-якій

зміні коду. UI Toolkit може оновлювати структуру в реальному часі, проте початкове створення шаблону та його прив’язка до компонента чи об’єкта даних все ж потребуватимуть рекомпіляції рішення.

На рисунку 1.2 наведено структурну різницю між системами. IMGUI жорстко пов'язує логіку з відмальовуванням, натомість UI Toolkit пропонує об'єктну модель документа, подібну до вебтехнологій. Проте етап рекомпіляції скриптів все ще залишається головним вузьким місцем, що перериває робочий процес розробника на час від 3 до 60 секунд залежно від масштабу проєкту. Також варто врахувати, що факт необхідності створення скрипту для прив’язки редактора до відповідного типу об’єктів в UI Toolkit потребує базових знань у програмуванні, що суперечить ідеї редактора без необхідності втручання в код.

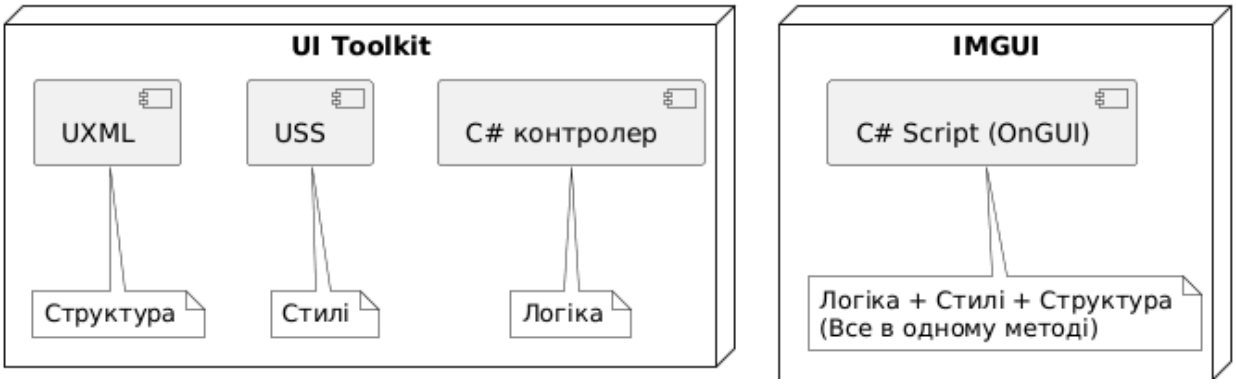


Рисунок 1.2 – Архітектурні підходи UI Toolkit та IMGUI

Аналіз предметної області дав змогу виокремити основні групи користувачів, на яких орієнтовано розроблюване програмне забезпечення: інженери та нетехнічні спеціалісти. Інженери потребують засобів для мінімізації рутинного написання коду для візуалізації даних, тоді як нетехнічним спеціалістам необхідна можливість прямого керування параметрами системи та адаптації її вигляду під себе без посередництва розробників.

Для будь-якого рішення з візуального редагування необхідно мати підтримку використання даних пов’язаного об’єкта. UI Toolkit реалізує механізм прив’язки даних [7]. Цей механізм має певні обмеження. По-перше, він дозволяє

прив’язувати лише ті дані, які відповідають певним критеріям: є серіалізованими або мають спеціальний атрибут у скрипті. По-друге, цей механізм не дозволяє опрацьовувати дані об’єкта перед їх використанням, наприклад, виконувати математичні операції, конкатенацію рядків, обчислення логічних виразів тощо. Це значно зменшує гнучкість при створенні інтерфейсу із візуального редактора. Описана логіка потребуватиме втручання у код, що унеможливлює її реалізацію нетехнічними спеціалістами.

Важливим аспектом проєктування таких систем є забезпечення цілісності та безпеки даних. Оскільки інструмент працює безпосередньо із серіалізованими даними Unity, критично важливим є впровадження механізмів валідації типів на етапі візуального зв'язування. Такий підхід дозволить уникнути помилок і пошкодження об’єктів та забезпечить відповідність введених значень вимогам програмної логіки, щоб гарантувати якість програмного засобу [8].

Підсумовуючи змістовий аналіз, можна дійти висновку, що наявні вбудовані методи розробки редакторів Unity мають суттєві обмеження у швидкості ітерацій через залежність від компіляції та високий поріг входження. Виявлена потреба у створенні інструменту, який би поєднував потенціал UI Toolkit із гнучкістю візуального моделювання структури та логіки, дозволяючи створювати складні інтерфейси без статичного написання коду. Це забезпечить не лише прискорення розробки, а й покращить ергономіку внутрішнього інструментарію проєктів.

1.2 Аналіз наявного програмно-технічного забезпечення заданої предметної області

Сучасний ринок інструментарію для ігрового рушія Unity пропонує широкий спектр рішень, спрямованих на розширення можливостей стандартного інспектора та створення власних редакторів. Аналіз цих засобів дозволив

класифікувати їх за методом взаємодії розробника з інтерфейсом на дві основні категорії: кодові фреймворки на основі атрибутів та візуальні конструктори. Основною метою даного огляду є оцінка функціональності існуючих систем у контексті швидкості ітераційного циклу, підтримки візуальної логіки та можливостей безшовної інтеграції з вбудованими компонентами Unity.

1.2.1 Фреймворки на основі атрибутів

Спочатку були розглянуті декларативні фреймворки на основі атрибутів (Odin Inspector, Alchemy, artificetoolkit). Дана група інструментів базується на використанні метаданих мови C# (атрибутів) для генерації інтерфейсів без необхідності написання окремих класів редактора. Найбільш поширеним комерційним рішенням у цьому сегменті є Odin Inspector (розробник Sirenix, Данія, 2017–2025), що фактично став стандартом індустрії для великих студій [9]. Поряд із ним існують сучасні open-source альтернативи, такі як Alchemy та Artifice, які пропонують подібний, а в деяких випадках навіть унікальний функціонал у межах відкритого доступу [10, 11].

Odin Inspector пропонує понад 100 атрибутів для зміни вигляду полів (групування, валідація, умовне відображення). Безкоштовні аналоги забезпечують схожий функціонал, проте часто мають менший набір атрибутів.

Створення інтерфейсу за допомогою цих систем відбувається виключно через програмний код. Розробник маркує поля класу спеціальними атрибутами, після чого механізм рефлексії інтерпретує ці позначки та відмальовує відповідні елементи UI у інспекторі.

На рисунку 1.3 продемонстровано типовий приклад робочого процесу: для додавання кнопки або групи вкладок розробник змушений вносити зміни безпосередньо у вихідний код компонента. Кожна така зміна ініціює цикл рекомпіляції скриптів проєкту, що є основним часовим бар’єром у розробці.

```
[SerializeField, HorizontalGroup("horizontal1"), VerticalGroup("horizontal1/vertical")]
private List<int> leftColumn;

[SerializeField, HorizontalGroup("horizontal1"), VerticalGroup("horizontal1/vertical")]
private int leftColumnInteger;

[SerializeField, HorizontalGroup("horizontal1")]
private List<int> rightColumn;
```

Рисунок 1.3 – Типова реалізація інтерфейсу у artificetoolkit

Переваги:

- надзвичайна гнучкість для програмістів;
- зазвичай присутня підтримка відображення несеріалізованих членів;
- зазвичай присутня підтримка серіалізації типів, які Unity не підтримує за замовчуванням (Dictionary, Generic-класи);
- швидке створення простих кнопок налагодження.

Недоліки:

- жорстка прив’язка до коду;
- тривале очікування рекомпіляції;
- неможливість залучення фахівців без знань C# до проєктування інтерфейсів редактору.

Попри свою функціональність, атрибутивні системи не забезпечують візуальної наочності. У них відсутня можливість побудови умовної логіки через графи. Замість цього використовуються текстові рядки в атрибутах, в яких легко

можна допустити помилку через відсутність перевірки виразів в режимі реального часу. Також ці рішення не мають механізму глобального перепризначення редактора для класу без модифікації скриптів.

1.2.2 Odin Inspector Visual Designer

Далі було розглянуто Odin Visual Designer, що є розширенням раніше описаного Odin Inspector, випущеним як відповідь на потребу у візуалізації процесу створення складних інструментів [9]. На відміну від базової системи атрибутів, цей модуль пропонує графічний інтерфейс для компонування елементів редактора.

Інструмент глибоко інтегрований у пропрієтарну систему серіалізації Odin, що дозволяє працювати з типами даних, які недоступні стандартному інспектору Unity. Проєктування відбувається у спеціалізованому вікні Visual Designer, де розробник створює ієрархію елементів (груп, полів, кнопок) шляхом вибору зі списку та налаштування їхніх властивостей через панель параметрів.

На рисунку 1.4 показано інтерфейс конструктора, де структура майбутнього інспектора представлена у вигляді впорядкованих блоків. Це дозволяє бачити структуру безпосередньо під час її створення. Редактор дозволяє перетягувати блоки елементів для змінення їхнього порядку у ієрархії. Для кожного поля можна додати та налаштувати атрибути відображення із колекції Odin, натиснувши декілька кнопок. Аналогічно до базового Odin Inspector, у конструкторі можна візуалізувати несеріалізовані типи, включно із приватними та статичними членами.

Переваги:

- висока стабільність;
- підтримка величезної кількості вбудованих атрибутів (із Odin Inspector);
- підтримка візуалізації несеріалізованих членів;

– можливість створювати вікна редактору (Editor Windows).

Недоліки:

– жорстка залежність від платного плагіна Odin;
– складність у розширенні базового функціоналу конструктора користувачем, доступ до вихідного коду потребує значної доплати.

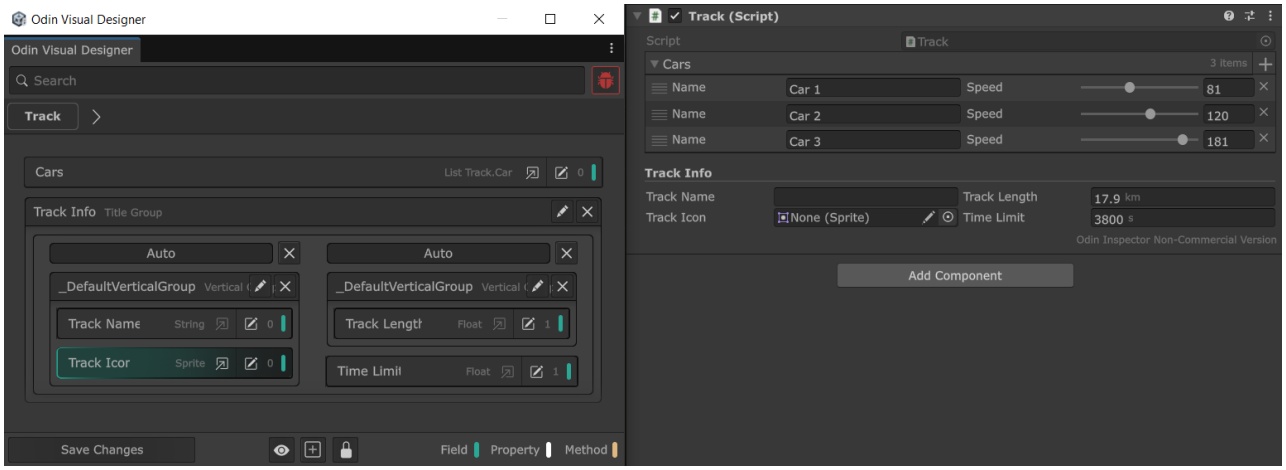


Рисунок 1.4 – Типова реалізація інтерфейсу у Odin Visual Designer

Odin Visual Designer, попри свою назву, все ще орієнтований на лінійне відображення. У ньому відсутня повноцінна система для візуального моделювання розгалуженої умовної логіки та прив'язка даних до характеристик елементів керування.

1.2.3 Vibe: No-Code Custom Inspector Builder & Editor

Vibe – це спеціалізований програмний засіб, доступний у Unity Asset Store (розробник Chisely). Його основна ідея полягає у наданні no-code досвіду, дозволяючи змінювати вигляд будь-якого компонента без створення C# скриптів [12]. Інструмент фокусується на ергономіці та швидкості налаштування.

Як видно з рисунку 1.5, процес розробки максимально спрощений. Редактор має вигляд звичної ієрархії з можливістю згортання, на відміну від Odin

Visual Designer, який дозволяв створювати горизонтальні групи, розміщуючи елементи один біля одного. Користувач відкриває вікно конструктора Vibe, обирає цільовий скрипт і за допомогою графічних блоків визначає, які поля та в якому порядку будуть відображатися в інспекторі.

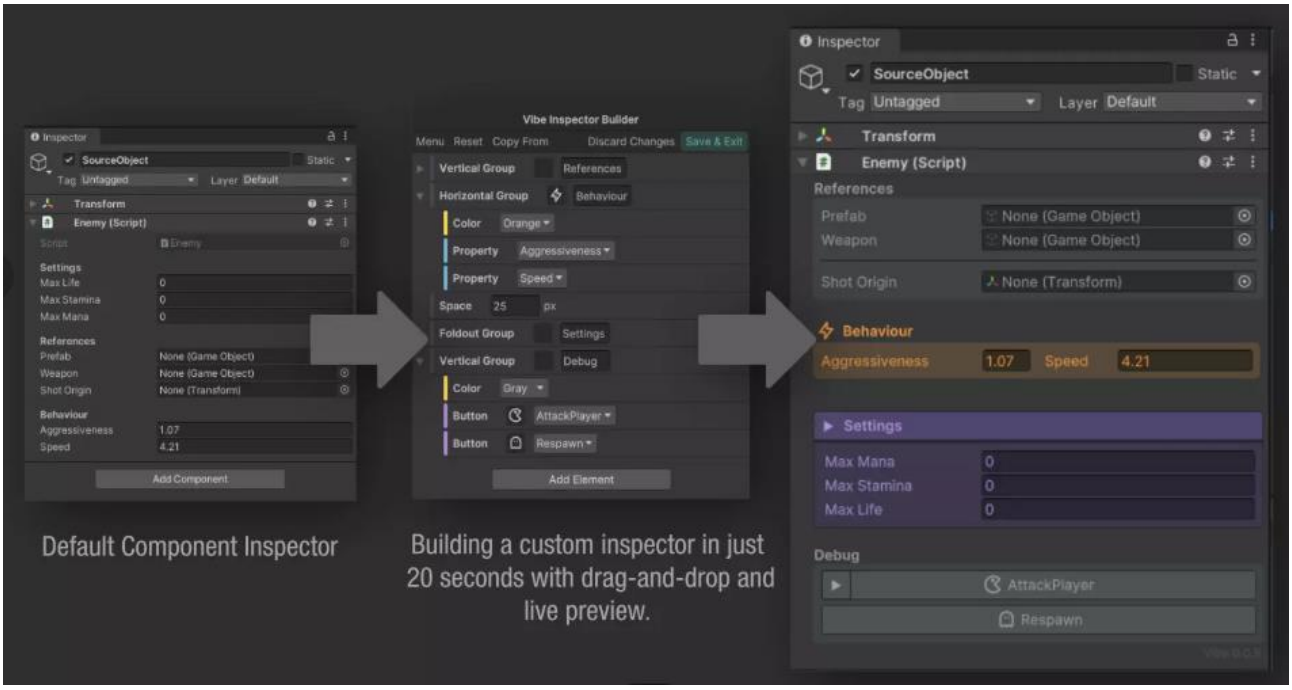


Рисунок 1.5 – Типова реалізація інтерфейсу у Vibe: No-Code

Переваги:

– найнижчий поріг входження серед аналогів;
– зручні інструменти декорації інспектора (кольори, іконки, групи);
– інтуїтивна ієрархія конструктора.

Недоліки:

– значно менший набір атрибутів для полів;
– не підтримує серіалізацію типів даних, що не підтримуються Unity;
– тривала відсутність оновлень.

Загалом Vibe є більш примітивним варіантом Odin Visual Designer. Він не пропонує унікальних функцій, проте вирізняється зручним та інтуїтивно зрозумілим інтерфейсом.

1.2.4 Вбудоване рішення Unity: UI Builder (UI Toolkit)

UI Builder – це офіційний візуальний редактор від компанії Unity Technologies, що є частиною фреймворку UI Toolkit [13]. Він був розроблений для заміни застарілих систем IMGUI та uGUI, пропонуючи розробникам підхід, запозичений із сучасних вебтехнологій.

Інструмент повністю безкоштовний і постачається у складі рушія Unity (починаючи з версії 2021.3 LTS та новіших). Основною особливістю є використання декларативних форматів UXML (для структури) та USS (для стилізації), що дозволяє відокремити візуальне представлення від логіки. Створення інтерфейсу відбувається в окремому вікні, де користувач маніпулює ієрархічним деревом елементів. Процес включає перетягування стандартних блоків та налаштування їхніх параметрів.

На рисунку 1.6 представлено робоче середовище UI Builder. Хоча інструмент надає багато можливостей для верстки, він вимагає від розробника ручного створення логіки через C# скрипти для кожного окремого інспектора, якщо вона виходить за рамки прямого зв’язування даних із полями.

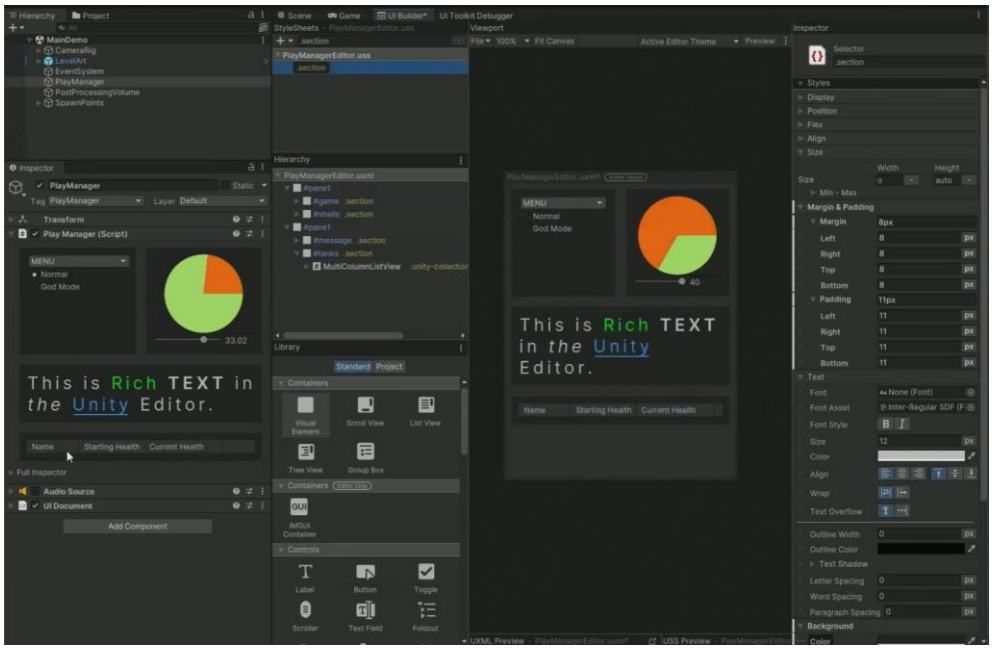


Рисунок 1.6 – Типова реалізація інтерфейсу у UI Builder

Переваги:

- висока продуктивність;
- відповідність стандартам Unity;
- вбудований інструмент відлагодження створених інтерфейсів;
- вбудована підтримка векторної графіки та складних анімацій.

Недоліки:

- високий поріг входження;
- заплутаний інтерфейс;
- необхідність у створенні мінімального скрипта для прив’язки.

UI Builder є загальним інструментом верстки, а не спеціалізованим конструктором інспекторів. Реалізація умовної логіки відображення (наприклад, приховування полів) потребує програмування, оскільки у системі не передбачено жодного механізму для обробки виразів.

1.2.5 Порівняльна характеристика та висновки

Аналіз показав, що сучасні інструменти мають сильні сторони: Odin забезпечує відмінну роботу з даними та широкий набір атрибутів, Vibe – простоту для нетехнічних спеціалістів, а UI Builder – технологічну досконаліть верстки. Однак було виявлено значну проблему у вказаних рішеннях, жодне із них не пропонує:

- інтегрованої системи, де б умовна чи математична логіка будувалася візуально (наприклад, через систему графів);
- паралельне створення та використання декількох макетів, з можливістю оперативного перемикання між ними;
- можливості вибору та використання уже визначених в проєкті відмальовувачів для полів;
- передачі обробки відмальовування іншому встановленому фреймворку.

1.3 Визначення функціональних та нефункціональних вимог до програмного забезпечення

На основі проведеного аналізу предметної області та виявлених недоліків існуючих рішень, були сформовані вимоги до розроблюваного програмного забезпечення. Вимоги поділено на функціональні, що описують можливості системи, та нефункціональні, які визначають технічні обмеження та якісні характеристики продукту. Такий підхід дозволяє чітко окреслити межі проєктування та забезпечити відповідність результату поставленій меті.

Функціональні вимоги до системи охоплюють можливості візуального проєктування та керування даними. Основним завданням є реалізація візуального конструктора, який дозволяє формувати структуру інспектора та його візуальне дерево елементів без написання програмного коду.

Для забезпечення ефективної роботи розробника система повинна відповідати наступним функціональним критеріям:

а) автоматизація побудови інтерфейсу, шляхом створення та редагування ієрархії компонентів інспектора за допомогою візуального конструктора;

б) механізм макетів, що керуються даними, реалізований на основі підходу збереження структури редактора в окремі файли конфігурації, які зчитуються в режимі реального часу;

в) можливість додавання атрибутів шаблонів та декораторів до полів;

г) можливість створення груп, панелей, вкладок;

д) гнучке керування джерелами даних для атрибутів, для кожного поля в редакторі розробник повинен мати можливість обрати один із трьох режимів отримання інформації:

1) ручне введення шляхом задання статичних значень параметрів безпосередньо в конструкторі;

2) об’єктна прив’язка шляхом автоматичного підтягування значень із членів цільового об’єкта;

3) візуальний граф логіки, що формуватиме динамічні значення через вузлову систему.

Особлива увага приділяється роботі з візуальним графом логіки. Він має забезпечувати обробку складних даних об’єктів для отримання примітивних типів у результаті, зокрема:

- цілих та дійсних чисел (int, float);
- рядкових значень (string);
- логічного типу (bool).

Використання графа дозволить реалізовувати складну логіку, таку як математичні операції над параметрами, конкатенація рядків для виведення інформаційних повідомлень та умовне відображення певних елементів чи цілих груп залежно від стану об’єкта чи об’єктів.

Окрім роботи зі стандартними властивостями та групами, програмний засіб повинен забезпечувати:

а) візуалізацію несеріалізованих типів:
   1) відображення та редагування приватних полів;
   2) відображення та редагування статичних членів класу;
   3) відображення обчислювальних властивостей чи методів;

б) підтримку інших рішень та відмальовувачів:
   1) підтримка перегляду та вибору скриптових відмальовувачів серед наявних у контексті проєкту;
   2) створення спеціалізованого «містка», що дозволятиме вбудовувати стандартні відмальовувачі старого зразка у нову систему;
   3) можливість примусової відмальовки базового інспектору Unity, навіть при наявності спеціально визначеного макету для типу;
   4) підтримка перемикання на інші фреймворки для певних класів.

Визначені у таблиці 1.1 нефункціональні вимоги вказують на технічні обмеження, експлуатаційні характеристики та критерії якості, яким повинен відповідати продукт для забезпечення стабільної роботи в середовищі Unity.

Таблиця 1.1 – Аспекти реалізації для нефункціональних вимог

Тип вимоги	Аспекти реалізації
Сумісність	підтримка версій ігрового рушія Unity 6000.3 LTS та вище
Продуктивність	використання UI Toolkit для мінімізації навантаження та забезпечення плавності інтерфейсу під час проєктування; миттєве оновлення шаблону (максимум 300 мс.); кешування результатів повторюваних складних обчислень
Надійність	реалізація механізмів автоматичної валідації типів даних; автоматичне відловлювання помилок виконання та показ локальних супровідних повідомлень для користувача; система не має створювати помилок на рівні всього редактору Unity, навіть у найгіршому випадку; механізми допомоги у відновленні при втраті посилань, наприклад через перейменування скриптів чи їхніх змінних
Зручність використання	візуальне оформлення конструктора має поєднуватися із загальною стилістикою редактора Unity; підтримка стандартної системи скасування та повторення дій користувача; відсутність «унікальних» рішень у інтерфейсі, неочевидних для користувача, засіб має бути інтуїтивно зрозумілим

Для моделювання функціональних можливостей системи та ролей користувачів було розроблено діаграму варіантів використання, яка наочно показує, які саме функції програмний засіб надаватиме акторам (рисунок 1.7).

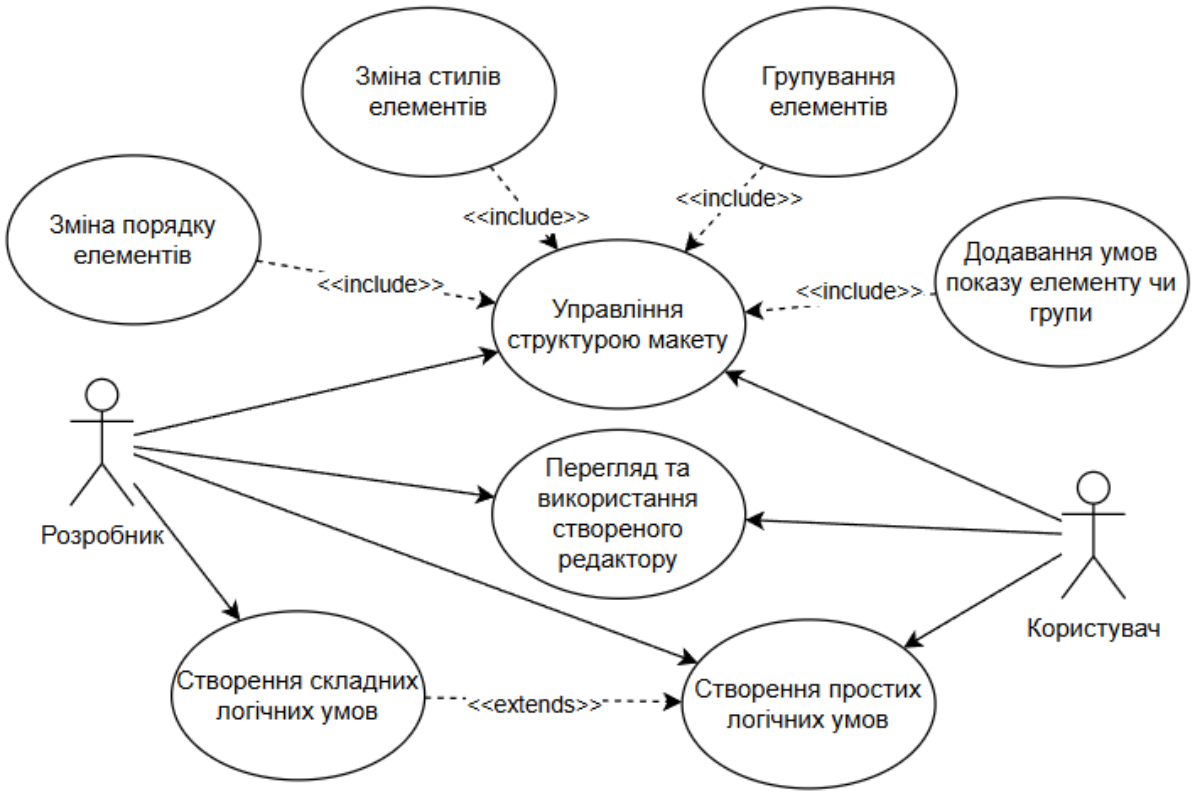


Рисунок 1.7 — Діаграма варіантів використання програмного забезпечення

На діаграмі виокремлено двох основних акторів:

- користувач, завданням якого є перегляд та безпосереднє використання створеного редактора для перегляду та маніпуляції параметрами ігрових компонентів, окрім цього, він може здійснювати безпосереднє управління структурою макета, включаючи зміну стилів, групування елементів та налаштування умов відображення чи приховування, додаючи відповідну примітивну логіку;
- розробник, який має розуміння внутрішньої будови проєкту та досвід у програмуванні, відповідає за створення логічних умов різної складності, є необов'язковою ланкою у роботі системи.

Для пояснення взаємодії між даними та сутностями було розроблено модель даних процесів передавання та оброблення інформації в межах системи у вигляді діаграми потоків даних (рисунок 1.8).

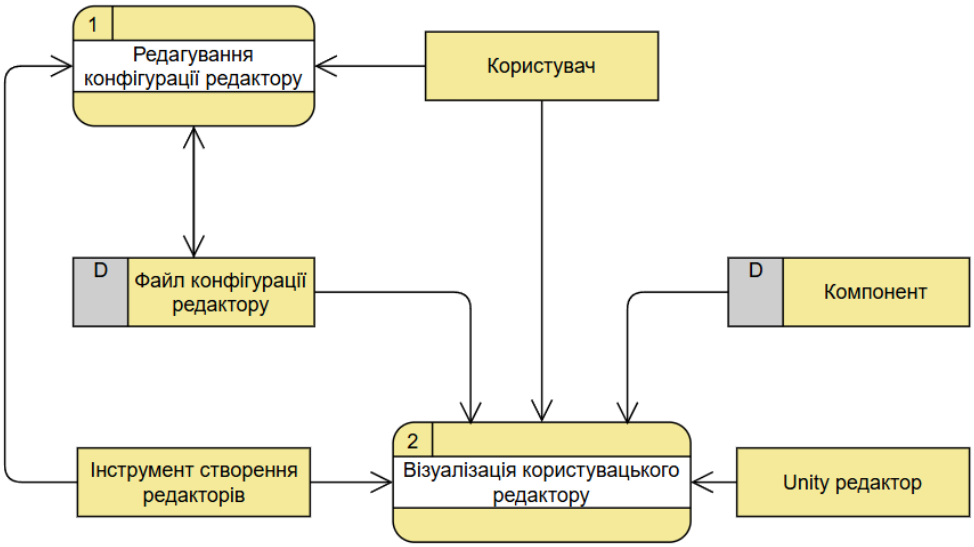


Рисунок 1.8 — Діаграма потоків даних системи

Згідно з представленою моделлю, процес починається з редагування файлу конфігурації. Ці дані в режимі реального часу перетворюються у формат, що може застосовуватися для відмальовування, та разом із параметрами конкретного компонента й станом Unity редактора, надходять до блоку візуалізації, який формує кінцевий інтерфейс для користувача.

На основі проведеного аналізу та визначених вимог було розроблено технічне завдання, яке можна переглянути у додатку А.

1.4 Висновки. Постановка задачі

У межах цього розділу було проведено комплексне дослідження предметної області розробки інструментів для Unity. Результати аналізу підтвердили актуальність створення засобу візуального проєктування редакторів, що спростить додавання логіки, дозволить усунути потребу в рекомпіляції скриптів та знизить поріг входження для нетехнічних фахівців.

На основі вивчення наявного програмно-технічного забезпечення (Odin Inspector, Vibe, UI Builder) було виявлено відсутність рішень, які б поєднували

вузлову систему моделювання логіки з можливістю використання макетів, що керуються даними, та функціоналом динамічного перемикання між способами відмальовування, включно із можливістю використання декількох різних шаблонів для одно й того ж типу в межах одного інспектора. Визначені функціональні вимоги, зокрема система динамічного вибору джерела даних із трьома режимами (ручний, об'єктний, графовий) та підтримка несеріалізованих і статичних членів класу, стали підґрунтям для подальшого проєктування архітектури системи.

Враховуючи виявлені проблеми та сформовані вимоги, метою кваліфікаційної роботи було встановлено розроблення інструментального засобу візуального створення редакторів для Unity на базі фреймворку UI Toolkit, який забезпечуватиме зміну логіки та структури інтерфейсу інспектора без необхідності рекомпіляції коду.

Для досягнення мети необхідно вирішити наступні задачі на етапах проєктування, реалізації та тестування:

- спроєктувати архітектуру системи, яка дозволить відокремити опис логіки інтерфейсу від статичного коду компонентів, використовуючи шаблони проєктування для забезпечення розширюваності;
- розробити структуру даних для зберігання візуальних макетів та графів логіки у форматі, що підтримує швидке динамічне зчитування в режимі реального часу;
- спроєктувати вузловий редактор логіки для отримання результатів у вигляді примітивних типів даних, на основі об’єкту в поточному контексті;
- створити механізм «містка» для інтеграції стандартних та визначених в проєкті відмальовувачів, включно із іншими фреймворками, в нову систему, забезпечуючи зворотню сумісність;
- виконати програмну реалізацію конструктора та провести серію тестів для перевірки відповідності функціональним та нефункціональним вимогам.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Вибір типу архітектури та шаблонів проєктування

Для вибору доречної архітектури розроблюваного програмного забезпечення було проаналізовано підходи до побудови графічних інтерфейсів користувача. Оскільки візуальну підсистему буде побудовано на базі фреймворку UI Toolkit, що функціонує в режимі збереження стану (Retained Mode), вибір базової архітектури вимагав врахування механізмів обробки подій та життєвого циклу елементів [14].

У процесі проєктування було розглянуто кілька архітектурних шаблонів, зокрема MVC (Model-View-Controller) та MVP (Model-View-Presenter). У парадигмі MVC контролер безпосередньо маніпулює станом відображення, що в умовах використання UI Toolkit призводить до високої зв'язності коду та необхідності ручного оновлення кожного елемента візуального дерева, тому від використання класичного шаблону MVC було вирішено відмовитися [15]. Шаблон MVP, хоча і забезпечує кращу ізоляцію через використання інтерфейсів, вимагає створення значної кількості проміжного коду для зв'язування даних. Це ускладнює динамічну генерацію інтерфейсу, оскільки структура редактора не є статичною, а формується під час виконання на основі макетів [16].

Оптимальним архітектурним рішенням було визначено шаблон MVVM (Model-View-ViewModel) [17]. Його вибір обґрунтовується наявністю в UI Toolkit вбудованої підсистеми прив'язки даних, яка ідеально інтегрується з концепцією ViewModel. Відповідно до обраного шаблону, архітектуру програмного засобу було розподілено на три ключові рівні.

Model (Модель) – рівень зберігання даних. Включає цільові об’єкти користувача та конфігураційні файли. Конфігурації реалізовано як макети, що керуються даними, які зберігають структуру візуального редактора.

View (Вид) – рівень представлення. Формується динамічно під час виконання як дерево елементів, згенероване на основі конфігураційних макетів.

ViewModel (Модель представлення) – рівень проміжного кешування та синхронізації. Виконує роль посередника, який зчитує дані будови з об'єкта, формує адаптовані структури та забезпечує двосторонню прив’язку даних із Видом, відокремлюючи логіку відмальовування від прямої взаємодії з Моделлю.

Описану архітектуру у контексті розроблюваної програмної системи було показано на рисунку 2.1.

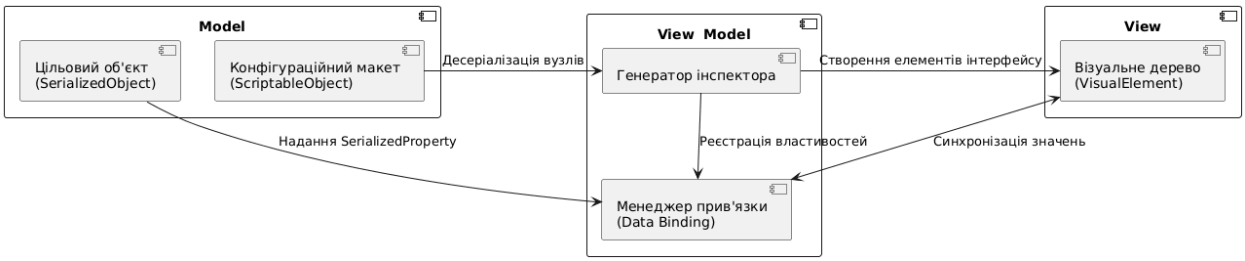


Рисунок 2.1 – Діаграма компонентів «MVVM у контексті системи»

Для вирішення специфічних інженерних завдань та забезпечення гнучкості взаємодії між компонентами системи, окрім базової архітектури MVVM, було застосовано низку допоміжних шаблонів проєктування [18, 19].

«Фабричний метод» (Factory Method) було використано для проєктування динамічної генерації вузлів візуального дерева. Оскільки макети, що керуються даними, зберігатимуть структуру майбутнього редактора у вигляді списку абстрактних інструкцій (поліморфних об’єктів), безпосереднє створення графічних елементів потребує відокремленої логіки. Спеціалізований клас-фабрика братиме на себе відповідальність за створення екземплярів конкретних графічних компонентів (контейнерів, полів вводу, міток тощо) залежно від типу вузла. Таке рішення дозволить масштабувати програмний засіб, додаючи нові типи візуальних блоків без необхідності модифікації основного коду генератора.

«Проксі» (Proxy) є фундаментальним архітектурним рішенням для функціонування модуля віртуалізації несеріалізованих даних. Стандартні механізми рушія Unity, зокрема відмальовувачі властивостей (Property Drawers) та підсистема скасування та повторення дій (Undo/Redo), здатні працювати

виключно з серіалізованими полями [20, 21]. Для подолання цього обмеження було спроєктовано механізм динамічного обгортання несеріалізованих змінних у тимчасові серіалізовані класи-посередники (сесійні проксі-об'єкти). Цей підхід забезпечуватиме прозору взаємодію інтерфейсу з даними цільового об'єкта, повністю імітуючи стандартну поведінку середовища розробки без необхідності втручання в ядро серіалізатора.

Ще одним важливим архітектурним рішенням є застосування шаблону «Стратегія» (Strategy) для реалізації підсистеми перехоплення та відображення інспекторів [18]. Оскільки розроблюваний інструмент повинен підтримувати зворотну сумісність з існуючими компонентами та стандартними редакторами Unity, виникла об’єктивна потреба у динамічному виборі алгоритму відмальовування інтерфейсу під час виконання. Застосування цього шаблону дозволяє системі інкапсулювати різні підходи до відображення та автоматично перемикатися між ними залежно від конфігурації цільового об'єкта. Було визначено три основні стратегії для відмальовування:

- генерація візуального дерева елементів на основі створеного користувачем конфігураційного макета;
- делегування відмальовування існуючому користувацькому редактору на основі коду, якщо такий був створений розробниками раніше;
- виклик стандартного механізму відображення рушія Unity, ігноруючи будь-які перевизначення.

У підсумку, інтеграція шаблону MVVM із допоміжними шаблонами проєктування сформувала надійну основу для програмного засобу. Такий комплексний підхід забезпечує необхідну гнучкість для конструювання макетів, що керуються даними, гарантує високу швидкодію завдяки використанню сучасного фреймворку UI Toolkit та дозволяє реалізувати складні механізми обробки інформації. Найважливішим досягненням обраної архітектури є повне усунення необхідності ініціації небажаного процесу рекомпіляції коду під час редагування інтерфейсів.

2.2 Детальне проєктування структур даних та конфігураційних макетів

Специфіка розроблюваного інструментального засобу виключає необхідність інтеграції класичних систем керування базами даних на етапі конфігурування інтерфейсів. Для забезпечення максимальної продуктивності та сумісності з екосистемою рушія Unity, базовим контейнером для збереження структури візуальних макетів було обрано вбудований клас ScriptableObject, об’єкти якого зберігаються на диску як окремі ресурси [22]. Застосування цього підходу гарантує безшовну інтеграцію конфігураційних файлів у внутрішній конвеєр обробки ресурсів, автоматизує процеси керування пам'яттю та забезпечує повноцінну підтримку системи скасування та повторення дій на рівні середовища розробки.

Фундаментом спроєктованої моделі даних є макети, що керуються даними. Кожен такий макет представляє собою збережену ієрархічну конфігурацію, яка виступає набором інструкцій для генератора під час динамічної побудови візуального дерева елементів.

Так як редактор вимагає збереження неоднорідних структурних блоків у межах єдиної конфігурації, було проаналізовано підходи до обробки поліморфних даних. Для вирішення цього завдання було застосовано механізм поліморфної серіалізації на базі атрибута SerializeReference [23]. Це архітектурне рішення дозволяє інкапсулювати масив об'єктів із базовим абстрактним інтерфейсом, де кожен елемент колекції може належати до відмінного класу-реалізації. Відповідно до вимог предметної області, усі структурні вузли макета класифіковано за такими логічними категоріями:

- вузли даних, що визначатимуть конфігурацію відмальовування властивостей, членів об’єкту та обрахованих значень;
- вузли-контейнери, що представлятимуть структурні елементи, призначені для логічного та візуального групування інших вузлів (вкладки, панелі згортання, блоки компонування);

– декоративні вузли, що відповідатимуть за візуальне оформлення інтерфейсу (текстові мітки, розділювачі).

Для розширення функціональності базових вузлів без надмірного збільшення глибини візуального дерева було застосовано підхід на основі композиції. Кожен абстрактний вузол макета може містити масив модифікаторів, які діють як додаткові конфігураційні інструкції для зміни зовнішнього вигляду або поведінки елемента (наприклад, перетворення числового поля на повзунок або зміна кольору фону). Відповідно до принципів шаблону MVVM, інтерфейси вузлів та модифікаторів спроєктовано виключно як пасивні структури даних, які не містять методів взаємодії, повністю делегуючи процес обробки спеціалізованому класу на рівні моделі представлення. У графічному документі «Діаграма класів схем даних та модифікаторів макету» представлено базовий набір елементів описаних схем, модифікаторів та їхніх обмежень.

Для забезпечення безпеки типів та уникнення помилок під час налаштування макетів було реалізовано декларативну систему обмежень, що функціонує за такими принципами:

– кожен вузол макета визначає власні можливості за допомогою побітового переліку, що дозволяє системі ідентифікувати підтримувані ним базові властивості та поведінку;
– класи модифікаторів використовують спеціальний атрибут для декларування жорстких вимог до цільового вузла та масив підтримуваних типів серіалізованих даних;
– візуальний редактор на етапі конструювання динамічно фільтрує доступні для вибору модифікатори на основі логічного зіставлення можливостей обраного вузла та вимог зазначеного атрибута.

Важливою складовою конфігураційних макетів є механізм збереження посилань на типи даних (класи, інтерфейси, структури). Оскільки стандартна система серіалізації рушія Unity не підтримує пряме збереження об'єктів System.Type, виникає необхідність у розробленні надійної архітектурної

обгортки. Використання базового текстового поля для збереження повного імені типу (AssemblyQualifiedName) є небезпечним: у випадку рефакторингу коду користувачем (зміни назви класу або простору імен) посилання автоматично руйнується, а візуальний редактор втрачає контекст помилки [24].

Для розв'язання цієї проблеми було спроєктовано спеціалізований клас SerializedType, який інкапсулює логіку безпечного збереження, відновлення та валідації типів. Цей клас реалізує базовий інтерфейс ISerializationCallbackReceiver, що дозволяє інтегрувати власну логіку в процеси збереження та завантаження даних рушієм [25].

Спроєктований механізм збереження посилань на типи характеризується такими функціональними особливостями:

- відкладене lazy-кешування при якому під час виконання програми об'єкт не ініціює пошук типу до моменту першого звернення, а після успішного знаходження типу за його текстовим ідентифікатором результат кешується в оперативній пам'яті, що знижує кількість ресурсомістких викликів рефлексії під час роботи інструмента [26];
- стійкість до втрати посилань у разі неможливості ідентифікувати тип, об'єкт не очищає збережений ідентифікатор, а фіксує внутрішній стан помилки, це рішення дозволяє інтерфейсу правильно проінформувати користувача про втрату конкретного посилання замість аварійного завершення генерації візуального дерева.

Незважаючи на високу ефективність та зручність використання ScriptableObject як основного контейнера збереження даних у середовищі Unity, перспективним напрямом подальшого вдосконалення системи було визначено розроблення додаткового механізму експорту та імпорту макетів у стандартизовані відкриті (наприклад, JSON) або власні текстові формати. Реалізація такого архітектурного розширення в майбутньому дозволить забезпечити низку суттєвих переваг:

- покращення сумісності та взаємодії із системами контролю версій завдяки прозорій та читабельній структурі текстових файлів;
- спрощення процесу виявлення та вирішення конфліктів злиття під час паралельної командної розробки інструментарію;
- забезпечення можливості ручного редагування або програмної генерації конфігураційних макетів у зовнішніх середовищах та текстових редакторах, використовуючи визначений синтаксис;
- полегшення процесу обміну створеними візуальними редакторами між різними проєктами без жорсткої прив'язки до внутрішніх метаданих рушія.

Спроєктована модель даних та конфігураційних макетів утворює надійний, оптимізований фундамент для функціонування інструментального засобу. Інтеграція стандартних файлів ресурсів із механізмом поліморфної серіалізації SerializeReference та безпечною обгорткою SerializedType повністю задовольняє функціональні вимоги щодо побудови макетів, що керуються даними. Обраний підхід нівелює потребу в застосуванні сторонніх баз даних, гарантує швидкодію підсистеми збереження на рівні базового конвеєра ресурсів та забезпечує максимальний рівень стійкості конфігурацій до структурних змін у користувацькому коді.

2.3 Декомпозиція та проєктування залежностей підсистем

Для забезпечення високої відмовостійкості та масштабованості системи було проаналізовано можливі варіанти архітектури модулів. Від монолітного підходу, який передбачає тісну інтеграцію механізмів генерації інтерфейсу, обробки даних та математично-логічних обчислень в єдину кодову базу, було вирішено відмовитися. Таке об'єднання значно підвищує вразливість програмного засобу [27]. Виникнення помилки під час зчитування складної несеріалізованої структури даних або зміна зовнішнього експериментального

API, який буде використовуватися в графі логіки, неминуче призвели б до повної зупинки роботи всього інструменту.

Натомість було застосовано принцип жорсткої ізольованої модульності з використанням механізму збірок Assembly Definitions [28]. Для організації взаємодії ядра з іншими, більш складними підсистемами було застосовано принцип інверсії залежностей. Ядро не має прямих посилань на інші модулі, а натомість містить абстрактні інтерфейси, через які розширення можуть інтегрувати свій функціонал. Архітектурно модуль ядра бере на себе відповідальність за такі завдання:

- керування процесом зчитування конфігураційних макетів та ініціалізації базового генератора візуального дерева елементів;
- обробка, створення прив'язки даних та відмальовування виключно серіалізованих властивостей об'єкта;
- надання абстрактних інтерфейсів розширення для підключення зовнішніх механізмів віртуалізації, а також математичної та логічної обробки;
- забезпечення повноцінної базової працездатності візуального редактора у випадку відключення додаткових підсистем шляхом автоматичного використання класів-заглушок, які реалізовують визначені абстрактні інтерфейси без виконання реальної логіки.

Складніші та потенційно нестабільні механізми винесено в окремі, необов'язкові для базового функціонування підсистеми-модулі.

Модуль віртуалізації відповідатиме за аналіз, обробку та тимчасове збереження несеріалізованих полів, методів, а також результатів їхнього виконання. Його архітектурна ізоляція дозволяє уникнути непередбачуваного впливу на процеси в ядрі. Увімкнений модуль віртуалізації створюватиме проксі серіалізовані поля для відповідних несеріалізованих у поточному об’єкті, дозволяючи ядру безперешкодно використовувати їх у звичайному процесі.

Модуль графу логіки забезпечуватиме обчислення математичних та логічних виразів на основі статичних даних та даних поточного об’єкту, функціонуючи на базі відокремлених графових API рушія.

Такий архітектурний розподіл гарантує безперебійну генерацію візуального інтерфейсу основним редактором навіть у випадку навмисного відключення модулів віртуалізації чи обрахувань, або ж у разі виникнення виняткових ситуацій під час їхньої роботи (рисунок 2.2). Коли необов’язкові модулі мають бути явно відключені, замість них будуть встановлені заглушки.

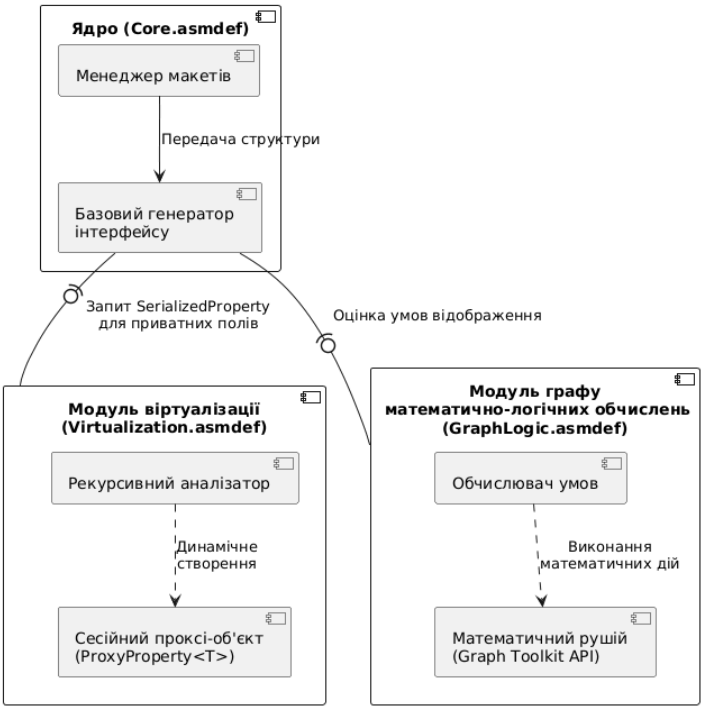


Рисунок 2.2 – Діаграма компонентів «Модулі системи»

2.4 Проєктування ядра генерації візуального інтерфейсу

Центральним компонентом рівня моделі представлення розроблюваного інструментального засобу є ядро генерації візуального інтерфейсу. Його головне архітектурне завдання полягає у перетворенні статичних інструкцій конфігураційного макета на динамічне дерево графічних елементів із

налаштованою двосторонньою прив'язкою даних. Для забезпечення слабкої зв'язності ядро спроєктовано як повністю незалежний конвеєр обробки, який абстрагований від зовнішніх механізмів перехоплення інспекторів та не залежить від способу свого виклику середовищем розробки Unity.

Процес генерації ініціюється спеціалізованим класом-оркестратором, який бере на себе відповідальність за керування послідовністю етапів побудови. На вхід цього конвеєра система передає два фундаментальні параметри: десеріалізований об'єкт макета, що містить ієрархію поліморфних конфігураційних вузлів, та екземпляр базового об'єкта серіалізації рушія, який визначає цільовий скрипт користувача. Вказана взаємодія вказана на рисунку 2.3. Життєвий цикл формування візуального дерева розпочинається з етапу первинної ініціалізації, який охоплює наступні логічні кроки:

– створення порожнього кореневого контейнера візуального елемента, що є основою для подальшого розміщення всіх дочірніх компонентів макету;
– підготовка контексту виконання, під час якої оркестратор перевіряє цілісність переданих структур даних та завантажує абстрактні інтерфейси модулів розширення, зокрема підсистеми віртуалізації та графів;
– запуск алгоритму рекурсивного обходу дерева інструкцій, який ітерується по масиву поліморфних вузлів та делегує їхню обробку наступним складовим частинам конвеєра генерації.

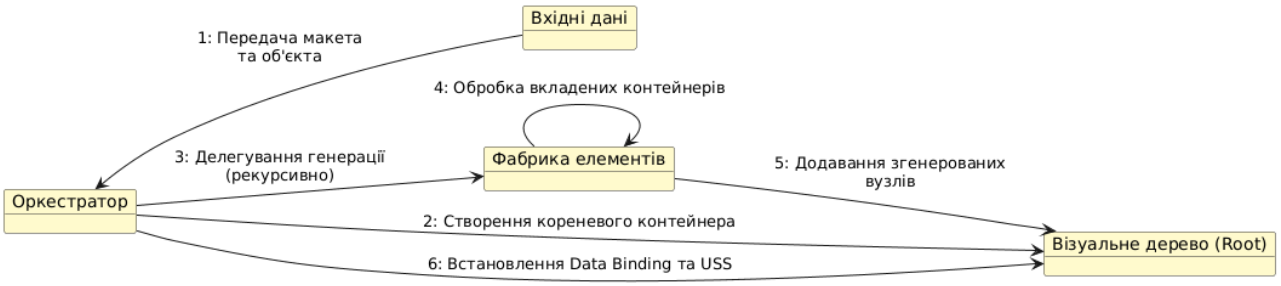


Рисунок 2.3 – Комунікаційна діаграма взаємодії компонентів ядра

Цей підхід дозволяє чітко розмежувати відповідальність у системі, залишаючи за класом-оркестратором виключно функції маршрутизації даних та управління процесом перетворення макета на графічний інтерфейс.

Другим етапом конвеєра генерації є безпосередня трансляція абстрактних структур даних у конкретні графічні компоненти, що реалізується за допомогою фабрики візуальних елементів. Цей архітектурний компонент повністю ізолює логіку створення інтерфейсу від моделі даних, дотримуючись шаблону MVVM та гарантуючи, що конфігураційні вузли залишаються виключно пасивними носіями інформації.

Алгоритм роботи фабрики базується на розпізнаванні типів поліморфних вузлів, що реалізують базовий інтерфейс інструкцій макета, і передбачає виконання таких операцій:

- динамічний аналіз переданого абстрактного вузла та визначення його класу-реалізації, яким може виступати властивість цільового об'єкта, структурний контейнер або декоративний елемент;
- створення відповідного об'єкта фреймворку UI Toolkit, такого як поле введення, текстова мітка чи панель компонування, із початковим налаштуванням його параметрів відображення, враховуючи модифікатори;
- ініціація рекурсивного виклику фабрики у випадку ідентифікації вузла як контейнера, що дозволяє системі обробити всі вкладені списки елементів та точно відтворити багатошарову ієрархію, задану користувачем;
- додавання згенерованих дочірніх графічних компонентів до батьківського вузла-контейнера з подальшим поверненням повністю сформованого піддерева до класу-оркестратора.

Така організація процесу побудови забезпечує високу масштабованість спроєктованого ядра, оскільки додавання підтримки нових типів конфігураційних блоків у майбутньому вимагатиме лише додавання нових класів конфігурації та логіки без необхідності модифікації існуючих структур даних, коду фабрики або зміни загального конвеєра генерації.

2.5 Проєктування модуля віртуалізації несеріалізованих даних

Під час розроблення підсистеми взаємодії з несеріалізованими даними цільового об'єкта, зокрема приватними й статичними полями, властивостями та методами, постала проблема їхньої інтеграції у стандартний конвеєр відмальовування інтерфейсів. Механізми рушія Unity спроєктовані для роботи виключно з даними, що підтримують серіалізацію, що унеможливлює пряме створення графічних елементів контролю для інформації, що не відслідковується серіалізатором Unity.

Для розв'язання цієї проблеми було проаналізовано кілька альтернативних підходів, однак від більшості з них довелося відмовитися:

- від використання системного Reflection.Emit для динамічної генерації типів під час виконання було відмовлено через неминучий процес рекомпіляції коду при кожній зміні структури об'єкта, тому що відповідні класи на рівні рушія не будуть створеними при такому підході [29];
- застосування механізму поліморфної серіалізації на базі атрибута SerializeReference та списків типу System.Object та UnityEngine.Object було визнано неефективним через його архітектурну нездатність підтримувати значущі типи даних, що унеможливлює роботу з будь-якими структурами, включно із примітивними.

У результаті аналізу обмежень було обрано рішення, засноване на шаблоні «Сесійний проксі-об'єкт», яке дозволяє безпечно обійти обмеження стандартного серіалізатора без втручання в його базовий код або порушення стабільності середовища розробки [18, 19].

Основою затвердженого архітектурного рішення є проєктування узагальненого класу ProxyProperty<T>, який функціонує як контейнер-посередник для тимчасового збереження значень приватних полів, властивостей або результатів виконання методів. Для забезпечення сумісності цього класу з підсистемою прив'язки даних, яка вимагає наявності кореневого серіалізованого

об'єкта, розроблено механізм динамічного створення сесійного середовища безпосередньо в оперативній пам'яті.

Так як Unity не може серіалізувати відкриті узагальнення, було спроєктовано базовий інтерфейс IProxyProperty, від якого б наслідувався узагальнений ProxyProperty<T> [30]. У поєднанні з атрибутом SerializeReference це дає змогу серіалізувати в типі IProxyProperty все, що може серіалізувати Unity за замовчуванням, включно із структурами.

Алгоритм функціонування підсистеми віртуалізації на етапі ініціалізації макета включає наступні кроки:

- під час виділення користувачем цільового об'єкта в інспекторі модуль автоматично генерує тимчасовий екземпляр класу ScriptableObject, повністю виключаючи його фізичне збереження як ресурсу на диску;
- у межах створеного екземпляра динамічно формується список обгорток ProxyProperty<T>, де узагальнений параметр T строго відповідає типу оригінальної несеріалізованої змінної;
- за допомогою механізмів рефлексії значення з оперативної пам'яті цільового компонента зчитуються та записуються у відповідні проксі-поля, формуючи серіалізоване представлення даних для їхнього подальшого використання генератором візуального дерева.

Такий підхід дозволяє системі трактувати приховані дані як звичайні серіалізовані властивості, ізолюючи складність роботи з рефлексією на рівні спеціалізованого модуля та не перевантажуючи ядро системи.

Ключовою перевагою розробленої архітектури віртуалізації є те, що згенеровані проксі-властивості сприймаються рушієм Unity як серіалізовані властивості, що дозволяє безперешкодно інтегрувати їх у стандартний конвеєр побудови інтерфейсів. У результаті система успішно розв'язує проблему сумісності з інструментами редактора, забезпечуючи наступні переваги:

- повна сумісність із вбудованими та користувацькими відмальовувачами властивостей, оскільки базова фабрика інтерфейсу передає їм

стандартизований об'єкт SerializedProperty, що дозволяє розробникам перевикористовувати раніше написаний код без жодних модифікацій;

– вбудована підтримка підсистеми скасування та повторення дій, яка функціонує шляхом реєстрації змін безпосередньо у тимчасовому проксі-контейнері, після чого спеціальний обробник модуля віртуалізації перехоплює подію оновлення та за допомогою рефлексії синхронізує змінені дані з оперативною пам'яттю реального компонента.

Для візуалізації описаної системи було створено діаграми послідовності, що показують усі ключові фази процесу віртуалізації: створення, редагування, застосування відміни та знищення. Ці діаграми можна переглянути у графічному документі «Діаграми послідовностей системи віртуалізації».

Застосування шаблону «Сесійний проксі-об'єкт» у поєднанні з узагальненим типом ProxyProperty<T> та динамічним виділенням пам'яті під тимчасовий контейнер утворює надійний міст між статичною підсистемою серіалізації та динамічним станом програми. Такий підхід повністю задовольняє вимоги архітектури MVVM, абстрагуючи ядро генерації візуального дерева від складності роботи з рефлексією та гарантуючи високу стабільність розроблюваного інструментального засобу.

2.6 Проєктування механізму перехоплення відмальовування інспекторів

Для безшовної інтеграції розробленого інструментального засобу в екосистему рушія Unity було спроєктовано механізм перехоплення стандартного процесу відмальовування компонентів. Оскільки система повинна підтримувати роботу з довільними користувацькими скриптами, базове архітектурне рішення передбачає створення глобального класу-перехоплювача, який успадковується від системного класу Editor та використовує атрибут прив'язки з увімкненим параметром поширення на всі дочірні типи об'єктів [31]. Цей механізм

активується автоматично під час виділення будь-якого ігрового об'єкта або ресурсу, заміщуючи собою стандартний виклик методу побудови інтерфейсу.

З метою збереження зворотної сумісності та уникнення конфліктів із вже існуючими компонентами, логіку вибору способу відмальовування побудовано на базі шаблону «Стратегія», що був частково описаний у підрозділі 2.1. Після перехоплення фокусу система не починає негайну генерацію візуального дерева, а виконує аналіз контексту, який складається з таких кроків:

– ідентифікація системного типу виділеного об'єкта та пошук відповідного конфігураційного макета у внутрішньому реєстрі інструменту;
– динамічне сканування існуючих збірок за допомогою рефлексії на наявність специфічних користувацьких редакторів, написаних програмістами вручну для цього конкретного типу;
– ухвалення рішення щодо застосування пріоритетної стратегії відображення, якою може бути побудова інтерфейсу за знайденим макетом, делегування відмальовування сторонньому користувацькому редактору або повернення до базового механізму рушія за замовчуванням.

Реалізацію описаного шаблону можна детальніше розглянути на відповідній діаграмі класів, що представлена на рисунку 2.4.

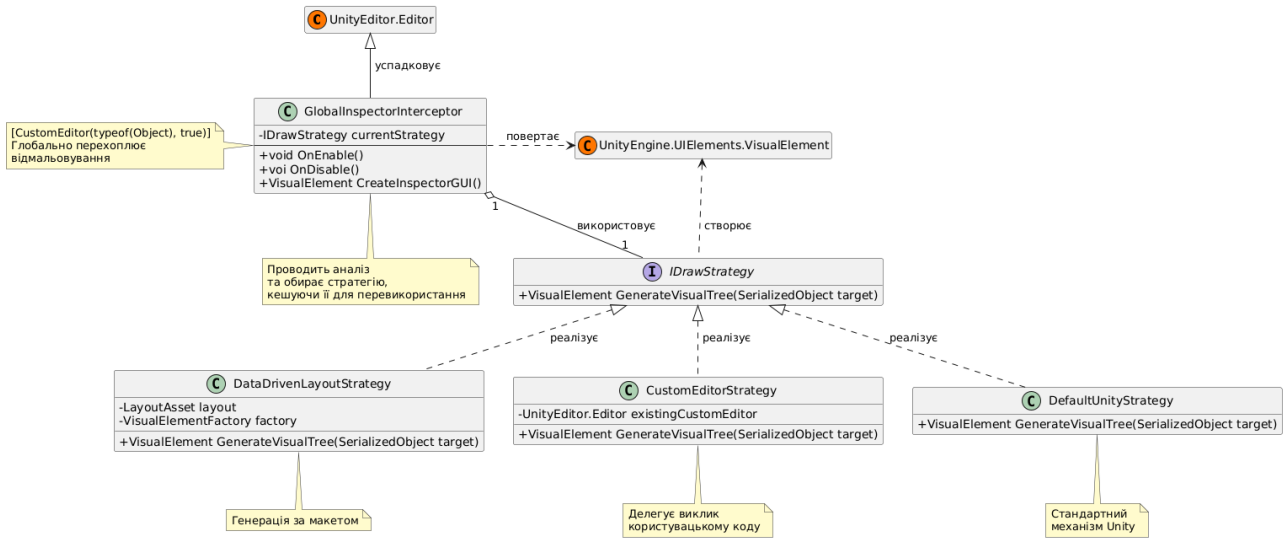


Рисунок 2.4 – Діаграма класів шаблону «Стратегії» при виборі відмальовувача

Для збереження конфігурації про типи та їхні стратегії відмальовування, було використано глобальний реєстр у вигляді ScriptableObject, використовуючи шаблон «Одинака» (Singleton) [18, 19]. Перед створенням відмальовувача для типу, система перевірятиме чи є цей тип в реєстрі. Якщо так, то буде обрано вказану стратегію. Якщо цільовий тип відсутній у реєстрі, система використає стандартний відмальовувач.

У разі вибору генерації з макету як пріоритетної стратегії, клас-перехоплювач ініціює процес побудови графічного інтерфейсу за допомогою фреймворку UI Toolkit. Оскільки цей фреймворк функціонує в режимі збереження стану, перехоплювач перевизначає системний метод CreateInspectorGUI, який відповідає за надання середовищу розробки кореневого візуального елемента інспектора.

Процес формування візуального дерева та його інтеграції з підсистемою відображення відбувається через взаємодію з фабрикою елементів. Візуально цей процес відображений у графічному документі «Діаграма послідовності формування візуального дерева». З ключових етапів можна виділити:

- створення базового контейнера VisualElement, який слугує кореневим вузлом для всього згенерованого інтерфейсу та автоматично адаптується під доступну ширину панелі інспектора;
- передача структури конфігураційного макета та серіалізованого представлення цільового об'єкта до генератора, який рекурсивно обробляє кожен вузол інструкцій і формує ієрархію графічних блоків;
- автоматичне встановлення двосторонньої прив'язки даних між створеними графічними компонентами та відповідними властивостями об'єкта, що гарантує миттєву синхронізацію значень без необхідності ручної обробки подій зміни інтерфейсу;
- динамічне підключення таблиць стилів (USS) для забезпечення візуальної відповідності згенерованого редактора стандартам дизайну Unity та коректного відображення елементів у різних колірних темах середовища.

2.7 Проєктування модуля графу логіки

Для забезпечення динамічної поведінки згенерованого візуального інтерфейсу, такої як динамічні значення параметрів модифікаторів та приховування або блокування певних елементів залежно від поточного стану об'єкта, було спроєктовано спеціалізований модуль графу логіки. Оскільки реалізація власного текстового синтаксичного аналізатора для математичних та логічних виразів є надлишковою та схильною до помилок, архітектурне рішення базується на використанні готового офіційного фреймворку Graph Toolkit від Unity Technologies [32]. Цей модуль виступає ізольованою підсистемою, яка підключається до ядра генератора через заздалегідь визначені абстрактні інтерфейси, що гарантує збереження модульності системи та дозволяє ядру функціонувати навіть у разі відключення графового рушія.

Проєктування модуля графу логіки передбачає реалізацію конвеєра обробки даних, який характеризується такими архітектурними особливостями:

- використання бібліотеки Graph Toolkit як базового рушія для виконання математичних та логічних операцій, що дозволяє уникнути розроблення власного транслятора коду та надає зручний вузловий інтерфейс;
- забезпечення механізму глибокого доступу до членів цільового об'єкта, що дає змогу звертатися до вкладених властивостей;
- визначення як перспективного напряму проєктування системи інтеграції статичних класів та їхніх членів, що в майбутньому дозволить розробникам оперувати глобальними даними середовища (наприклад, класами Time або Mathf) під час обчислення логічних умов безпосередньо у візуальному редакторі;
- оптимізація виконання графу шляхом компіляції логіки вузлів при наявності такої можливості у відповідному контексті.

Для детального розуміння процесу обчислення динамічних умов та взаємодії між ядром і графовим модулем розроблено діаграму діяльності,

представлену на рисунку 2.5, яка демонструє покроковий алгоритм від запиту на значення параметру до зміни графічного інтерфейсу при його отриманні.

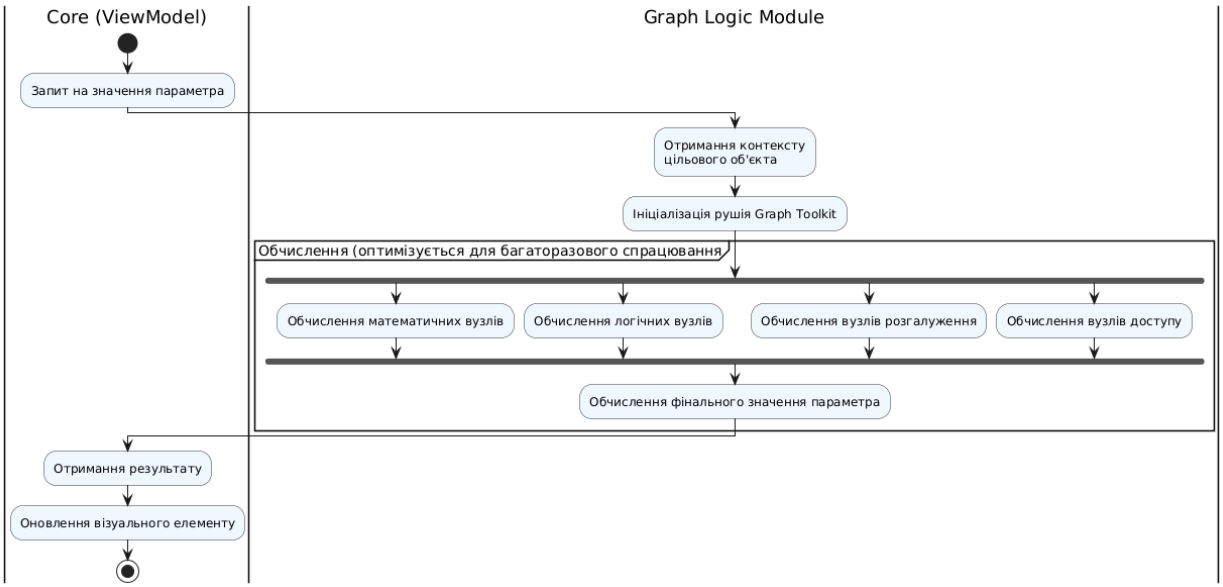


Рисунок 2.5 - Діаграма діяльності процесу обчислення параметра у графі

Процес обчислення логічних умов базується на послідовному перетворенні значень членів цільового об'єкта на динамічний параметр для візуального інтерфейсу. Алгоритм функціонування внутрішнього конвеєра виконання модуля графу логіки включає такі взаємопов'язані етапи:

- зчитування поточних значень із цільового об'єкта та їхня передача як вхідних параметрів до відповідних вузлів графу логіки;
- ініціалізація процесу обходу графу, під час якого послідовно виконуються задані розробником математичні обчислення, логічні порівняння та інші операції, використовуючи отримані вхідні дані;
- формування результату обчислень у кінцевому вузлові графу;
- безпечне повернення отриманого результату назад до обробника редактора на рівні моделі представлення, який на основі цього значення ухвалює рішення зміни стану відповідного візуального елемента.

Такий архітектурний підхід до відокремлення логіки обчислень дозволяє розвантажити базовий генератор інтерфейсу та уникнути жорсткої прив'язки конвеєра відображення до специфіки обробки виразів. Делегування цих завдань офіційному зовнішньому API Graph Toolkit забезпечує максимальну гнучкість налаштування поведінки редактора, виключаючи при цьому потребу в написанні та підтримці власного лексичного аналізатора.

2.8 Проєктування інтерфейсу користувача

У цьому підрозділі описано ключові особливості користувацького інтерфейсу для усіх раніше описаних елементів системи. Усі вони мають дотримуватися наступних загальних вимог:

- візуально інтегруватися під редактор Unity та поточну кольорову тему, використовуючи вбудовані кольорові USS-стилі у випадку власних вікон на основі UI Toolkit;
- підтримувати механізми відміни та повтору дій;
- по можливості використовувати кешування, збереження стану, ліниву ініціалізацію та компіляцію виразів для збільшення швидкодії та загальної оптимізації відмальовки.

Спочатку було розглянуто раніше спроєктований тип даних SerializedType, що відповідає за безпечне збереження посилання на інші типи даних. Він використовуватиметься для:

- визначення цільових класів та структур у реєстрі макетів;
- визначення типу віртуальних членів для цільових об’єктів;
- виборі типів даних у логічному графі.

Для ефективної взаємодії користувача з об'єктами SerializedType було спроєктовано спеціалізоване вікно пошуку TypeSelector на основі UI Toolkit. Його завданнями є автоматичне сканування доступних збірок проєкту,

групування знайдених типів за просторами імен та надання інструменту пошуку по елементам із системою пріоритезації. Система пріоритезації надаватиме перевагу коротшим назвам типів та тим типам, що не знаходяться в глибоких ієрархіях просторів імен, тим самим пропонуючи користувачеві більш релевантні результати. Знайдені у збірках типи даних будуть кешуватися для перевикористання у наступних запитах. Такий підхід до вікна пошуку повністю усуває необхідність ручного введення складних текстових ідентифікаторів та мінімізує ризик виникнення синтаксичних помилок на етапі конфігурування макета, водночас безшовно інтегруючись у взаємодію з редактором.

Далі було спроєктовано графічний інтерфейс конструювальника макетів. Так як макет представлятиме собою об’єкт ScriptableObject, найпростішим варіантом є використання стандартного автоматичного інспектора серіалізованих полів з додаванням незначних косметичних змін. Для вибору типу поліморфних посилань буде використовуватися окремий селектор на базі раніше спроєктованого TypeSelector. Цей метод має бути достатнім для функціонування та розуміння кінцевим користувачем.

У перспективі розглядається реалізувати інтерфейс конфігуратора у вигляді окремого нестандартного вікна редактора, яке надаватиме користувачеві візуальний інструментарій для збирання та редагування конфігураційних файлів з додатковими зручностями. Структурно вікно конструювальника буде розроблено на базі фреймворку UI Toolkit та поділено на кілька взаємопов'язаних робочих зон, які забезпечуватимуть зручний процес розробки:

- зона ієрархії блоків, що відображатиме поточну структуру макета у вигляді дерева з підтримкою операцій перетягування для зміни порядку елементів та їхнього логічного вкладення у відповідні контейнери;
- панель інспектора властивостей, яка динамічно оновлюватиметься під час виділення конкретного блоку та надаватиме розширений доступ до його внутрішніх налаштувань, таких як шлях до цільової серіалізованої властивості, характеристик та списку застосованих модифікаторів;

– верхня панель інструментів та контекстна бібліотека компонентів, що міститимуть категоризований перелік доступних для створення блоків та дозволятиме швидко додавати нові структурні блоки до відкритого макета.

Останнім було розглянуто користувацький досвід взаємодії з модулем графу логіки, що орієнтований на надання розробникам зручного візуального інструментарію для налаштування динамічної поведінки інтерфейсу без необхідності написання програмного коду. Редактор вузлів було спроєктовано як окреме вікно побудоване з використанням UI Toolkit, яке можна викликати із проєктувальника макетів редактора, що забезпечуватиме виконання наступних ключових користувацьких сценаріїв:

– прив'язки створеного логічного графу до конкретного конфігураційного вузла макета, наприклад, додавання умови відображення до поля або визначення динамічного максимального значення для повзунка;
– роботи безпосередньо у вузловому середовищі, де користувач додає вхідні змінні, обираючи їх із розгорнутого списку доступних властивостей цільового об'єкта, та об'єднує їх візуальними зв'язками із вузлами математичних чи логічних операторів;
– збереження та валідація побудованої схеми обчислень, що дозволяє системі автоматично перевірити граф на наявність замкнених циклів або конфліктів типів даних перед застосуванням його в генераторі інтерфейсу.

Для візуалізації описаних можливостей взаємодії користувача з інструментальним засобом під час роботи із графом було створено діаграму (рисунок 2.6), яка відображає основні варіанти використання системи.

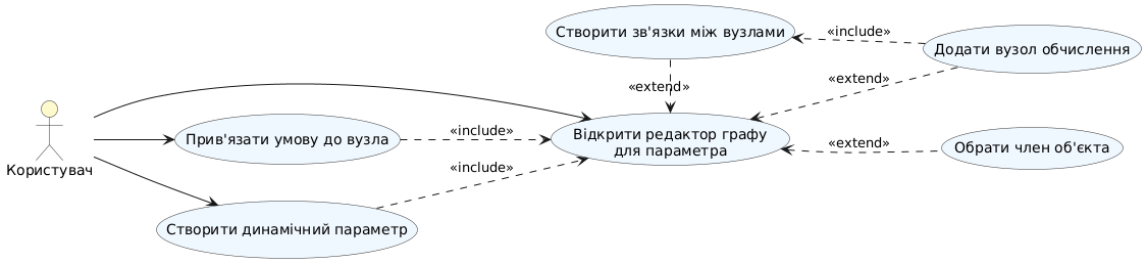


Рисунок 2.6 – Діаграма варіантів використання налаштування графу логіки

2.9 Висновки та результати проєктування

У цьому розділі було проведено комплексне проєктування архітектури, структур даних та взаємодії підсистем розроблюваного інструментального засобу. Фундаментом обрано шаблон MVVM для інтеграції з UI Toolkit та двосторонньої прив'язки даних. Для досягнення надійності, гнучкості та масштабованості архітектуру побудовано за принципом ізольованої модульності з автономним ядром та необов'язковими підсистемами, а також доповнено шаблонами проєктування, що підтримують легку розширюваність.

Модель збереження даних базується на вбудованих контейнерах ScriptableObject із застосуванням механізмів поліморфної серіалізації та безпечної обгортки посилань. Це гарантує надійне збереження ієрархічних макетів, безшовну сумісність із системою скасування й повторення дій та високу стійкість конфігурацій до структурних змін у цільовому коді користувача.

Спроєктована система функціонує як набір незалежних модулів. Ядро відповідає за генерацію візуального дерева, тоді як механізм глобального перехоплення інспекторів забезпечує інтеграцію інструменту в екосистему Unity з підтримкою зворотної сумісності. Ізольовані підсистеми віртуалізації та графу логіки дозволяють прозоро обробляти несеріалізовані дані й складні вирази без створення власного синтаксичного аналізатора чи зміни коду генератора.

У результаті проєктування сформовано комплексну, відмовостійку та розширювану систему візуального конструювання. Розроблено специфікації графічного інструментарію, що включають конфігуратор макетів, пошук типів та вузловий редактор логіки. Визначені специфікації повністю задовольняють вимоги до сучасного користувацького досвіду в середовищі розробки. Запропоновані рішення усувають потребу в ручному написанні коду для користувацьких інспекторів і розв'язують головну проблему дослідження, дозволяючи ітеративно налаштовувати інтерфейси, повністю уникаючи ресурсомісткого процесу рекомпіляції скриптів проєкту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Реалізація базових контрактів та засобів навігації у структурах даних

Відповідно до декомпозиції підсистем, описаної у підрозділі 2.3, програмну реалізацію було розпочато із закладення фундаменту, створення базових контрактів для забезпечення жорсткої ізольованої модульності Ядра. Було розроблено абстрактні інтерфейси IVirtualizationProvider та IGraphProvider. За відсутності модулів віртуалізації та графів логіки, згідно з планом проєктування було створено класи-заглушки NoOpVirtualizationProvider та NoOpGraphProvider. Для керування цими залежностями було реалізовано статичний клас ModuleRegistry, який функціонує за шаблоном «Службовий локатор» (Service Locator) [18]. Це рішення забезпечило повну автономність Ядра та його готовність до інтеграції складніших підсистем у майбутньому без потреби модифікувати наявну кодову базу.

Наступним кроком стала програмна реалізація раніше спроєктованих механізмів безпечного збереження посилань на типи. Було реалізовано клас-обгортку SerializedType. Для мінімізації ресурсомістких викликів рефлексії під час роботи інструменту було застосовано механізм відкладеного кешування знайдених типів. Для взаємодії користувача із цими даними було розроблено візуальний компонент TypeSelectionWindow. Під час його створення виявилося, що у API Unity відсутні готові інструменти для отримання структурованого дерева типів. Проблему було розв'язано розробленням власного механізму:

- динамічне сканування всіх доступних збірок проєкту через виклик AppDomain.CurrentDomain.GetAssemblies();
- збереження та оптимізація результатів у статичному класі TypeSelector;
- реалізація системи пошуку з пріоритезацією результатів.

Окрім виконання пошуку за рядком, вікно TypeSelectionWindow дозволяє ієрархічно групувати типи за просторами імен. Також можливо задати базовий тип, що обмежуватиме вибір лише своїми нащадками.

Далі було реалізовано інструментарій для визначення та аналізу шляхів до членів класу. Створено клас MemberPath, який зберігає ієрархічний шлях у вигляді текстового рядка та обчислює кінцевий тип і значення під час виконання. Для отримання інформації про доступні поля, властивості та методи застосовано клас MemberScanner, результати роботи якого акумулюються у класі ReflectionCache. Щоб забезпечити гнучкість налаштування візуального відображення, було впроваджено систему побітових масок MemberFilter, яка дозволяє розмежовувати видимі елементи від тих, що доступні для остаточного вибору. Фінальну структуру реалізованих класів представлено на рисунку 3.1.

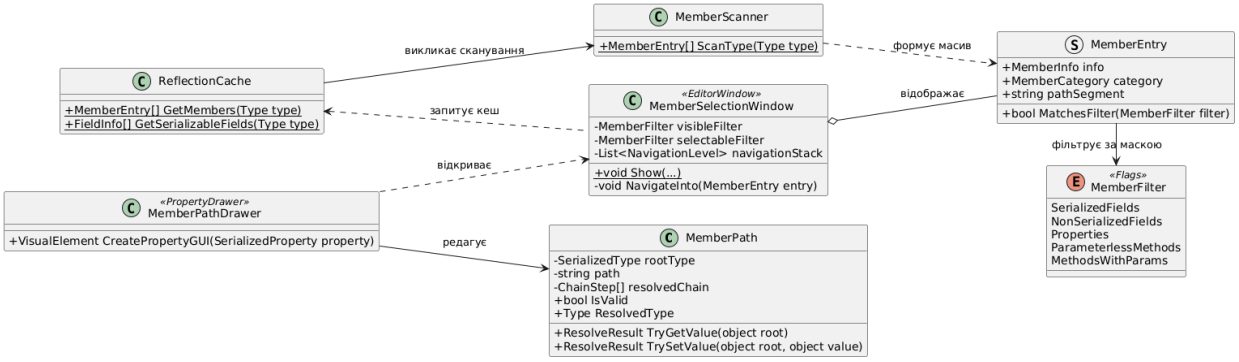


Рисунок 3.1 – Діаграма класів MemberPath

Візуальну взаємодію з побудованими шляхами забезпечено створенням вікна MemberSelectionWindow. Цей компонент проєктувався з ухилом на полегшення навігації та вибору, враховуючи різні сценарії внутрішньої будови типу. Для досягнення цієї мети у вікні було реалізовано:

- механізм ієрархічного обходу складних структур даних із заглибленням;
- панель навігації у вигляді «хлібних крихт» (breadcrumbs) для зручного повернення на попередні рівні вкладеності;
- візуальну індикацію характеристик членів класу за допомогою спеціалізованих кольорових позначок.

У результаті було отримано досить зручне та інформативне вікно. Його можна переглянути на рисунку 3.2.

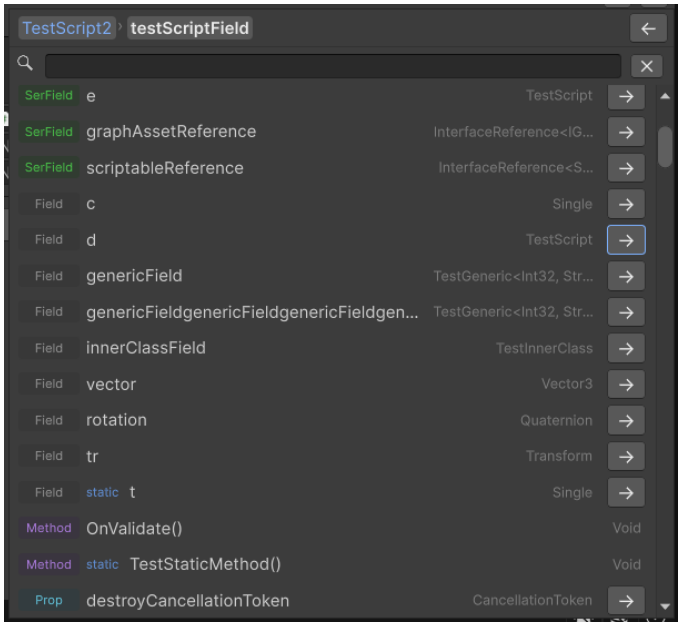


Рисунок 3.2 – Вікно MemberSelectionWindow

3.2 Програмна реалізація системи джерел даних та макетів

Відповідно до спроєктованої структури конфігураційних макетів, виникла необхідність у програмній реалізації системи поліморфних полів даних для зв'язування візуального інтерфейсу з даними об'єктів. Основою цієї підсистеми став базовий клас DataField. Для гарантії безпеки типів та гнучкості архітектури його було розділено на дві конкретні реалізації: строго типізований DataField<T> та абстрактний DataFieldAny. Використання узагальненого типу DataField<T> забезпечило перевірку типів під час компіляції для візуальних вузлів та модифікаторів, що вимагають конкретних даних (наприклад, числових значень для повзунка), тоді як DataFieldAny дозволив реалізувати динамічне зв'язування для полів, тип яких визначається лише під час виконання.

Інженерною перевагою класу DataField стала інкапсуляція логіки отримання даних з трьох джерел, що визначаються переліком DataSourceMode:

– режим ObjectBound використовує реалізований раніше клас MemberPath для динамічного доступу до членів цільового об'єкта, а також IVirtualizationProvider для візуалізації, якщо член не серіалізований;

- режим Manual використовує введене в редакторі значення, при потребі у візуалізації виконує запит до інтерфейсу IVirtualizationProvider для динамічного створення проксі-об'єкта в оперативній пам'яті;
- режим Graph повністю делегує обчислення інтерфейсу IGraphProvider, можливість візуалізації забезпечується IVirtualizationProvider.

Такий підхід дозволяє користувачу обирати динамічні значення для параметрів макету, а також джерела даних для полів. При відсутності модулів віртуалізації чи графів виводиться повідомлення з поясненнями для користувача.

Наступним етапом стала програмна реалізація моделі макетів. Було створено базові абстрактні класи LayoutElement, LayoutNode та LayoutModifier. Оскільки макет повинен зберігати неоднорідні елементи в єдиному списку, було застосовано механізм серіалізації поліморфних даних за допомогою атрибута SerializeReference. Вбудована система Unity не вміє візуалізувати механізм вибору поліморфного значення, тому для зручності роботи було спроєктовано спеціальну структуру-обгортку PolymorphicElement<T>. Це рішення дозволило відокремити логіку вибору конкретного типу поля, використовуючи TypeSelectionWindow, від механізмів серіалізації та візуалізації вже створених об’єктів Unity, зберігши при цьому абсолютну цілісність даних.

Ієрархічні дерева вузлів було інкапсульовано у контейнері LayoutAsset, який успадковано від класу ScriptableObject. Цей клас зберігає цільовий тип у вигляді об'єкта SerializedType та список поліморфних вузлів макета.

Для зв'язування створених макетів із цільовими типами було реалізовано глобальний реєстр InspectorLayoutRegistry, також побудований на базі ScriptableObject. Для забезпечення гнучкості налаштувань записи у реєстрі було розділено на дві категорії:

- InspectorEntry призначені для налаштування компонентів MonoBehaviour та ресурсів ScriptableObject;
- PropertyDrawerEntry призначені для будь-яких користувацьких серіалізованих типів даних.

Вибір стратегії відображення було реалізована через переліки RenderStrategy та PropertyDrawerStrategy. Завдяки цьому система отримала здатність динамічно ухвалювати рішення щодо способу відмальовування:

- генерувати інтерфейс з макета;
- викликати раніше створений користувачем CustomEditor для інспекторів або CustomPropertyDrawer для властивостей з можливістю вибору;
- повертатися до стандартного механізму Unity, ігноруючи нашу систему;
- генерувати інтерфейс, ігноруючи будь-які перевизначення, як для типу, так і для поля (лише для властивостей).

Для елементів, що візуалізують поля, було додано можливість перевизначати бажану стратегію. Таким чином стає можливим використовувати декілька видів відмальовки одного й того ж типу в залежності від ситуації.

Для оптимізації пошуку потрібного макета, реєстр було оптимізовано шляхом кешування результатів у системні словники. Це забезпечило миттєвий пошук LayoutAsset за типом об'єкта під час виконання зі складністю O(1), що позитивно вплинуло на продуктивність при великій кількості макетів у реєстрі.

3.3 Програмна реалізація структурних вузлів та модифікаторів макета

Для забезпечення гнучкості та розширюваності системи візуального конструювання було застосовано підхід, що розмежовує дані та логіку їхнього відображення. Кожен компонент було розділено на дві незалежні сутності: клас для збереження конфігураційних даних, який успадковується від базових абстракцій LayoutNode або LayoutModifier, та відповідний клас-обробник, що реалізує абстрактні інтерфейси INodeGenerator або IModifierApplicator.

З метою забезпечення надійності та уникнення конфліктів під час конструювання інтерфейсу було розроблено спеціалізовану систему обмежень та можливостей. Вона функціонує за допомогою таких складових:

- використання переліку побітових прапорців NodeCapabilities (наприклад, SupportsLabel, HasValue, IsContainer), який чітко визначає функціональні можливості кожного конкретного структурного вузла;
- застосування атрибута ModifierConstraint безпосередньо до класів модифікаторів для визначення їхніх вимог до можливостей цільових вузлів та підтримуваних типів даних;
- автоматична перевірка сумісності на етапі валідації макета, що унеможливлює помилкові конфігурації (наприклад, застосування модифікатора повзунка SliderModifier до текстової мітки LabelNode).

Для покриття більшості потреб розробників було створено базовий набір структурних вузлів, які відповідають за логічну організацію простору та безпосереднє відображення даних. Детальний перелік реалізованих структурних вузлів наведено у таблиці Б.1.

Модифікатори були розроблені для зміни візуального представлення або поведінки наявних вузлів. За порядком виконання їх було розділено на структурні, які застосовуються першими і повністю змінюють тип графічного елемента, та декоративні, які застосовуються в кінці і впливають лише на його стилістику. Перелік реалізованих модифікаторів наведено у таблиці Б.2.

3.4 Автоматизація створення макетів

Для забезпечення ефективного переходу на нову візуальну систему без необхідності ручного перенесення всіх наявних налаштувань відображення, було розроблено інструмент LayoutDisassembler. Він автоматизує процес первинного створення конфігураційного макета шляхом аналізу наявного коду C#.

Основою механізму автоматизації стала система декларативного відображення. Було створено спеціальний атрибут DisassemblesFromAttribute, який застосовується до класів-спадкоємців LayoutElement. Замість жорсткого

кодування перевірок для кожного стандартного атрибута Unity, система використовує рефлексію та механізм TypeCache для динамічного сканування збірок [33]. У результаті сканування формується словник відповідностей, який пов'язує атрибути Unity (наприклад, Range, Space) із відповідними вузлами або модифікаторами розробленої системи (SliderModifier, SpaceNode). Такий підхід забезпечує дотримання принципу відкритості/закритості, дозволяючи легко додавати підтримку нових атрибутів-замісників у майбутньому [34].

Процес автоматизованого розбирання ініціюється викликом відповідного методу і виконує генерацію макета за таким алгоритмом:

- система отримує цільовий тип із LayoutAsset та ініціалізує словник відповідностей атрибутів;
- якщо цільовий тип є спадкоємцем UnityEngine.Object, на початок макета додається ScriptFieldNode (поле посилання на скрипт) із модифікатором ReadOnlyModifier, після чого додаються всі серіалізовані поля;
- якщо цільовий тип не є спадкоємцем UnityEngine.Object (наприклад, звичайний серіалізований клас), система перевіряє його на наявність складної структури: якщо клас має серіалізовані поля, вони автоматично поміщаються всередину контейнера FoldoutNode, якому призначається модифікатор LabelModifier для відображення назви властивості;
- у випадку, якщо тип не є UnityEngine.Object і не має серіалізованих полів (простий тип), до макета додається лише базовий вузол DefaultEditorNode;
- під час обробки кожного окремого поля рекурсивно аналізуються його атрибути, на основі яких створюються структурні вузли або модифікатори;
- перед додаванням згенерованих модифікаторів до PropertyFieldNode, вони проходять перевірку на сумісність із типом поля за допомогою допоміжного класу ModifierConstraintValidator;
- сформований список елементів перезаписує конфігураційній файл.

Для зручності користувача кнопка для ініціювання розбирання була розміщена знизу інспектора LayoutAsset. При натисканні користувачу

повідомляється про перезапис існуючої конфігурації та запитується його згода. Такий підхід дозволяє мінімізувати випадкові перезаписи, навіть коли кнопка розміщена на видному місці.

3.5 Програмна реалізація конвеєра генерації візуального інтерфейсу

Відповідно до архітектури рівня представлення, спроєктованої у підрозділі 2.4, головним завданням етапу програмної реалізації було забезпечення перетворення статичних інструкцій макета на динамічне дерево графічних елементів VisualElement із налаштованою двосторонньою прив'язкою даних. Керування конвеєром побудови було інкапсульовано у класі InspectorOrchestrator. Під час виклику цей клас ініціює контекст генерації GenerationContext та розпочинає рекурсивний обхід конфігураційного макета.

GenerationContext зберігає в собі інформацію про поточний об’єкт, властивість, редактор, бажаний текст мітки, провайдери модулів, батьківський контекст та інструменти діагностики. Надаючи фабриці усі необхідні дані.

Безпосереднє конструювання візуальних об'єктів було делеговано динамічній фабриці ElementFactory. Важливим рішенням під час розроблення фабрики стала відмова від використання жорстко закодованих умовних конструкцій для ідентифікації та обробки типів вузлів. Замість цього було впроваджено архітектуру на основі спеціалізованих атрибутів CustomNodeGenerator та CustomModifierApplicator.

Під час ініціалізації редактора система використовує механізм TypeCache для сканування збірок, знаходить всі класи, позначені вказаними атрибутами, створює їх екземпляри та реєструє у внутрішніх словниках як активні обробники. Такий підхід забезпечує високу розширюваність, оскільки додавання нових типів вузлів або модифікаторів вимагатиме лише створення нових класів без жодного втручання у вихідний код фабрики та інших класів Ядра.

У ході інтеграції системи прив'язки даних було виявлено технічну проблему несумісності форматів запису шляхів. Внутрішня система серіалізації Unity, представлена властивістю SerializedProperty.propertyPath, використовує специфічний синтаксис для вказівки на елементи списків (.Array.data[n]). Водночас розроблений інструмент MemberPath використовує стандартну нотацію C# для доступу до несеріалізованих даних. Для підтримки списків він має механізм для прямого індексування ланок шляху (у вигляді [n]). Проте у певних ситуаціях існує необхідність поєднувати ці типи шляхів.

Для розв'язання цього конфлікту було розроблено утиліту EditorBindingUtility, яка на льоту виконує трансляцію синтаксису, забезпечуючи коректне поєднання шляхів батьківського об'єкта та локальних членів класу виражених через MemberPath. Трансляція працює в двох режимах: поєднання для рефлексії з поверненням MemberPath як результату та поєднання для UI Toolkit з поверненням шляху у вигляді рядка. Це дозволило безшовно інтегрувати стандартну прив'язку даних UI Toolkit із власною рефлексією, необхідною для роботи системи.

Схематичну модель трансляції форматів шляхів за допомогою розробленої утиліти наведено на рисунку 3.3.

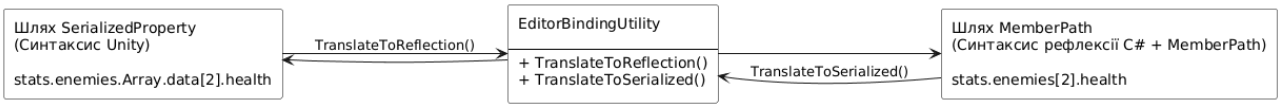


Рисунок 3.3 – Схематична діаграма трансляції форматів шляхів до властивостей

Під час конструювання складних інтерфейсів було виявлено критичну проблему адаптивного вирівнювання елементів. Стандартний механізм вирівнювання міток у полях вводу Unity, представлений класом unity-base-field__aligned, був жорстко розрахований виключно на вертикальне компонування інспектора [35]. При використанні вузла GroupNode у горизонтальному режимі це призводило до порушення пропорцій, через що

мітки не стискалися належним чином, а елементи перекривали один одного. Для розв'язання цієї проблеми було розроблено клас FieldAlignmentHelper. Цей клас автоматично заміняє стандартні стилістичні класи вирівнювання міток Unity та перехоплює події зміни геометрії GeometryChangedEvent [36]. Замість стандартного розрахунку, алгоритм динамічно обчислює ширину міток пропорційно до кількості дочірніх елементів у поточній горизонтальній групі, що забезпечує коректне відображення вкладених полів. Якщо мітка не знаходиться у горизонтальній групі, відбувається стандартне вирівнювання.

Ще одним складним викликом стала необхідність захисту конвеєра генерації від циклічних подій зміни стану. У випадках, коли до одного поля застосовувалося декілька взаємовиключних або обмежувальних модифікаторів (наприклад, одночасне застосування MinModifier та MaxModifier), виникали критичні зависання редактора Unity. Це відбувалося через те, що один модифікатор змінював значення, ініціюючи подію ChangeEvent, яку відразу перехоплював інший модифікатор, знову змінюючи значення і створюючи нескінченний цикл викликів [36].

Для усунення цього недоліку було програмно реалізовано утиліту SafeChangeEventHelper, яка інтегрувалася у логіку роботи модифікаторів. Функціонування відбувається на таких принципах:

- перевірка наявності фокусу користувача на елементі для ігнорування програмних або фонових змін;
- фіксація часу виникнення події через EditorApplication.timeSinceStartup та обчислення різниці з попереднім викликом;
- ведення підрахунку кількості змін для кожного окремого графічного елемента за допомогою ConditionalWeakTable;
- блокування подальшої обробки подій, якщо кількість змін перевищила встановлений ліміт 10 викликів за проміжок часу менший 0.001 секунди.

Діаграму діяльності, що ілюструє процес безпечної обробки подій модифікаторами за допомогою SafeChangeEventHelper, наведено на рисунку 3.4.

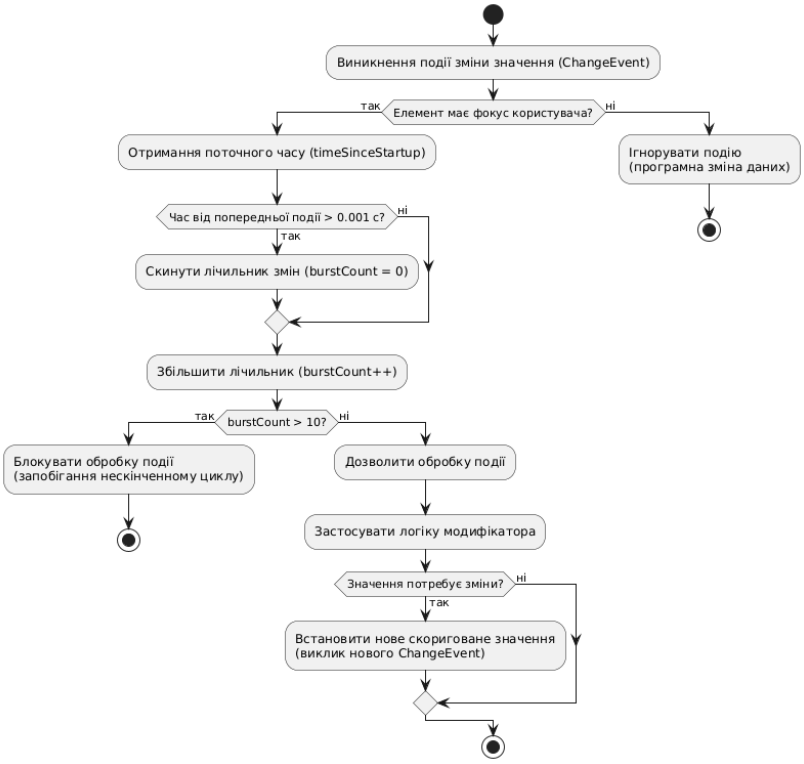


Рисунок 3.4 – Діаграма діяльності алгоритму безпечної обробки подій

3.6 Механізми перехоплення та забезпечення зворотної сумісності

Згідно з результатами проєктування система повинна автоматично замінювати стандартні інспектори та властивості Unity без необхідності написання користувачем атрибутів (CustomEditor, CustomPropertyDrawer) для кожного окремого скрипта. З цією метою було розроблено підсистему глобального перехоплення відмальовування інспекторів.

Перехоплення компонентів та скриптабельних об’єктів було реалізовано через класи, які позиціонують себе як редактори для всіх MonoBehaviour, ScriptableObject та їхніх наслідників. Щоб уникнути дублювання коду, процеси відображення були делеговані до статичного класу InterceptorLogic. Він аналізує конфігурацію у реєстрі та обирає відповідну стратегію відображення: застосування макета, виклик користувацького редактора або повернення до стандартного механізму Unity. Код механізму перехоплення:

```
[CustomEditor(typeof(MonoBehaviour), true)]
[CanEditMultipleObjects]
public class VisualEditorMonoBehaviourInterceptor : UnityEditor.Editor
{
    public override VisualElement CreateInspectorGUI()
    {
        return InterceptorLogic.CreateInspectorGUI(this);
    }
}

[CustomEditor(typeof(ScriptableObject), true)]
[CanEditMultipleObjects]
public class VisualEditorScriptableObjectInterceptor : UnityEditor.Editor
{
    public override VisualElement CreateInspectorGUI()
    {
        return InterceptorLogic.CreateInspectorGUI(this);
    }
}
```

Особливу увагу було приділено класу для перехоплення відмальовування окремих властивостей. Його реалізовано таким чином, щоб він був здатен працювати незалежно від головного інспектора. Це означає, що перехоплювач властивостей успішно функціонує навіть у тих інспекторах, які не використовують розроблювану систему як основу для свого відображення.

Впровадження глобального перехоплювача створювало ризик конфлікту з уже наявними користувацькими редакторами та відмальовувачами властивостей, які розробники могли створити раніше. Для розв'язання цієї проблеми було створено клас CustomEditorResolver. Він використовує системний механізм TypeCache для сканування збірок на наявність користувацьких атрибутів CustomEditor та CustomPropertyDrawer. Для коректної роботи цього механізму було додано жорстку фільтрацію, яка виключає власні класи-перехоплювачі з процесу пошуку, запобігаючи хибним спрацюванням.

Під час інтеграції системи виникла проблема забезпечення зворотної сумісності з парадигмою IMGUI. Розроблений механізм узагальнює та обробляє дві різні ситуації:

- якщо обраний для відображення користувацький Editor чи CustomDrawer не підтримує сучасний фреймворк UI Toolkit (не має реалізації CreatePropertyGUI), система автоматично створює для нього обгортку IMGUIContainer, усередині цієї обгортки викликається застарілий метод OnGUI, що дозволяє безпечно інтегрувати старий процедурний код усередину нового декларативного візуального дерева;
- якщо перехоплювач властивості викликається в контексті інспектора, що повністю відмальовується через IMGUI (наприклад, у вбудованих компонентах Unity), система перехоплює метод OnGUI та викликає EditorGUI.PropertyField, який автоматично створює найбільш доречну візуалізацію поля з підтримкою IMGUI, забезпечуючи безперебійну роботу старого інтерфейсу.

Ще однією проблемою стало виникнення нескінченної рекурсії при виборі стратегії відмальовування Native або Standard. При спробі повернутися до стандартного інспектора Unity, рушій намагався відмалювати об'єкт і знову викликав глобальний перехоплювач, створюючи замкнений цикл. Для уникнення рекурсії було розроблено клас DefaultPropertyFieldHelper. Замість виклику загального механізму відмальовування, він використовує безпечні прямі обгортки (наприклад, конкретні класи IntegerField, ObjectField), розриваючи таким чином петлю викликів. Окрім цього також створений захист від рекурсивної вкладеності на основі контексту GenerationContext.

Процес обробки властивостей із врахуванням режимів зворотної сумісності проілюстровано на діаграмі послідовності (рисунок 3.5).

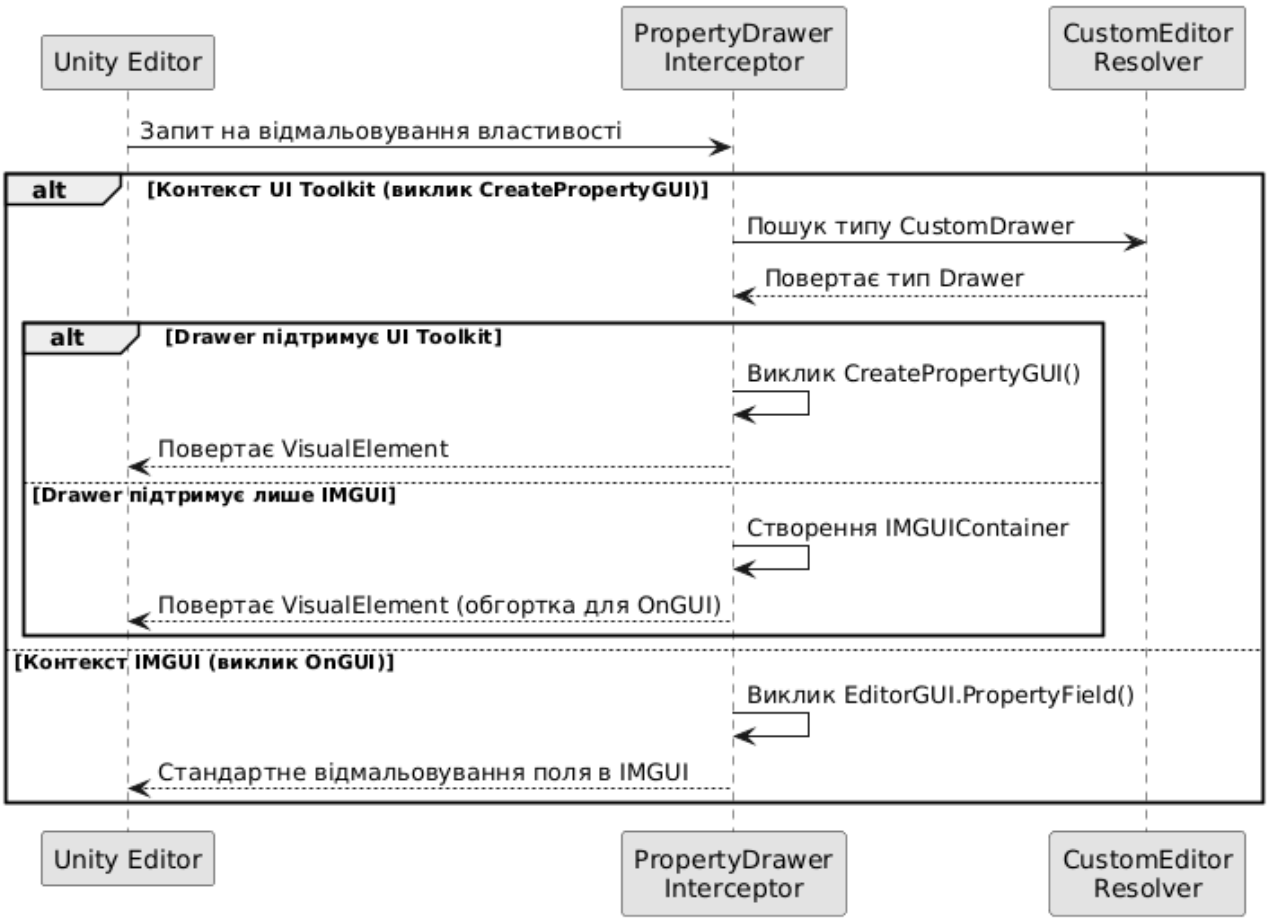


Рисунок 3.5 – Діаграма послідовності процесу перехоплення властивостей із підтримкою зворотної сумісності

3.7 Забезпечення оновлення макетів у реальному часі

Для покращення користувацького досвіду при конфігуруванні редакторів було реалізовано підсистему оновлення макетів у реальному часі. Хоча фреймворк UI Toolkit підтримує динамічну зміну візуального дерева, необхідно було рішення, яке могло б замінити корінний елемент цілком, включно із його дочірніми елементами, а також не потребувало б зміни логіки самих елементів. Для цього системі був необхідний централізований контролер для ініціації процесу перебудови під час модифікації даних конфігурації.

Механізм відстеження змін було побудовано на основі вбудованого в Unity методу OnValidate(), який викликається при будь-якій модифікації об'єкта в

редакторі. Усередині класів LayoutAsset та InspectorLayoutRegistry цей метод генерував відповідні події OnLayoutAssetValidated та OnRegistryValidated.

Керування життєвим циклом оновлень було делеговано статичному класу InspectorReloadManager, який автоматично ініціалізується під час запуску редактора завдяки атрибуту InitializeOnLoad [37].

Оскільки зміна параметрів макета (наприклад, швидке перетягування повзунка в інспекторі) могла викликати події модифікації сотні разів на секунду, безпосереднє перемальовування інтерфейсу призводило б до значного падіння частоти кадрів редактора. Для оптимізації продуктивності було впроваджено механізм затримки перебудови. Система використовує таймер nextReloadTime та підписується на подію EditorApplication.update [38]. При надходженні сигналу про зміну таймер запускається, а фактичне перемальовування відбувається лише після короткої затримки. Якщо таймер вже запущений, то сигнал ігнорується.

Процес підміни інтерфейсу вимагає збереження контексту відкритого вікна інспектора. Оскільки пряма заміна кореневого елемента призводила б до втрати цілісності структури, було реалізовано метод RegisterInterceptor, який створював постійний візуальний контейнер-обгортку ve-inspector-reload-wrapper. Під час спрацювання таймера метод PerformReload очищав уміст цієї обгортки та повторно викликав метод CreateInspectorGUI у класі InterceptorLogic, генеруючи нове візуальне дерево і вставляючи його у наявну обгортку, повторно ініціюючи прив’язку даних до тих самих об’єктів.

Для запобігання витокам пам'яті було впроваджено механізм управління життєвим циклом відкритих інспекторів. Система зберігала посилання на них у спеціалізованих словниках activeInterceptors та activeLayouts. Перед кожним циклом оновлення викликався метод CleanDisposedEditors, який виявляв закриті або знищені вікна та видаляв пов'язані з ними дані з оперативної пам'яті системи.

Діаграму станів, що ілюструє оптимізований процес відстеження змін та оновлення інтерфейсу, наведено на рисунку 3.6.

Код основних складових розробленої системи наведено у додатку В.

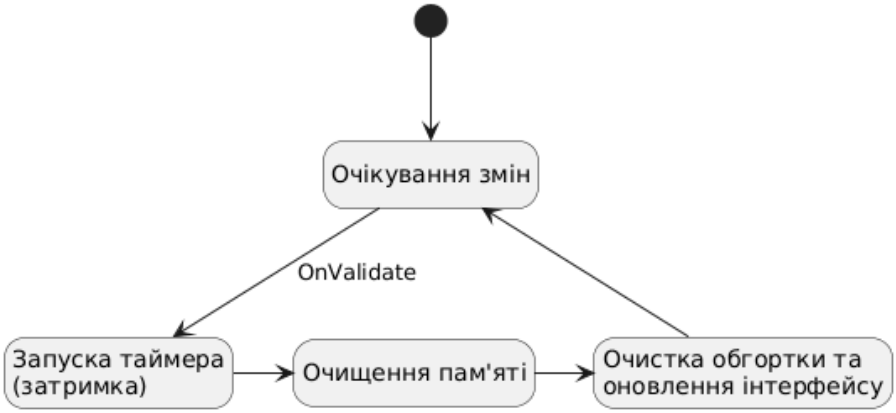


Рисунок 3.6 – Діаграма станів процесу оновлення макетів

3.8 Технічні вимоги до програмного забезпечення

Розроблений інструментальний засіб є внутрішнім розширенням для середовища розробки Unity. Оскільки він не функціонує як самостійний застосунок, його системні вимоги безпосередньо базуються на вимогах необхідних для базової роботи самого ігрового рушія.

Узагальнені вимоги до складу апаратури, операційної системи, обсягу пам'яті та додаткових програм подано у таблиці 3.1.

Основні фактори, що зумовили формування зазначених технічних вимог:

- глибока інтеграція із сучасним фреймворком UI Toolkit та новими API, що унеможливлює використання старіших версій рушія;
- активне залучення механізмів кешування, рефлексії та динамічної генерації візуальних дерев, що може потребувати розширеного обсягу оперативної пам'яті;
- орієнтація користувацького інтерфейсу на базові пристрої введення, що повністю виключає необхідність використання спеціалізованої периферії.

Таблиця 3.1 – Технічні вимоги до програмного та апаратного забезпечення

Характеристика	Вимога або рекомендація
Базове програмне забезпечення	редактор Unity версії 6000.3 LTS або новішої
Операційна система (ОС)	Windows 10 21H1 (64-bit), Windows 11 21H2 (64-bit), macOS 13 або новіші; підтримувані дистрибутиви Linux (64-bit), Ubuntu 22.04, Ubuntu 24.04
Процесор (CPU)	сумісний з архітектурою x64 та набором інструкцій SSE2, Arm64 або ж Apple Silicon
Оперативна пам'ять (RAM)	мінімально 8 ГБ, рекомендовано 16 ГБ та більше для комфортної роботи, особливо для великих проєктів
Зовнішня пам'ять (Storage)	SSD-накопичувач, до 10 МБ вільного простору для розширення та 10–20 ГБ для рушія Unity
Спеціальні пристрої	не вимагаються (достатньо звичайної клавіатури, миші та монітора)

3.9 Тестування програмного забезпечення

3.9.1 Вибір та обґрунтування методів тестування

Оскільки розроблений інструментальний засіб орієнтований на візуальне конструювання інтерфейсів без написання програмного коду, було обрано функціональне тестування за методом «чорної скриньки». Цей підхід дозволив перевірити коректність реакції графічного інтерфейсу на дії користувача без втручання у внутрішню логіку роботи програми [40]. Для перевірки коректності взаємодії розробленого Ядра зі стандартними підсистемами рушія Unity тестування проводилося на рівні інтеграції для підтвердження працездатності:

- механізмів перехоплення інспекторів та відмальовувачів властивостей;
- підсистеми поліморфної серіалізації даних та збереження макетів;

– режиму зворотної сумісності зі старими користувацькими інтерфейсами.

Окрему увагу було приділено нефункціональному тестуванню, зокрема перевірці продуктивності та надійності системи. Для цього проводилися експерименти, що включали:

– навантажувальне тестування конвеєра генерації під час обробки макетів із великою кількістю вкладених полів та контейнерів;
– перевірку стійкості інструменту до критичних ситуацій та апаратних перевантажень, зокрема тестування захисного алгоритму SafeChangeEventHelper при навмисному створенні конфліктів між математичними модифікаторами.

Враховуючи специфіку візуального інструменту, експерименти проводилися у ручному режимі безпосередньо в середовищі розробки Unity. Від використання автоматизованих модульних тестів було вирішено відмовитися, оскільки вони не здатні об'єктивно імітувати реальну поведінку користувача під час взаємодії зі складним ієрархічним візуальним деревом у панелі інспектора.

Окрім цього також було вирішено провести статичний аналіз, використовуючи офіційний пакета від Unity – Project Auditor.

3.9.2 Статичний аналіз через Project Auditor

Для виконання статичного аналізу було використано новітній пакет від Unity Technologies, а саме Project Auditor. Цей пакет дозволяє виконувати глибинний аналіз коду, проєкту, налаштувань та збірок у контексті Unity. У випадку розроблюваного візуального редактора вплив мав лише аналіз коду. В результаті аналізу пакет виявив 892 рекомендації щодо коду, 249 з яких були позначені як помірні, решта містили лише інформаційний характер. Аналіз поділений на категорії можна побачити на рисунку 3.7.

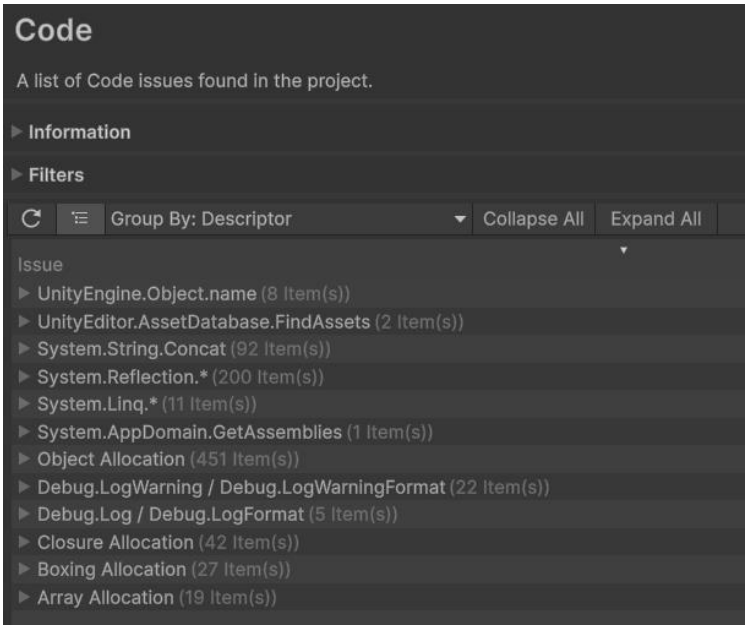


Рисунок 3.7 – Статистичний аналіз коду використовуючи Project Auditor

Майже усі помірні рекомендації стосуються використання рефлексії та інтерполяції рядків. Найбільшу частку рекомендацій складають інформаційні повідомлення щодо виділення пам’яті під час створення об’єктів, насамперед у візуальних елементах та їхніх керуючих лямбда-виразах. Варто врахувати, що більшість цих рекомендацій дійсно є правильними для ігрових застосунків реального часу, проте розроблюване рішення орієнтується на сам редактор, а переважаючі в кількості виклики рефлексії та створення об’єктів виконуються одноразово при створенні структури, а також кешуються при можливості.

3.9.3 Формування вимог та підготовка тестових даних

Для забезпечення об'єктивності та повноти перевірки працездатності розробленого програмного забезпечення було сформовано комплекс вимог до проведення експериментів. Тестові сценарії повинні були мати реальний характер та охоплювати всі ключові етапи конвеєра генерації інтерфейсів, починаючи від аналізу атрибутів коду і закінчуючи відмальовуванням кінцевих графічних елементів.

Оскільки вхідними даними для системи візуального конструювання є безпосередньо структура типів користувача, для проведення тестування було підготовлено набір спеціалізованих тестових скриптів. Підготовлені масиви даних було розділено на дві категорії: «правильні» для перевірки штатного функціонування та «неправильні» або граничні для перевірки стійкості до помилок та виняткових ситуацій.

До категорії правильних вихідних даних увійшли:

- скрипти з базовими примітивними типами (числа, рядки, логічні змінні) та серіалізованими посиланнями на ігрові об'єкти;
- скрипти, що містять складні вкладені структури та класи для перевірки коректності роботи ієрархічних контейнерів;
- компоненти з використанням стандартних декоративних та обмежувальних атрибутів Unity для перевірки роботи LayoutDisassembler.

До категорії неправильних та граничних даних, призначених для тестування захисних механізмів системи, увійшли:

- структури даних із циклічними залежностями для перевірки здатності генератора виявляти та зупиняти нескінченну побудову макета;
- макети зі штучно видаленими або зміненими типами даних (SerializedType) для перевірки відображення попереджень про втрату посилань;
- поля з навмисно призначеними несумісними модифікаторами для перевірки роботи механізму валідації обмежень.

Очікуваним результатом у кожному експерименті була коректна робота інструментального засобу. Відповідно до специфікацій, коректна робота при обробці неправильних вхідних даних передбачала не аварійне завершення роботи редактора Unity, а безпечне перехоплення винятку та виведення відповідного інформаційного повідомлення безпосередньо у панелі інспектора або у консолі рушія.

3.9.4 Валідація та верифікація програмного забезпечення

Після підготовки тестових скриптів та визначення очікуваних показників було проведено серію експериментів у середовищі розробки. Кожен сценарій перевіряв окрему підсистему конвеєра генерації та її здатність обробляти як штатні, так і виняткові ситуації. Стислі результати виконання тестових сценаріїв зведено у таблиці 3.2.

Таблиця 3.2 – Результати виконання тестових сценаріїв

Дія	Реакція програми	Результат
1. Автоматичне розбирання макета. Запуск LayoutDisassembler для скрипта з атрибутами [Header] та [Space].	Генерація LayoutAsset із відповідними вузлами та синхронізація змін.	Частково успішно
2. Обробка вкладених структур. Генерація макета для об'єкта зі складними структурами.	Автоматичне розміщення полів у PropertyFieldNode або FoldoutNode.	Успішно
3. Валідація несумісних модифікаторів. Спроба додати SliderModifier до поля типу string.	Блокування дії через інтерфейс; виведення попередження несумісності.	Успішно
4. Захист від зациклення подій. Одночасне застосування MinModifier(50) та MaxModifier(10).	Фіксація нескінченного циклу та його миттєве блокування системою.	Успішно
5. Захист від рекурсії у макетах. Призначення макета самому собі (поле з [SerializeReference]).	Зупинка побудови дерева; виведення попередження про рекурсивний шаблон.	Успішно
6. Зворотна сумісність (IMGUI Fallback). Відмальовування стандартного типу Bounds.	Створення IMGUIContainer та коректне відображення процедурного інтерфейсу.	Успішно

Детальний аналіз результатів дозволив виділити такі ключові аспекти:

- під час тестування автоматичного розбирання макета у сценарії 1 зафіксовано візуальне дублювання згенерованих модифікаторів поверх стандартних атрибутів Unity, що тимчасово вирішується перемиканням у режим відмальовування Native;
- у перспективі розв'язання проблеми дублювання атрибутів вимагає розділення генератора на два незалежні режими: з повною конвертацією атрибутів (і встановленням режиму Native чи CustomDrawer) та з їхнім ігноруванням (та встановленням режиму Auto);
- система валідації довела свою надійність у сценарії 3, блокуючи створення несумісних комбінацій як на рівні користувацького інтерфейсу, так і через інструмент діагностики при спробі штучної підміни типів;
- спроби створення логічних парадоксів у сценарії 4 підтвердили ефективність захисного алгоритму SafeChangeEventHelper, який успішно зупинив лавинну генерацію подій та запобіг зависанню середовища;
- тестування рекурсивних макетів у сценарії 5 з використанням циклічних посилань довело здатність системи конвеєра вчасно зупиняти нескінченну побудову візуального дерева, проте варто розглянути визначення таких зациклень на етапі конфігурування.

Ознайомитися з тестовими класами та візуальними результатами тестування можна у матеріалах додатку Г.

3.10 Аналіз результатів тестування та висновки до розділу

Проведений аналіз результатів тестування підтверджує, що розроблена архітектура успішно справляється з поставленими завданнями. Найважливішим досягненням є стабільна робота захисних механізмів: інструментальний засіб продемонстрував високу стійкість до критичних помилок користувача, таких як

створення рекурсивних макетів або логічних парадоксів між модифікаторами. Замість аварійного завершення роботи середовища розробки, система безпечно перехоплює винятки, зупиняє циклічні процеси та інформує користувача через локальну систему діагностики. Усі помилки фіксуються та презентуються користувачеві, у більшості випадків як в конфігураторі, так і у інспекторі.

У результаті проведених експериментів виявлено, що розроблений інструментальний засіб є стабільним та готовим до практичного використання. Базова функціональність генерації візуальних інтерфейсів на основі керованих даними макетів реалізована у повній відповідності з архітектурними вимогами та є працездатною. Алгоритми перехоплення інспекторів та режим зворотної сумісності зі старим кодом довели свою ефективність під час інтеграційного тестування. Існують некритичні проблеми, пов'язані з візуальним дублюванням атрибутів під час автоматичного розбирання макетів, проте вони мають чітко визначені шляхи вирішення і не впливають на виконання системою своїх основних задач. Статичний аналіз коду не виявив критичних чи серйозних проблем із кодом програмного забезпечення.

На основі проведеної роботи було створено керівництво користувача, що описує загальні відомості про розроблену система, а також містить покрокову інструкція з роботи над макетами. Готове керівництво користувача можна переглянути у додатку Д.

ВИСНОВКИ

У кваліфікаційній роботі розв'язано актуальне завдання інженерії програмного забезпечення, що полягає у створенні засобу для візуального конструювання користувацьких інтерфейсів. Метою роботи було розроблення інструменту візуального створення редакторів користувача для рушія Unity на базі фреймворку UI Toolkit з метою усунення необхідності рекомпіляції коду під час налаштування відображення. У результаті виконання проєкту розроблено працездатний програмний продукт, що відповідає визначеним функціональним та нефункціональним вимогам.

Середовищем розробки для реалізації програмного забезпечення виступив ігровий рушій Unity версії 6000.3 LTS. Програмна логіка реалізована мовою програмування C# з використанням об'єктно-орієнтованого підходу та архітектурного шаблону MVVM. Як основу для побудови графічного інтерфейсу було застосовано фреймворк UI Toolkit. Для збереження конфігураційних макетів було використано вбудовані об’єкти даних Unity у поєднанні з механізмом поліморфної серіалізації. Для аналізу структури класів та динамічного доступу до несеріалізованих даних було застосовано механізми рефлексії. Обчислення логічних виразів спроєктовано на базі інтеграції з офіційним API Graph Toolkit.

Виконання кваліфікаційної роботи здійснювалося послідовно через етапи дослідження, проєктування, програмної реалізації та тестування. На етапі дослідження предметної області було проведено аналіз наявних засобів розширення редактора Unity, в результаті якого було сформовано перелік функціональних та нефункціональних вимог для реалізації програмної системи, що вирішує архітектурні недоліки існуючих рішень.

На етапі проєктування було розроблено модульну архітектуру програмного продукту, що складається з незалежного ядра, модуля віртуалізації та модуля графу логіки. У ході виконання цього етапу було:

- обрано шаблон MVVM для забезпечення ізоляції візуального представлення від моделей даних;
- спроєктовано механізм глобального перехоплення відмальовування інспекторів на базі шаблону «Стратегія» з підтримкою IMGUI Fallback;
- розроблено структуру зберігання ієрархічних макетів та модифікаторів з декларативною системою обмежень;
- спроєктовано підсистему сесійних проксі-об'єктів для роботи з приватними та статичними даними без порушення цілісності серіалізатора;
- спроєктовано систему динамічного обчислення значень на основі графів.

Готова архітектура забезпечує модульність системи та можливість масштабування функціоналу, без необхідності у зміні вихідного коду системи.

На етапі програмної реалізації та тестування було розроблено вихідний код застосунку та проведено його перевірку на правильність функціонування. При реалізації було виконано наступні роботи:

- створено структури даних та допоміжні інструменти взаємодії;
- реалізовано конвеєр генерації візуального інтерфейсу здатний динамічно будувати дерево графічних елементів;
- розроблено модуль для автоматичної генерації макетів на основі наявних у коді C# атрибутів;
- впроваджено системи перехоплення інспекторів та динамічного перезавантаження представлень при модифікації макетів;
- створено механізми оптимізації та безпечної обробки подій зміни стану для уникнення циклічних зависань.

Для перевірки коректності роботи було проведено статичний аналіз коду за допомогою інструмента Project Auditor та функціональне тестування. В результаті проведених тестів виявлено, що розроблена підсистема є стабільною та забезпечує безпечне перехоплення помилок.

Впровадження розробленого програмного забезпечення надає користувачам наступні переваги:

- дозволяє скоротити затрати часу на верстку та коригування інтерфейсів редактора за рахунок миттєвого відображення змін без рекомпіляції;
- заощаджує ресурси розробників, дозволяючи залучати нетехнічних спеціалістів до налаштування власних робочих просторів;
- підвищує ергономіку та надійність розробки завдяки графічному середовищу з вбудованою системою валідації;
- забезпечує збереження зворотної сумісності з наявним кодом через підсистеми делегування та IMGUI Fallback.

Практичною галуззю застосування розробленого програмного продукту є індустрія розробки комп'ютерних ігор та створення інтерактивних застосунків на рушії Unity. Систему доцільно використовувати у студіях для розробки внутрішнього інструментарію.

Можливими напрямками продовження роботи є розширення функціональних можливостей архітектури, а також покращення вже готового функціоналу на основі здобутих під час розробки технічних знань. До перспективних завдань можна віднести:

- розроблення механізму експорту та імпорту конфігураційних макетів у стандартизовані текстові формати для оптимізації роботи з системами контролю версій;
- інтеграція доступу до глобальних статичних класів у селектор членів та візуальний граф обчислень;
- розширення базової бібліотеки вузлів та модифікаторів для підтримки додаткових типів даних та складних правил компонування інтерфейсу;
- покращення та розширення інтерфейсу роботи з макетами;
- впровадження спроєктованих модулів віртуалізації та графів;
- оптимізація внутрішніх процесів роботи системи.

Отже, розроблена система цілком відповідає поставленим завданням кваліфікаційної роботи та має високий потенціал для подальшого вдосконалення й практичного застосування.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. SerializedObject. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/SerializedObject.html (дата звернення 18.02.2026).

2. The consolidation of game software engineering: A systematic literature review of software engineering for industry-scale computer games / J. Chueca et al. *Information and Software Technology.* 2023. P. 107330. URL: https://doi.org/10.1016/j.infsof.2023.107330

3. Comparison of UI systems in Unity. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/UI-system-compare.html (дата звернення 18.02.2026).

4. IMGUI (Immediate Mode GUI). *Unity Documentation. Manual.* https://docs.unity3d.com/6000.3/Documentation/Manual/GUIScriptingGuide.html (дата звернення: 18.02.2026).

5. Introduction to UI Toolkit. *Unity Documentation. Manual.* https://docs.unity3d.com/6000.3/Documentation/Manual/ui-systems/introduction-ui-toolkit.html (дата звернення: 18.02.2026).

6. Unity UI Toolkit vs UGUI: 2025 Developer Guide. *Angry Shark Studio.* 18.09.2025. URL: https://www.angry-shark-studio.com/blog/unity-ui-toolkit-vs-ugui-2025-guide/ (дата звернення: 18.02.2026).

7. Data Binding. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/best-practice-guides/ui-toolkit-for-advanced-unity-developers/data-binding.html (дата звернення: 18.02.2026).

8. ДСТУ ISO/IEC 25010:2025 Інженерія систем і програмних засобів. Вимоги до якості систем і програмних засобів та її оцінювання (SQuaRE). Дата початку дії: 01.12.2025. Дата прийняття: 18.07.2025.

9. *Odin Inspector.* URL: https://odininspector.com/ (дата звернення: 18.02.2026).

10. Alchemy. *Annulus Games.* URL: https://annulusgames.github.io/Alchemy/articles/en/about.html (дата звернення: 18.02.2026).

11. artificetoolkit. *Github. AbZorbaGames.* URL: https://github.com/AbZorbaGames/artificetoolkit (дата звернення: 18.02.2026).

12. *Vibe: Visual Inspector Builder Documentation.* URL: https://superstatic.gitbook.io/vibe-documentation/ (дата звернення: 19.02.2026).

13. UI Builder. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/UIBuilder.html (дата звернення: 19.02.2026).
14. Kok B. Custom Editor with UI Toolkit. Beginning Unity Editor Scripting. Berkeley, CA, 2021. P. 71–110. URL: https://doi.org/10.1007/978-1-4842-7167-4_4
15. Necula, S. Exploring the model-view-controller (mvc) architecture: A broad analysis of market and technological applications. 2024. URL: https://doi.org/10.20944/preprints202404.1860.v1
16. Li D. D., Liu X. Y. Research on MVP Design Pattern Modeling Based on MDA. *Procedia Computer Science.* 2020. Vol. 166. P. 51–56. URL: https://doi.org/10.1016/j.procs.2020.02.012
17. Fuksa M., Speth S., Becker S. MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern. *Lecture Notes in Computer Science.* Cham, 2025. P. 163–181. URL: https://doi.org/10.1007/978-3-031-78338-8_9
18. Hu C. Introduction to Software Design: Concepts, Principles, Methodologies, and Techniques. Springer International Publishing AG, 2023.
19. Afteni, D., Poştaru, A. Impact of design patterns on maintainability in complex software systems. *In Conferinţa tehnico-ştiinţifică a studenţilor, masteranzilor şi doctoranzilor.* 2025. Vol. 1. P. 393-39.
20. PropertyDrawer. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/PropertyDrawer.html (дата звернення 17.03.2026).
21. Undo. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Undo.html (дата звернення 17.03.2026).
22. Scriptable Object. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/class-ScriptableObject.html (дата звернення 18.03.2026).
23. SerializeReference. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/SerializeReference.html (дата звернення 18.03.2026).
24. Type.AssemblyQualifiedName Property. *Microsoft Learn.* URL: https://learn.microsoft.com/dotnet/api/system.type.assemblyqualifiedname (дата звернення 23.03.2026).
25. ISerializationCallbackReceiver. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/ISerializationCallbackReceiver.html (дата звернення 23.03.2026).
26. Lam, C. K., Parreaux, L. Being lazy when it counts: Practical constant-time memory management for functional programming. *In International Symposium on Functional and Logic Programming.* 2024. P. 188-216. Singapore: Springer Nature Singapore.
27. Felisberto, M. The trade-offs between monolithic vs. distributed

architectures. 2024.
28. Introduction to assemblies in Unity. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/assembly-definitions-intro.html (дата звернення 25.03.2026).
29. Object. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/class-Object.html (дата звернення 25.03.2026).
30. Serialization rules. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/script-serialization-rules.html (дата звернення 26.03.2026).
31. Editor. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Editor.html (дата звернення 26.03.2026).
32. Introduction to Graph Toolkit. *Unity Documentation. Graph Tookit.* URL: https://docs.unity3d.com/Packages/com.unity.graphtoolkit@0.4/manual/introduction.html (дата звернення 27.03.2026).
33. TypeCache. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/TypeCache.html (дата звернення 13.04.2026).
34. Ramachandrappa, N. C. Solid design principles in software engineering. *International Journal of Computer Trends and Technology,* 2024. V. 72.
35. PropertyField. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/UIE-uxml-element-PropertyField.html (дата звернення 16.04.2026).
36. Event reference. *Unity Documentation. Manual.* URL: https://docs.unity3d.com/6000.3/Documentation/Manual/UIE-Events-Reference.html (дата звернення 17.04.2026).
37. InitializeOnLoadAttribute. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/InitializeOnLoadAttribute.html (дата звернення 20.04.2026).
38. EditorApplication.update. *Unity Documentation. Scripting API.* URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/EditorApplication-update.html (дата звернення 20.04.2026).
39. Reflection And Attributes. *Microsoft. Learn .NET C#.* URL: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/ (дата звернення 24.04.2026).
40. Maspupah A. LITERATURE REVIEW: ADVANTAGES AND DISADVANTAGES OF BLACK BOX AND WHITE BOX TESTING METHODS. *Jurnal Techno Nusa Mandiri.* 2024. Vol. 21, no. 2. P. 151–162. URL: https://doi.org/10.33480/techno.v21i2.5776
41. Project Auditor Introduction. *Unity Documentation. Project Auditor.* URL: https://docs.unity3d.com/Packages/com.unity.project-auditor@2.0/manual/project-auditor-introduction.html (дата звернення 28.04.2026).

ДОДАТОК А

(обов’язковий)

ТЕХНІЧНЕ ЗАВДАННЯ

на розробку комерційного програмного продукту «Інструмент візуального створення редакторів користувача для рушія Unity»

1. Загальні відомості

Цей документ є технічним завданням на розробку комерційного програмного продукту (плагіна), призначеного для розширення функціональних можливостей редактора Unity. Повна назва програмного засобу: «Інструмент візуального створення редакторів користувача для рушія Unity».

Продукт орієнтований на дві основні групи користувачів із різним рівнем технічної підготовки:

- перша група включає інженерів-програмістів, які прагнуть мінімізувати час на написання рутинного коду для верстки інтерфейсів та фокусуватися на бізнес-логіці гри;
- друга група складається з нетехнічних спеціалістів, зокрема геймдизайнерів, яким необхідна можливість прямого керування макетами інспекторів та їхньої візуальної адаптації без залучення програмістів.

Програмний засіб розповсюджуватиметься за платною ліцензією через офіційний магазин Unity Asset Store.

2. Призначення та цілі створення програмного продукту

Головним призначенням продукту є забезпечення розробників багатофункціональним візуальним конструктором інспекторів у середовищі Unity, що повністю усуває необхідність рекомпіляції коду під час ітеративного налаштування користувацьких інтерфейсів (Custom Editors).

Основні цілі розробки включають:

- впровадження no-code підходу до створення розширень редактора Unity;
- скорочення ітераційного циклу розробки внутрішнього інструментарію завдяки миттєвому оновленню інтерфейсів;
- зниження порогу входження для роботи зі складними структурами даних та їхнім відображенням;
- забезпечення високого рівня розширюваності, щоб інші програмісти могли легко додавати власні модулі до системи;
- створення стабільного, конкурентоспроможного та привабливого продукту для успішної монетизації на платформі Unity Asset Store.

3. Вимоги до функціональних характеристик

3.1. Візуальне проєктування та керування макетами

Підсистема візуального проєктування повинна надавати користувачам інтуїтивно зрозумілий інтерфейс для формування структури інспектора без необхідності написання C#-коду. Збереження створених макетів має відбуватися у вигляді незалежних конфігураційних файлів на базі ScriptableObject, які зчитуються в режимі реального часу та можуть легко переноситися між різними проєктами, за умови існування ідентичних цільових класів.

Для забезпечення процесу конструювання система повинна підтримувати:

- формування ієрархії графічних компонентів за допомогою конфігуратора візуального редактора;
- логічне групування елементів за допомогою горизонтальних і вертикальних панелей, вкладок та блоків згортання;
- гнучке налаштування візуальних атрибутів, стилів та декораторів для кожного окремого поля.

Окрім структурної побудови, система має забезпечувати гнучке керування джерелами даних для елементів інтерфейсу. Користувач повинен мати змогу отримувати дані шляхом:

- ручного введення статичних значень безпосередньо в конструкторі;
- використання автоматичної двосторонньої об'єктної прив'язки до членів цільового компонента;
- генерування значення динамічно за допомогою візуального графа логіки.

3.2. Обробка логіки та візуальний граф обчислень

Програмний продукт має містити підсистему візуального графа логіки для забезпечення динамічної поведінки згенерованих інтерфейсів. Цей модуль повинен дозволяти користувачам налаштовувати складну логіку обробки даних цільових об'єктів без написання скриптів.

Для забезпечення достатньої гнучкості динамічного керування інтерфейсом граф логіки має підтримувати:

- обробку числових типів даних (int, float) для виконання математичних операцій над параметрами об'єкта;
- роботу з рядковими значеннями (string), включаючи конкатенацію для формування динамічних інформаційних повідомлень або підписів у редакторі;
- обчислення логічних виразів (bool) для налаштування умовного відображення, приховування або блокування певних елементів чи груп інтерфейсу;
- безпечну передачу обчислених значень назад до ядра генерації для миттєвого оновлення стану візуального дерева.

3.3. Віртуалізація несеріалізованих даних

Окрім маніпуляцій зі стандартними серіалізованими властивостями, інструмент має вирішувати проблему доступу до прихованої інформації компонентів. Для забезпечення повноцінного керування станом ігрового об'єкта система повинна вміти працювати з інформацією, що не відслідковується стандартним серіалізатором Unity.

Підсистема віртуалізації повинна гарантувати:

- візуалізацію та редагування приватних полів;

- відображення та редагування статичних членів класу;
- відображення обчислювальних властивостей чи результатів виконання методів екземпляру;
- забезпечення взаємодії з цими даними через механізм динамічного створення тимчасових серіалізованих проксі-об'єктів.

3.4. Зворотна сумісність та підтримка зовнішніх відмальовувачів

Для збереження гнучкості системи та її безшовної інтеграції з існуючими проєктами користувачів, програмний засіб має взаємодіяти зі стандартними механізмами рушія Unity та вже написаним користувацьким кодом.

Підсистема зворотної сумісності повинна забезпечувати:

- підтримку перегляду, вибору та використання існуючих у проєкті користувацьких скриптових відмальовувачів (Property Drawers);
- наявність спеціалізованого механізму-містка для інтеграції стандартних відмальовувачів старого зразка (IMGUI) у нову візуальну систему;
- можливість примусового виклику стандартного інспектора Unity для визначених типів об'єктів, повністю ігноруючи будь-які створені для них макети;
- підтримку делегування процесів відмальовування іншим встановленим фреймворкам чи користувацьким редакторам.

4. Вимоги до нефункціональних характеристик

4.1. Сумісність, дистрибуція та продуктивність

Оскільки інструмент поширюватиметься на комерційній основі, він має відповідати суворим стандартам екосистеми Unity щодо продуктивності та зручності інтеграції в робочий процес студій.

Система повинна відповідати наступним критеріям:

- підтримка актуальних версій ігрового рушія, починаючи з Unity 6000.3 LTS;

- використання сучасного фреймворку UI Toolkit для мінімізації навантаження на процесор під час проєктування та відмальовування інспекторів;
- забезпечення миттєвого оновлення візуального шаблону під час його редагування із затримкою не більше 300 мілісекунд;
- реалізація агресивного кешування викликів рефлексії для уникнення падіння частоти кадрів у редакторі Unity.

4.2. Надійність та обробка помилок

Будь-які збої у роботі інструменту не повинні призводити до аварійного завершення роботи середовища розробки або пошкодження ігрових даних користувача (префабів, сцен, асетів).

Підсистема надійності повинна забезпечувати:

- автоматичну валідацію типів даних на етапі створення зв'язків у макеті чи графі логіки;
- безпечну ізоляцію винятків, при якій помилка в генерації одного користувацького інспектора не ламає роботу всього вікна редактора або ігрового рушія в цілому;
- інформативне логування помилок у консоль Unity (із використанням Debug.LogWarning або Debug.LogError) зі зрозумілими для розробника повідомленнями та можливими шляхами вирішення проблеми;
- механізми збереження цілісності макетів при рефакторингу коду користувачем (наприклад, при зміні назв полів або класів) із відповідним візуальним сповіщенням про втрачені посилання без знищення конфігурації.

4.3. Ергономіка

Інструмент має бути розроблений із фокусом на природну інтеграцію у звичний робочий процес користувачів редактора Unity. Ергономіка системи повинна забезпечувати мінімальний час на навчання та комфортну щоденну роботу. Для досягнення високого рівня UX/UI необхідно дотримуватися наступних умов:

- візуальне оформлення всіх вікон та елементів управління має повністю відповідати поточній колірній темі та стилістиці Unity за допомогою вбудованих стилів USS;
- повноцінна та глибока інтеграція зі стандартною системою скасування та повторення дій (Undo/Redo), що працює як для конструювання макетів, так і для зміни значень цільових об'єктів;
- використання інтуїтивних механізмів взаємодії користувача із продуктом, щоб уникнути неочевидних рішень.

5. Вимоги до архітектури системи та її розширюваності

Базовим архітектурним рішенням для розроблюваного програмного забезпечення має бути використання шаблону MVVM (Model-View-ViewModel), що ідеально інтегрується з фреймворком UI Toolkit та забезпечує двосторонню прив'язку даних за замовчуванням.

Для забезпечення високої відмовостійкості та розширюваності системи необхідно застосувати принцип жорсткої ізольованої модульності з використанням механізму збірок Assembly Definitions. Архітектура має складатися з наступних ізольованих підсистем:

- «Ядро» керує процесом зчитування макетів, обробляє серіалізовані властивості та надає абстрактні інтерфейси для підключення інших модулів;
- «Модуль віртуалізації» ізольовано відповідає за аналіз і тимчасове збереження несеріалізованих даних;

– «Модуль графу логіки» забезпечує обчислення виразів на базі відокремлених графових API на базі Graph Toolkit.

Система повинна проєктуватися як відкрита. Наданий публічний API має дозволяти стороннім розробникам створювати власні модифікатори для макетів та додавати нові типи операторів у візуальний граф без модифікації вихідного коду самого плагіна.

Для забезпечення гнучкості взаємодії між компонентами мають бути застосовані додаткові шаблони проєктування:

– «Фабричний метод» для динамічної генерації графічних вузлів;
– «Сесійний проксі-об'єкт» для обгортання несеріалізованих змінних;
– «Стратегія» для динамічного вибору алгоритми відмальовування інспекторів на основі встановленої конфігурації;
– «Одинак» для управління глобальним реєстром конфігурацій макетів.

6. Вимоги до структур даних та збереження конфігурацій

Базовим контейнером для збереження візуальних макетів має виступати вбудований клас ScriptableObject, що гарантує безшовну інтеграцію конфігурацій у конвеєр ресурсів Unity.

Обробка неоднорідних структурних блоків у межах конфігурації має базуватися на таких вимогах:

– застосування поліморфної серіалізації (атрибут SerializeReference) для збереження масиву різнотипних вузлів та інструкцій;
– використання спеціалізованого класу обгортки (SerializedType) для надійного збереження посилань на типи даних у користувацькому коді;
– реалізація механізмів відкладеного «лінивого» кешування посилань для зниження ресурсомісткості системи.

7. Вимоги до інтерфейсу користувача

Для ефективної розробки інструментів необхідно реалізувати на базі UI Toolkit три основні спеціалізовані робочі простори.

Вікно пошуку типів (TypeSelector). Інтерфейс із системою автоматичного сканування збірок та пріоритезації результатів пошуку, що позбавляє користувача необхідності вводити текстові імена класів вручну.

Вікно конструювальника макетів. Модифікований стандартний інспектор Unity, призначений для редагування макетів.

Вікно редактора графу логіки. Вузлове середовище для створення зв'язків між операторами та даними, яке включає механізми автоматичної валідації схем на наявність замкнених циклів чи конфліктів типів.

8. Вимоги до документації та підтримки

Для успішного розміщення на Unity Asset Store та забезпечення якісного користувацького досвіду, продукт має супроводжуватися комплексною документацією англійською мовою:

- посібник користувача (User Manual): ілюстрований документ або веб-сайт із покроковими інструкціями щодо налаштування першого редактора, роботи з графом та використання віртуалізації;
- документація розробника (API Reference): технічний опис публічних інтерфейсів та абстрактних класів для інженерів, які бажають розширювати функціонал плагіна;
- файл README: базова інструкція зі встановлення пакета, налаштування залежностей та посиланнями на ресурси підтримки;
- демонстраційні матеріали: наявність тестових класів та макетів у складі пакета, що демонструють базові та просунуті сценарії використання продукту.

9. Етапи розробки та випуску

Процес створення інструмента поділяється на наступні етапи.

Етап 1, прототипування та MVP. Розроблення базового ядра генерації (Core), реалізація макетів на основі ScriptableObject та створення спрощеного вікна конструювальника макетів.

Етап 2, розширення функціоналу. Розгляд доцільності розроблення модулів віртуалізації та підсистеми графа логіки, розроблення їхніх примітивних версій.

Етап 3, тестування. Внутрішнє тестування на реальних проєктах, оптимізація продуктивності (кешування рефлексії), відловлювання помилок та тестування зворотної сумісності зі старими відмальовувачами.

Етап 4, підготовка до релізу. Написання технічної документації, створення демо об’єктів та підготовка коду до випуску.

Етап 5, реліз. Подання пакета на розгляд до модераторів Unity Asset Store, проходження перевірки та подальший супровід продукту (виправлення багів, оновлення під нові LTS-версії Unity).

ДОДАТОК Б

(обов’язковий)

ТАБЛИЦІ РЕАЛІЗОВАНИХ ВУЗЛІВ ТА МОДИФІКАТОРІВ

Таблиця Б.1 – Перелік реалізованих вузлів

Назва вузла	**Короткий опис**	**Ключові можливості**
DefaultEditor	Відображає стандартний інспектор або типове поле властивості для об'єкта.	SupportsLabel, SupportsTooltip, SupportsReadOnly, SupportsVisibility
Foldout	Контейнер із можливістю згортання та розгортання вкладених елементів.	IsContainer, SupportsLabel, SupportsVisibility
Group	Контейнер для горизонтального або вертикального групування елементів.	IsContainer, SupportsVisibility
Label	Декоративна текстова мітка для логічного розділення блоків.	SupportsLabel, SupportsVisibility
MethodInvoke	Візуальний блок із кнопкою для виклику методу цільового об'єкта та полями параметрів.	SupportsLabel, SupportsTooltip, SupportsVisibility
PropertyField	Базове поле для відображення та редагування властивості об'єкта або обчисленого значення.	HasValue, SupportsLabel, SupportsTooltip, SupportsReadOnly, SupportsVisibility
ScriptField	Поле, що відображає посилання на поточний скрипт (недоступне для редагування).	SupportsLabel, SupportsTooltip, SupportsReadOnly, SupportsVisibility

Продовження таблиці Б.1

Назва вузла	**Короткий опис**	**Ключові можливості**
Separator	Горизонтальна розділювальна лінія для покращення читабельності інтерфейсу.	SupportsVisibility
SpaceNode	Порожній простір із можливістю налаштування висоти відступу.	SupportsVisibility
TabGroup	Контейнер для організації вкладених елементів у вигляді сторінок-вкладок	IsContainer, SupportsVisibility

Таблиця Б.2 – Перелік реалізованих структурних і декоративних модифікаторів

Назва модифікатора	**Короткий опис**	**Вимоги до сумісності**
BackgroundColor (дек.)	Змінює колір фону графічного елемента.	Немає вимог
FontSize (дек.)	Встановлює розмір шрифту для вузла та його дочірніх елементів.	SupportsLabel
Label (дек.)	Дозволяє перевизначити текст мітки вузла, приховати її або використати динамічну назву властивості.	SupportsLabel
Max (дек.)	Жорстко обмежує максимальне допустиме значення для числового поля.	HasValue (тільки типи int, float)
Min (дек.)	Жорстко обмежує мінімальне допустиме значення для числового поля.	HasValue (тільки типи int, float)
ReadOnly (дек.)	Блокує можливість редагування поля на основі динамічної або статичної логічної умови.	SupportsReadOnly
Slider (структ.)	Структурно перетворює стандартне числове поле вводу на інтерактивний повзунок із заданим діапазоном.	HasValue (тільки типи int, float)

Продовження таблиці Б.2

Назва модифікатора	**Короткий опис**	**Вимоги до сумісності**
TextArea (структ.)	Структурно перетворює рядкове поле на багаторядкову текстову область.	HasValue (тільки тип string)
TextColor (дек.)	Змінює колір тексту вузла та його дочірніх елементів.	SupportsLabel
TextStyle (дек.)	Додає стилістичні прапорці (жирний, курсив) до тексту мітки вузла.	SupportsLabel
Tooltip (дек.)	Додає спливаючу підказку при наведенні курсора на елемент.	SupportsTooltip
Visibility (дек.)	Динамічно приховує або показує елемент в інспекторі на основі заданої логічної умови.	SupportsVisibility

ДОДАТОК В

(обов’язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ СКЛАДОВИХ СИСТЕМИ

ModuleRegistry.cs:

```
using System;

namespace VisualEditor.Editor
{
    /// <summary>
    /// Static service locator for optional module providers.
    /// Initialized with no-op defaults. Future modules register their
implementations
    /// at editor startup via <see cref="SetVirtualizationProvider"/> and <see
cref="SetGraphProvider"/>.
    /// </summary>
    public static class ModuleRegistry
    {
        private static IVirtualizationProvider virtualization = new
NoOpVirtualizationProvider();
        private static IGraphProvider graph = new NoOpGraphProvider();

        /// <summary>
        /// The active Virtualization provider. Never null — returns <see
cref="NoOpVirtualizationProvider"/> when module is absent.
        /// </summary>
        public static IVirtualizationProvider Virtualization => virtualization;

        /// <summary>
        /// The active Graph provider. Never null — returns <see
cref="NoOpGraphProvider"/> when module is absent.
        /// </summary>
        public static IGraphProvider Graph => graph;

        /// <summary>
        /// Registers a Virtualization module implementation. Call at editor startup.
        /// </summary>
        public static void SetVirtualizationProvider(IVirtualizationProvider
provider)
        {
            virtualization = provider ?? throw new
ArgumentNullException(nameof(provider));
        }

        /// <summary>
        /// Registers a Graph module implementation. Call at editor startup.
        /// </summary>
        public static void SetGraphProvider(IGraphProvider provider)
        {
            graph = provider ?? throw new ArgumentNullException(nameof(provider));
        }

        /// <summary>
        /// Resets both providers to their no-op defaults. Primarily for testing.
        /// </summary>
```

```
        internal static void Reset()
        {
            virtualization = new NoOpVirtualizationProvider();
            graph = new NoOpGraphProvider();
        }
    }
}
```

IVirtualizationProvider.cs:

```
using System;
using UnityEditor;

namespace VisualEditor.Editor
{
    /// <summary>
    /// Contract for the Virtualization module.
    /// Supplies proxy <see cref="SerializedProperty"/> bindings for:
    /// <list type="bullet">
    ///   <item>Non-serialized members of the target object (ObjectBound
mode).</item>
    ///   <item>Standalone manual values not tied to any object member (Manual
mode).</item>
    ///   <item>Method parameters that need editable proxies.</item>
    /// </list>
    /// </summary>
    public interface IVirtualizationProvider
    {
        /// <summary>
        /// Returns true if this provider can create a proxy binding for the given
member on the target.
        /// </summary>
        bool CanProvideMemberBinding(UnityEngine.Object target, MemberPath
memberPath);

        /// <summary>
        /// Returns a proxy <see cref="SerializedProperty"/> wrapping a non-
serialized member.
        /// </summary>
        SerializedProperty GetMemberProxy(UnityEngine.Object target, MemberPath
memberPath);

        /// <summary>
        /// Creates a standalone proxy <see cref="SerializedProperty"/> for a manual
value
        /// not tied to any member of the target. Used by Manual data source mode.
        /// The <paramref name="identifier"/> is the node's <c>SerializeReference</c>
        /// <c>managedReferenceId</c> — stable, unique per node instance, and already
assigned by Unity.
        /// </summary>
        SerializedProperty GetManualProxy(UnityEngine.Object owner, Type valueType,
long identifier);

        /// <summary>
        /// Whether the Virtualization module is currently active and operational.
        /// </summary>
        bool IsActive { get; }

        /// <summary>
        /// Cleans up all proxy objects associated with the given owner when the
inspector is destroyed.
```

```
        /// The owner is the <see cref="UnityEngine.Object"/> that holds the current
property
        /// or is being inspected.
        /// </summary>
        void ReleaseProxies(UnityEngine.Object owner);
    }
}
```

IGraphProvider.cs:

```
namespace VisualEditor.Editor
{
    /// <summary>
    /// Contract for the Graph module.
    /// Evaluates visual graph assets and returns typed primitive results.
    /// Core calls this to resolve dynamic modifier values, visibility conditions,
etc.
    /// </summary>
    public interface IGraphProvider
    {
        /// <summary>
        /// Returns true if this provider can evaluate the given graph asset.
        /// </summary>
        bool CanEvaluate(IGraphAsset graphAsset);

        /// <summary>
        /// Evaluates the graph asset against the target object and returns a typed
result.
        /// The caller specifies the expected return type via <typeparamref
name="T"/>.
        /// </summary>
        GraphResult<T> Evaluate<T>(IGraphAsset graphAsset, object targetObject);

        /// <summary>
        /// Whether the Graph module is currently active and operational.
        /// </summary>
        bool IsActive { get; }
    }

    /// <summary>
    /// Generic result from graph evaluation. Avoids boxing of primitive values.
    /// </summary>
    public struct GraphResult<T>
    {
        public bool success;
        public T value;
        public string error;

        public static GraphResult<T> Failure(string error) => new()
        {
            success = false,
            value = default,
            error = error
        };

        public static GraphResult<T> Success(T value) => new()
        {
            success = true,
            value = value,
            error = null
```

```
        };
    }
}
```

SerializedType.cs:

```
using System;
using UnityEngine;

namespace VisualEditor
{
    /// <summary>
    /// Serializable wrapper for <see cref="System.Type"/> with implicit conversion
operators.
    /// Safely handles missing types by returning null and preserving error
information.
    /// </summary>
    [Serializable]
    public class SerializedType : ISerializationCallbackReceiver
    {
        [SerializeField] private string assemblyQualifiedName = string.Empty;

        private Type cachedType;
        private bool isResolved;

        public SerializedType() { }

        public SerializedType(Type type)
        {
            assemblyQualifiedName = type != null ? type.AssemblyQualifiedName :
string.Empty;
            cachedType = type;
            isResolved = true;
        }

        public Type Type
        {
            get
            {
                if (!isResolved)
                    Resolve();
                return cachedType;
            }
        }

        public bool HasSavedType => !string.IsNullOrEmpty(assemblyQualifiedName);

        public bool IsTypeMissing => HasSavedType && Type == null;

        public string SavedTypeName => assemblyQualifiedName;

        public string SavedShortName
        {
            get
            {
                if (string.IsNullOrEmpty(assemblyQualifiedName))
                    return string.Empty;

                int commaIndex = assemblyQualifiedName.IndexOf(',');
                string fullName = commaIndex >= 0
```

```
            ? assemblyQualifiedName.Substring(0, commaIndex)
            : assemblyQualifiedName;

        int dotIndex = fullName.LastIndexOf('.');
        return dotIndex >= 0 ? fullName.Substring(dotIndex + 1) : fullName;
    }
}

public static implicit operator Type(SerializedType serializedType)
{
    return serializedType != null ? serializedType.Type : null;
}

public static implicit operator SerializedType(Type type)
{
    return new SerializedType(type);
}

public void OnBeforeSerialize() { }

public void OnAfterDeserialize()
{
    cachedType = null;
    isResolved = false;
}

private void Resolve()
{
    isResolved = true;

    if (string.IsNullOrEmpty(assemblyQualifiedName))
    {
        cachedType = null;
        return;
    }

    try
    {
        cachedType = Type.GetType(assemblyQualifiedName);
    }
    catch
    {
        cachedType = null;
    }
}

/// <summary>
/// Returns a clean display name for a type, handling generic notation.
/// </summary>
public static string GetCleanTypeName(Type type)
{
    if (type == null)
        return "None";

    string name = type.Name;
    int backtickIndex = name.IndexOf('`');
    if (backtickIndex < 0)
        return name;

    var genericArgs = type.GetGenericArguments();
    if (type.IsGenericTypeDefinition || genericArgs.Length == 0)
    {
        // Open generic: show List<> or Dictionary<,>
        int argCount = genericArgs.Length;
```

```
                string commas = argCount > 1 ? new string(',', argCount - 1) :
string.Empty;
                return name.Substring(0, backtickIndex) + "<" + commas + ">";
            }

            // Closed generic: show List<Int32> or Dictionary<String,Int32>
            var argNames = new string[genericArgs.Length];
            for (int i = 0; i < genericArgs.Length; i++)
                argNames[i] = GetCleanTypeName(genericArgs[i]);
            return name.Substring(0, backtickIndex) + "<" + string.Join(", ",
argNames) + ">";
        }

        public override string ToString()
        {
            var type = Type;
            if (type != null)
                return GetCleanTypeName(type);
            if (HasSavedType)
                return "Missing: " + SavedShortName;
            return "None";
        }

        public override bool Equals(object obj)
        {
            if (obj is SerializedType other)
                return string.Equals(assemblyQualifiedName,
other.assemblyQualifiedName);
            if (obj is Type type)
                return Type == type;
            return false;
        }

        public override int GetHashCode()
        {
            return assemblyQualifiedName != null ?
assemblyQualifiedName.GetHashCode() : 0;
        }
    }
}
```

MemberPath.cs:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using UnityEngine;

namespace VisualEditor
{
    /// <summary>
    /// Serializable reference to a member (field, property, or method) via a dot-
separated path.
    /// </summary>
    [Serializable]
    public class MemberPath : ISerializationCallbackReceiver
    {
        // For static cache
        private readonly struct TypeStringKey : IEquatable<TypeStringKey>
```

```
        {
            public readonly Type type;
            public readonly string stringValue;
            public TypeStringKey(Type type, string stringValue) { this.type = type;
this.stringValue = stringValue; }
            public bool Equals(TypeStringKey other) => type == other.type &&
stringValue == other.stringValue;
            public override bool Equals(object obj) => obj is TypeStringKey k &&
Equals(k);
            public override int GetHashCode() => HashCode.Combine(type, stringValue);
        }

        private struct ChainStep
        {
            public MemberInfo member;
            public int arrayIndex;
        }

        // ── Serialized state ──
        [SerializeField] private SerializedType rootType = new();
        [SerializeField] private string path = string.Empty;

        // ── Cached resolution state ──
        private ChainStep[] resolvedChain;
        private Type resolvedType;
        private bool isResolved;

        private static readonly Dictionary<TypeStringKey, MemberInfo> memberCache =
new();

        private static readonly Dictionary<TypeStringKey, MemberPath> memberPathCache
= new();

        public MemberPath() { }

        public MemberPath(SerializedType rootType, string path)
        {
            this.rootType = rootType;
            this.path = path;
        }

        /// <summary>
        /// Creates a runtime-only <see cref="MemberPath"/> resolved against a known
<see cref="SerializedType"/>.
        /// Uses cache if possible.
        /// </summary>
        public static MemberPath FromRootType(SerializedType rootType, string path)
        {
            if (rootType == null) throw new ArgumentNullException(nameof(rootType));

            var key = new TypeStringKey(rootType, path);
            if (!memberPathCache.TryGetValue(key, out var memberPath))
            {
                memberPath = new(rootType, path ?? string.Empty);
                memberPathCache[key] = memberPath;
            }
            return memberPath;
        }

        /// <summary>
        /// Creates a runtime-only <see cref="MemberPath"/> resolved against a known
<see cref="SerializedType"/>.
        /// </summary>
```

```
        public static MemberPath FromRootType(Type type, string rootPath, string
path)
        {
            var fullPath = string.IsNullOrEmpty(rootPath) ? path :
$"{rootPath}.{path}";
            return FromRootType(type, fullPath);
        }

        public SerializedType RootType => rootType;

        public string Path => path;

        public bool IsBroken => HasPath && !IsValid;

        public bool IsValid
        {
            get
            {
                if (!isResolved)
                    Resolve();
                return resolvedChain != null;
            }
        }

        public Type ResolvedType
        {
            get
            {
                if (!isResolved)
                    Resolve();
                return resolvedType;
            }
        }

        public MemberInfo FinalMember
        {
            get
            {
                if (!isResolved)
                    Resolve();
                return resolvedChain != null && resolvedChain.Length > 0
                    ? resolvedChain[^1].member
                    : null;
            }
        }

        /// <summary>
        /// Traverses the member chain on the given root object and returns the
value.
        /// Returns error if any intermediate is null or the path is invalid.
        /// </summary>
        public ResolveResult TryGetValue(object root)
        {
            if (!IsValid)
                return ResolveResult.Fail($"Invalid member path: {path}");

            try
            {
                object current = root;
                foreach (var step in resolvedChain)
                {
                    if (current == null)
                        return ResolveResult.Fail("Null intermediate in member
chain");
```

```
                    current = step.member switch
                    {
                        FieldInfo fi   => fi.GetValue(current),
                        PropertyInfo pi => pi.GetValue(current),
                        MethodInfo mi  => mi.Invoke(current, null),
                        _ => throw new InvalidOperationException(
                                 $"Unsupported member type:
{step.member.MemberType}")
                    };

                    // Array-index step: navigate into the IList element
                    if (step.arrayIndex >= 0)
                    {
                        if (current is not IList list)
                            return ResolveResult.Fail(
                                $"Member '{step.member.Name}' does not implement
IList.");
                        if (step.arrayIndex >= list.Count)
                            return ResolveResult.Fail(
                                $"Index {step.arrayIndex} out of bounds for
'{step.member.Name}'.");
                        current = list[step.arrayIndex];
                    }
                }

                return ResolveResult.Success(current);
            }
            catch (Exception e)
            {
                return ResolveResult.Fail(e.Message);
            }
        }

        /// <summary>
        /// Sets the value at the end of the member chain. Only works if the final
member is writable.
        /// </summary>
        public ResolveResult TrySetValue(object root, object value)
        {
            if (!IsValid || resolvedChain.Length == 0)
                return ResolveResult.Fail($"Invalid member path: {path}");

            try
            {
                object current = root;
                for (int i = 0; i < resolvedChain.Length - 1; i++)
                {
                    if (current == null)
                        return ResolveResult.Fail("Null intermediate in member
chain");

                    var step = resolvedChain[i];
                    current = step.member switch
                    {
                        FieldInfo fi   => fi.GetValue(current),
                        PropertyInfo pi => pi.GetValue(current),
                        _ => throw new InvalidOperationException("Cannot traverse
through method member")
                    };

                    if (step.arrayIndex >= 0)
                    {
                        if (current is not IList list)
```

```
                            return ResolveResult.Fail(
                                $"Member '{step.member.Name}' does not implement
IList.");
                        if (step.arrayIndex >= list.Count)
                            return ResolveResult.Fail(
                                $"Index {step.arrayIndex} out of bounds for
'{step.member.Name}'.");
                        current = list[step.arrayIndex];
                    }
                }

                var lastStep = resolvedChain[^1];
                switch (lastStep.member)
                {
                    case FieldInfo fi:
                        fi.SetValue(current, value);
                        break;
                    case PropertyInfo pi when pi.CanWrite:
                        pi.SetValue(current, value);
                        break;
                    default:
                        return ResolveResult.Fail("Final member is not writable");
                }

                return ResolveResult.Success(value);
            }
            catch (Exception e)
            {
                return ResolveResult.Fail(e.Message);
            }
        }

        public void OnBeforeSerialize() { }

        public void OnAfterDeserialize()
        {
            isResolved = false;
            resolvedChain = null;
            resolvedType = null;
        }

        public override string ToString()
        {
            if (!HasPath)
                return "None";
            if (IsValid)
                return path;
            return $"Missing ({path})";
        }

        private void Resolve()
        {
            isResolved = true;
            resolvedChain = null;
            resolvedType = null;

            Type root = rootType?.Type;
            if (root == null || string.IsNullOrEmpty(path))
                return;

            string[] segments = SplitPath(path);
            var chain = new ChainStep[segments.Length];
            Type currentType = root;
```

```
            for (int i = 0; i < segments.Length; i++)
            {
                string segment = segments[i];
                const BindingFlags flags = BindingFlags.Public |
BindingFlags.NonPublic |
                                           BindingFlags.Instance |
BindingFlags.Static |
                                           BindingFlags.FlattenHierarchy;

                MemberInfo member;
                int arrayIndex = -1;

                int parenIndex = segment.IndexOf('(');
                if (parenIndex >= 0)
                {
                    // Method with signature: "MethodName(Type1,Type2)" or
"MethodName()"
                    string methodName = segment.Substring(0, parenIndex);
                    string paramsPart = segment.Substring(parenIndex + 1,
segment.Length - parenIndex - 2);
                    member = FindMethod(currentType, methodName, paramsPart, flags);
                }
                else if (TryParseArraySegment(segment, out string memberName, out
arrayIndex))
                {
                    // Array-indexed segment: "items[2]" → field "items", index 2
                    var key = new TypeStringKey(currentType, memberName);
                    if (!memberCache.TryGetValue(key, out member))
                    {
                        member = (MemberInfo)currentType.GetField(memberName, flags)
                                 ?? (MemberInfo)currentType.GetProperty(memberName,
flags);
                        memberCache[key] = member;
                    }
                }
                else
                {
                    // Plain field or property
                    var key = new TypeStringKey(currentType, segment);
                    if (!memberCache.TryGetValue(key, out member))
                    {
                        member = (MemberInfo)currentType.GetField(segment, flags)
                                 ?? (MemberInfo)currentType.GetProperty(segment,
flags);
                        memberCache[key] = member;
                    }
                }

                if (member == null)
                    return; // resolution failed

                chain[i] = new ChainStep { member = member, arrayIndex = arrayIndex
};

                // Advance the type for the next segment
                currentType = member switch
                {
                    FieldInfo fi   => fi.FieldType,
                    PropertyInfo pi => pi.PropertyType,
                    MethodInfo mi  => mi.ReturnType,
                    _ => null
                };

                if (currentType == null)
```

```
                {
                    // A null type is only fatal if we still have more segments to
traverse.
                    // A void-returning terminal method is valid — don't abort.
                    if (i < segments.Length - 1)
                        return;
                    break;
                }

                // When indexed, step into the list element type
                if (arrayIndex >= 0)
                {
                    currentType = GetListElementType(currentType);
                    if (currentType == null)
                        return; // not a recognisable list type
                }
            }

            resolvedChain = chain;
            resolvedType = currentType;
        }

        private static bool TryParseArraySegment(string segment, out string name, out
int index)
        {
            name = null;
            index = -1;
            int open  = segment.IndexOf('[');
            int close = segment.IndexOf(']');
            if (open < 0 || close < 0 || close <= open) return false;
            name = segment.Substring(0, open);
            return int.TryParse(segment.Substring(open + 1, close - open - 1), out
index);
        }

        private static Type GetListElementType(Type listType)
        {
            if (listType.IsArray)
                return listType.GetElementType();

            if (listType.IsGenericType)
            {
                var def = listType.GetGenericTypeDefinition();
                if (def == typeof(System.Collections.Generic.List<>)
                    || def == typeof(System.Collections.Generic.IList<>)
                    || def == typeof(System.Collections.Generic.IReadOnlyList<>))
                    return listType.GetGenericArguments()[0];
            }

            // Non-generic IList — element type is object
            if (typeof(IList).IsAssignableFrom(listType))
                return typeof(object);

            return null;
        }

        /// <summary>
        /// Splits a path string on '.' while respecting parenthesized method
signatures.
        /// E.g., "stats.Heal(Int32,Boolean).result" → ["stats",
"Heal(Int32,Boolean)", "result"]
        /// </summary>
        public static string[] SplitPath(string path)
        {
```

```
            var segments = new System.Collections.Generic.List<string>();
            int depth = 0;
            int start = 0;

            for (int i = 0; i < path.Length; i++)
            {
                char c = path[i];
                if (c == '(') depth++;
                else if (c == ')') depth--;
                else if (c == '.' && depth == 0)
                {
                    segments.Add(path.Substring(start, i - start));
                    start = i + 1;
                }
            }

            if (start < path.Length)
                segments.Add(path.Substring(start));

            return segments.ToArray();
        }

        private static MethodInfo FindMethod(Type type, string methodName, string
paramsPart, BindingFlags flags)
        {
            string[] expectedParamNames = string.IsNullOrEmpty(paramsPart)
                ? Array.Empty<string>()
                : paramsPart.Split(',');

            // Trim whitespace
            for (int i = 0; i < expectedParamNames.Length; i++)
                expectedParamNames[i] = expectedParamNames[i].Trim();

            foreach (var method in type.GetMethods(flags))
            {
                if (method.Name != methodName)
                    continue;

                var parameters = method.GetParameters();
                if (parameters.Length != expectedParamNames.Length)
                    continue;

                bool match = true;
                for (int i = 0; i < parameters.Length; i++)
                {
                    string paramTypeName =
SerializedType.GetCleanTypeName(parameters[i].ParameterType);
                    if (paramTypeName != expectedParamNames[i])
                    {
                        match = false;
                        break;
                    }
                }

                if (match)
                    return method;
            }

            return null;
        }
    }

    /// <summary>
    /// Result of a <see cref="MemberPath"/> resolution or value operation.
```

```
    /// </summary>
    public struct ResolveResult
    {
        public bool success;
        public object value;
        public string error;

        /// <summary>Creates a successful result carrying <paramref
name="value"/>.</summary>
        public static ResolveResult Success(object value)
            => new ResolveResult { success = true, value = value };

        /// <summary>Creates a failed result carrying <paramref
name="error"/>.</summary>
        public static ResolveResult Fail(string error)
            => new ResolveResult { success = false, error = error };
    }
}
```

DataField.cs:

```
using System;
using UnityEditor;
using UnityEngine;

namespace VisualEditor.Editor
{
    /// <summary>
    /// Base class for all data fields.
    /// Provides the basic structure for data source selection.
    /// </summary>
    [Serializable]
    public abstract class DataField
    {
        public DataSourceMode mode = DataSourceMode.ObjectBound;
        [MemberPathRoot(MemberPathRootAttribute.RootSource.LayoutAsset)]
        public MemberPath path = new();
        public InterfaceReference<IGraphAsset> graph = new();

        public abstract Type FieldType { get; }
        public abstract bool TryGetValue(object rootObject, string rootPath, out
object result);

        public bool TryGetValue(GenerationContext ctx, out object result)
        {
            return TryGetValue(ctx.TargetObject, ctx.CurrentProperty?.propertyPath,
out result);
        }

        public virtual bool TryGetSerializedProperty(GenerationContext ctx, long
manualId, out SerializedProperty prop)
        {
            prop = null;
            string rootPath = ctx.CurrentProperty?.propertyPath;

            switch (mode)
            {
                case DataSourceMode.ObjectBound:
                    if (!path.HasPath || path.IsBroken) return false;
```

```
                    string bindingPath =
EditorBindingUtility.CombineForUIToolkit(rootPath, path.Path);
                    prop = ctx.SerializedObject.FindProperty(bindingPath);
                    if (prop != null) return true;

                    // Virtualization fallback
                    if (ctx.Virtualization.IsActive &&
ctx.Virtualization.CanProvideMemberBinding(ctx.TargetObject, path))
                    {
                        prop = ctx.Virtualization.GetMemberProxy(ctx.TargetObject,
path);
                        return prop != null;
                    }
                    break;

                case DataSourceMode.Manual:
                    if (ctx.Virtualization.IsActive)
                    {
                        prop = ctx.Virtualization.GetManualProxy(ctx.TargetObject,
FieldType ?? typeof(object), manualId);
                        return prop != null;
                    }
                    break;
            }

            return false;
        }

        public bool TryGetValueFromGraph(object rootObject, string rootPath, out
object result)
        {
            return TryGetValueFromGraph(rootObject, rootPath, true, out result);
        }

        public bool TryGetValueFromGraph(GenerationContext ctx, out object result)
        {
            return TryGetValueFromGraph(ctx.TargetObject,
ctx.CurrentProperty?.propertyPath, out result);
        }

        public bool TryGetValueFromGraph<T>(object rootObject, string rootPath, out T
result)
        {
            return TryGetValueFromGraph(rootObject, rootPath, false, out result);
        }

        public bool TryGetValueFromGraph<T>(GenerationContext ctx, out T result)
        {
            return TryGetValueFromGraph<T>(ctx.TargetObject,
ctx.CurrentProperty?.propertyPath, out result);
        }

        private bool TryGetValueFromGraph<T>(object rootObject, string rootPath, bool
allowAnyType, out T result)
        {
            result = default;
            var graphModule = ModuleRegistry.Graph;
            if (!graphModule.IsActive)
            {
                Debug.LogWarning($"[VisualEditor] Graph module is not active. Cannot
evaluate graph for {rootPath}.");
                return false;
            }
            var graphAsset = graph.Value;
```

```
            if (graphAsset == null)
            {
                Debug.LogWarning($"[VisualEditor] No graph assigned for
{rootPath}.");
                return false;
            }
            if (!graphModule.CanEvaluate(graphAsset))
            {
                Debug.LogWarning($"[VisualEditor] Cannot evaluate graph for
{rootPath}.");
                return false;
            }
            if (!allowAnyType && graphAsset.ResultType != typeof(T))
            {
                Debug.LogWarning($"[VisualEditor] Graph result type mismatch:
expected {typeof(T)}, got {graphAsset.ResultType}");
                return false;
            }

            object targetObject = rootObject;
            if (!string.IsNullOrEmpty(rootPath))
            {
                MemberPath memberPath = MemberPath.FromRootType(rootObject.GetType(),
rootPath);
                if (!memberPath.IsValid)
                {
                    Debug.LogWarning($"[VisualEditor] Invalid member path:
{rootPath}");
                    return false;
                }
                var targetObjectResult = memberPath.TryGetValue(rootObject);
                if (!targetObjectResult.success)
                {
                    Debug.LogWarning($"[VisualEditor] Failed to get target object:
{targetObjectResult.error}");
                    return false;
                }
                targetObject = targetObjectResult.value;
            }

            var graphResult = graphModule.Evaluate<T>(graphAsset, targetObject);
            if (!graphResult.success)
            {
                Debug.LogWarning($"[VisualEditor] Graph evaluation failed for
{rootPath}: {graphResult.error}");
                return false;
            }

            result = graphResult.value;
            return true;
        }
    }

    /// <summary>
    /// Untyped version of DataField that allows selecting any member.
    /// Used when the target type is dynamic or unknown at compile time.
    /// </summary>
    [Serializable]
    public class DataFieldAny : DataField
    {
        // Inherits everything from DataField, acts as a concrete implementation for
"any" type.

        public override Type FieldType => mode switch
```

```
        {
            DataSourceMode.ObjectBound => path.ResolvedType,
            DataSourceMode.Graph => graph.Value?.ResultType,
            _ => null // Not supported yet
        };

        public override bool TryGetValue(object rootObject, string rootPath, out
object result)
        {
            switch (mode)
            {
                case DataSourceMode.ObjectBound:
                    var memberPath = MemberPath.FromRootType(rootObject.GetType(),
rootPath, path.Path);
                    if (!memberPath.IsValid)
                    {
                        Debug.LogWarning($"[VisualEditor] Invalid member path for
object-bound data field: {rootPath} + {path.Path}");
                        result = default;
                        return false;
                    }

                    var objectBoundResult = memberPath.TryGetValue(rootObject);
                    if (!objectBoundResult.success)
                    {
                        Debug.LogWarning($"[VisualEditor] Failed to get value from
object for {rootPath}: {objectBoundResult.error}");
                        result = default;
                        return false;
                    }

                    result = objectBoundResult.value;
                    return true;
                case DataSourceMode.Graph:
                    return TryGetValueFromGraph(rootObject, rootPath, out result);
                //case DataSourceMode.Manual:
                //    result = value;
                //    return true;
                default:
                    Debug.LogError($"[VisualEditor] Unsupported data source mode:
{mode}");
                    result = default;
                    return false;
            }
        }
    }

    /// <summary>
    /// Generic version of DataField that enforces type-safety.
    /// </summary>
    /// <typeparam name="T">The required type of the data.</typeparam>
    [Serializable]
    public class DataField<T> : DataField
    {
        /// <summary>
        /// Manual value used when mode is DataSourceMode.Manual.
        /// </summary>
        public T value;

        public override Type FieldType => typeof(T);

        public DataField() { }

        public DataField(T initialValue)
```

```
        {
            value = initialValue;
            mode = DataSourceMode.Manual;
        }

        public override bool TryGetValue(object rootObject, string rootPath, out
object result)
        {
            return TryGetValue(rootObject, rootPath, out result);
        }

        public bool TryGetValue(object rootObject, string rootPath, out T result)
        {
            switch (mode)
            {
                case DataSourceMode.ObjectBound:
                    var memberPath = MemberPath.FromRootType(rootObject.GetType(),
rootPath, path.Path);
                    if (!memberPath.IsValid)
                    {
                        Debug.LogWarning($"[VisualEditor] Invalid member path for
object-bound data field: {rootPath} + {path.Path}");
                        result = default;
                        return false;
                    }

                    var objectBoundResult = memberPath.TryGetValue(rootObject);
                    if (!objectBoundResult.success)
                    {
                        Debug.LogWarning($"[VisualEditor] Failed to get value from
object for {rootPath}: {objectBoundResult.error}");
                        result = default;
                        return false;
                    }

                    if (objectBoundResult.value is not T typedValue)
                    {
                        Debug.LogWarning($"[VisualEditor] Type mismatch for object-
bound value at {rootPath}: expected {typeof(T)}, got
{objectBoundResult.value?.GetType()}");
                        result = default;
                        return false;
                    }

                    result = typedValue;
                    return true;
                case DataSourceMode.Graph:
                    return TryGetValueFromGraph(rootObject, rootPath, out result);
                case DataSourceMode.Manual:
                    result = value;
                    return true;
                default:
                    Debug.LogError($"[VisualEditor] Unsupported data source mode:
{mode}");
                    result = default;
                    return false;
            }
        }

        public bool TryGetValue(GenerationContext ctx, out T result)
        {
            return TryGetValue(ctx.TargetObject, ctx.CurrentProperty?.propertyPath,
out result);
        }
```

```
    }
}
```

LayoutElement.cs:

```
using System;

namespace VisualEditor.Editor
{
    /// <summary>
    /// Shared base for LayoutNode and LayoutModifier.
    /// Provides the disassembler hook used by [DisassemblesFrom]-annotated
subclasses.
    /// </summary>
    [Serializable]
    public abstract class LayoutElement
    {
        /// <summary>
        /// Called by the disassembler to populate this element from a matched Unity
attribute.
        /// Override in concrete subclasses that declare [DisassemblesFrom].
        /// </summary>
        public virtual void PopulateFromAttribute(Attribute attribute) { }
    }
}
```

LayoutAsset.cs:

```
using System.Collections.Generic;
using UnityEngine;

namespace VisualEditor.Editor
{
    [CreateAssetMenu(fileName = "InspectorLayout", menuName = "Visual
Editor/Inspector Layout")]
    public class LayoutAsset : ScriptableObject
    {
        public SerializedType targetType = new();
        public List<PolymorphicElement<LayoutNode>> nodes = new();

        public static event System.Action<LayoutAsset> OnLayoutAssetValidated;

        private void OnValidate()
        {
            OnLayoutAssetValidated?.Invoke(this);
        }
    }
}
```

InspectorLayoutRegistry.cs:

```
using System;
using System.Collections.Generic;
using UnityEditor;
```

```
using UnityEngine;

namespace VisualEditor.Editor
{
    /// <summary>
    /// Central registry mapping target types to their layout assets.
    /// Supports both inspector editor entries (MonoBehaviour/ScriptableObject)
    /// and property drawer entries (any serializable type).
    /// Can live anywhere in the project — located via
<c>AssetDatabase.FindAssets</c>.
    /// </summary>
    [CreateAssetMenu(
        fileName = "InspectorLayoutRegistry",
        menuName = "Visual Editor/Inspector Layout Registry")]
    public class InspectorLayoutRegistry : ScriptableObject
    {
        // ── Inspector editor entries (MonoBehaviour / ScriptableObject) ──

        [Serializable]
        public class InspectorEntry
        {
            public SerializedType targetType = new();
            public LayoutAsset layoutAsset;
            public RenderStrategy strategy = RenderStrategy.Layout;
            [SerializedTypeFilter(typeof(UnityEditor.Editor))]
            public SerializedType customEditorType = new();
            public bool includeInherited;
        }

        // ── Property drawer entries (any serializable type) ──

        public enum PropertyDrawerStrategy
        {
            /// <summary>Use the data-driven layout from the registered
asset.</summary>
            Layout,
            /// <summary>Route to a specific existing PropertyDrawer type.</summary>
            CustomDrawer,
            /// <summary>Use standard UI Toolkit PropertyField (respects other
PropertyDrawers).</summary>
            Standard,
            /// <summary>Bypass all custom drawers — use Unity's internal raw
rendering.</summary>
            Native
        }

        [Serializable]
        public class PropertyDrawerEntry
        {
            public SerializedType targetType = new();
            public LayoutAsset layoutAsset;
            public PropertyDrawerStrategy strategy = PropertyDrawerStrategy.Layout;
            [SerializedTypeFilter(typeof(PropertyDrawer))]
            public SerializedType customDrawerType = new();
            public bool includeInherited;
        }

        public enum RenderStrategy
        {
            /// <summary>Use the data-driven layout from the registered
asset.</summary>
            Layout,
            /// <summary>Use the original custom editor for the target
type.</summary>
```

```
            CustomEditor,
            /// <summary>Fall back to Unity's default inspector rendering.</summary>
            UnityDefault
        }

        [SerializeField] private List<InspectorEntry> inspectorEntries = new();
        [SerializeField] private List<PropertyDrawerEntry> propertyDrawerEntries =
new();

        // ── Project-wide settings ──

        [SerializeField] private bool enableDynamicReload;
        [SerializeField] private uint dynamicReloadInterval = 50;
        [SerializeField] private uint scheduledOperationsInterval = 500;
        [SerializeField] private DiagnosticsConfig diagnosticsConfig = new();

        // ── Cached lookup ──

        private Dictionary<Type, InspectorEntry> inspectorCache;
        private Dictionary<Type, PropertyDrawerEntry> propertyDrawerCache;

        public bool DynamicReloadEnabled => enableDynamicReload;

        public uint DynamicReloadInterval => dynamicReloadInterval;

        public uint ScheduledOperationsInterval => scheduledOperationsInterval;

        public DiagnosticsConfig DiagnosticsConfig => diagnosticsConfig;

        public IReadOnlyList<InspectorEntry> InspectorEntries => inspectorEntries;

        public IReadOnlyList<PropertyDrawerEntry> PropertyDrawerEntries =>
propertyDrawerEntries;

        public InspectorEntry FindInspectorEntry(Type targetType)
        {
            if (targetType == null)
                return null;

            EnsureInspectorCache();

            if (inspectorCache.TryGetValue(targetType, out var direct))
                return direct;

            // Walk inheritance chain for includeInherited entries
            Type current = targetType.BaseType;
            while (current != null && current != typeof(object))
            {
                if (inspectorCache.TryGetValue(current, out var entry) &&
entry.includeInherited)
                    return entry;
                current = current.BaseType;
            }

            return null;
        }

        /// <summary>
        /// Finds the property drawer entry for the given type, considering
<c>includeInherited</c>.
        /// Returns null if no entry matches.
        /// </summary>
        public PropertyDrawerEntry FindPropertyDrawerEntry(Type targetType)
        {
```

```
            if (targetType == null)
                return null;

            EnsurePropertyDrawerCache();

            if (propertyDrawerCache.TryGetValue(targetType, out var direct))
                return direct;

            // Walk inheritance chain for includeInherited entries
            Type current = targetType.BaseType;
            while (current != null && current != typeof(object))
            {
                if (propertyDrawerCache.TryGetValue(current, out var entry) &&
entry.includeInherited)
                    return entry;
                current = current.BaseType;
            }

            return null;
        }

        /// <summary>
        /// Invalidates the internal lookup caches. Call when entries are modified.
        /// </summary>
        public void InvalidateCache()
        {
            inspectorCache = null;
            propertyDrawerCache = null;
        }

        public static event System.Action OnRegistryValidated;

        private void OnValidate()
        {
            InvalidateCache();
            OnRegistryValidated?.Invoke();
        }

        private void EnsureInspectorCache()
        {
            if (inspectorCache != null)
                return;

            inspectorCache = new Dictionary<Type, InspectorEntry>();
            foreach (var entry in inspectorEntries)
            {
                Type type = entry.targetType?.Type;
                if (type != null)
                    inspectorCache.TryAdd(type, entry);
            }
        }

        private void EnsurePropertyDrawerCache()
        {
            if (propertyDrawerCache != null)
                return;

            propertyDrawerCache = new Dictionary<Type, PropertyDrawerEntry>();
            foreach (var entry in propertyDrawerEntries)
            {
                Type type = entry.targetType?.Type;
                if (type != null)
                    propertyDrawerCache.TryAdd(type, entry);
            }
```

```
        }

        // ── Singleton access ──

        private static InspectorLayoutRegistry cachedInstance;

        public static InspectorLayoutRegistry Instance
        {
            get
            {
                if (cachedInstance != null)
                    return cachedInstance;

                string[] guids =
AssetDatabase.FindAssets("t:InspectorLayoutRegistry");
                if (guids.Length == 0)
                    return null;

                string path = AssetDatabase.GUIDToAssetPath(guids[0]);
                cachedInstance =
AssetDatabase.LoadAssetAtPath<InspectorLayoutRegistry>(path);

                if (guids.Length > 1)
                {
                    Debug.LogWarning(
                        $"[VisualEditor] Multiple InspectorLayoutRegistry assets
found. Using: {path}. " +
                        "Only one registry is supported.");
                }

                return cachedInstance;
            }
        }

        /// <summary>
        /// Clears the cached instance. Call after creating or deleting a registry
asset.
        /// </summary>
        public static void ClearInstanceCache()
        {
            cachedInstance = null;
        }
    }
}
```

LayoutDisassembler.cs:

```
using System;
using System.Collections.Generic;
using System.Reflection;
using UnityEditor;
using UnityEngine;

namespace VisualEditor.Editor
{
    public static class LayoutDisassembler
    {
        // Mapping: Unity attribute type → list of (createdType, isNode)
        private static Dictionary<Type, List<DisassemblyMapping>> mappings;
```

```
        private struct DisassemblyMapping
        {
            public Type createdType;
            public bool isNode; // true = LayoutNode subclass, false = LayoutModifier
subclass
        }

        private static void EnsureMappings()
        {
            if (mappings != null) return;
            mappings = new();

            // Scan all LayoutElement subclasses (Nodes and Modifiers)
            foreach (var type in TypeCache.GetTypesDerivedFrom<LayoutElement>())
            {
                if (type.IsAbstract) continue;

                bool isNode = typeof(LayoutNode).IsAssignableFrom(type);

                foreach (var attr in
type.GetCustomAttributes<DisassemblesFromAttribute>())
                {
                    if (!mappings.TryGetValue(attr.UnityAttributeType, out var list))
                        mappings[attr.UnityAttributeType] = list = new();

                    list.Add(new DisassemblyMapping { createdType = type, isNode =
isNode });
                }
            }
        }

        public static void Disassemble(LayoutAsset asset)
        {
            if (asset == null || asset.targetType == null || asset.targetType.Type ==
null)
                return;

            EnsureMappings();

            Type targetType = asset.targetType.Type;
            asset.nodes.Clear();

            if (typeof(UnityEngine.Object).IsAssignableFrom(targetType))
            {
                // Prepend m_Script only for UnityEngine.Object types
                var scriptNode = new ScriptFieldNode();
                scriptNode.modifiers.Add(new PolymorphicElement<LayoutModifier>
                {
                    Value = new ReadOnlyModifier { condition = new
DataField<bool>(true) }
                });
                asset.nodes.Add(new PolymorphicElement<LayoutNode> { Value =
scriptNode });

                asset.nodes.AddRange(GetFieldNodes(targetType));
            }
            else
            {
                var fieldNodes = GetFieldNodes(targetType);
                if (fieldNodes.Count > 0)
                {
                    // Complex type: wrap in foldout
                    var foldout = new FoldoutNode();
                    foldout.modifiers.Add(new PolymorphicElement<LayoutModifier>
```

```
                    {
                        Value = new LabelModifier { mode =
LabelMode.PreferredSourceLabel }
                    });
                    foldout.children.AddRange(fieldNodes);
                    asset.nodes.Add(new PolymorphicElement<LayoutNode> { Value =
foldout });
                }
                else
                {
                    // Primitive/Simple type: single DefaultEditorNode
                    asset.nodes.Add(new PolymorphicElement<LayoutNode> { Value = new
DefaultEditorNode() });
                }
            }
        }

        private static List<PolymorphicElement<LayoutNode>> GetFieldNodes(Type
targetType)
        {
            List<PolymorphicElement<LayoutNode>> nodes = new();
            var fields = targetType.GetFields(BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);

            foreach (var field in fields)
            {
                if (!IsSerialized(field))
                    continue;

                List<PolymorphicElement<LayoutNode>> preFieldNodes = new();
                List<PolymorphicElement<LayoutModifier>> fieldModifiers = new();

                var customAttrs = field.GetCustomAttributes(true);
                foreach (var attrObj in customAttrs)
                {
                    if (attrObj is not Attribute unityAttr) continue;
                    var attrType = unityAttr.GetType();

                    if (mappings.TryGetValue(attrType, out var mapList))
                    {
                        foreach (var map in mapList)
                        {
                            if (map.isNode)
                            {
                                var node =
(LayoutNode)Activator.CreateInstance(map.createdType);
                                node.PopulateFromAttribute(unityAttr);
                                preFieldNodes.Add(new PolymorphicElement<LayoutNode>
{ Value = node });
                            }
                            else
                            {
                                if (!ModifierConstraintValidator.IsCompatible(
                                        map.createdType,
                                        NodeCapabilities.HasValue,
                                        field.FieldType))
                                    continue;

                                var modifier =
(LayoutModifier)Activator.CreateInstance(map.createdType);
                                modifier.PopulateFromAttribute(unityAttr);
                                fieldModifiers.Add(new
PolymorphicElement<LayoutModifier> { Value = modifier });
                            }
```

```
                        }
                    }
                }

                nodes.AddRange(preFieldNodes);

                var propNode = new PropertyFieldNode();
                propNode.dataField.path = new MemberPath(new
SerializedType(targetType), field.Name);
                propNode.modifiers.AddRange(fieldModifiers);
                nodes.Add(new PolymorphicElement<LayoutNode> { Value = propNode });
            }
            return nodes;
        }

        private static bool IsSerialized(FieldInfo field)
        {
            if (field.IsNotSerialized || field.IsDefined(typeof(HideInInspector),
true))
                return false;

            if (field.IsPublic)
                return true;

            return field.IsDefined(typeof(SerializeField), true) ||
field.IsDefined(typeof(SerializeReference), true);
        }
    }
}
```

ElementFactory.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using UnityEditor;
using UnityEngine;
using UnityEngine.UIElements;

namespace VisualEditor.Editor
{
    public static class ElementFactory
    {
        public const int StructuralModifierOrder = 0;
        public const int DecoratorModifierOrder  = 100;

        private static Dictionary<Type, INodeGenerator>      nodeGenerators;
        private static Dictionary<Type, IModifierApplicator> modifierApplicators;

        // ── Initialization ──

        private static void EnsureInitialized()
        {
            if (nodeGenerators != null) return;

            nodeGenerators = ScanHandlers<CustomNodeGeneratorAttribute,
INodeGenerator>(
                attr => attr.NodeType);
```

```
            modifierApplicators = ScanHandlers<CustomModifierApplicatorAttribute,
IModifierApplicator>(
                attr => attr.ModifierType);
        }

        /// <summary>
        /// Generic assembly scanner. Finds all non-abstract classes tagged with
TAttr
        /// that implement THandler, instantiates them, and maps keySelector(attr) →
instance.
        /// </summary>
        private static Dictionary<Type, THandler> ScanHandlers<TAttr, THandler>(
            Func<TAttr, Type> keySelector)
            where TAttr : Attribute
        {
            var result = new Dictionary<Type, THandler>();

            foreach (var type in TypeCache.GetTypesWithAttribute<TAttr>())
            {
                if (type.IsAbstract || type.IsInterface) continue;
                if (!typeof(THandler).IsAssignableFrom(type)) continue;

                var attr = type.GetCustomAttribute<TAttr>();
                var key  = keySelector(attr);
                if (key == null) continue;

                try
                {
                    result[key] = (THandler)Activator.CreateInstance(type);
                }
                catch (Exception e)
                {
                    Debug.LogError(
                        $"[VisualEditor] Failed to instantiate {type.Name}:
{e.Message}");
                }
            }

            return result;
        }

        // ── Public API ──

        /// <summary>
        /// Creates a <see cref="VisualElement"/> for one <see cref="LayoutNode"/>,
then applies
        /// all modifiers in execution order (structural first, then decorator).
        /// The pipeline owns all error rendering and diagnostic recording.
        /// </summary>
        public static VisualElement CreateElement(
            LayoutNode node, GenerationContext context, ref int nodeIndex)
        {
            EnsureInitialized();

            int currentIndex = nodeIndex;
            nodeIndex++;

            var config = context.InspectorLayoutRegistry != null ?
context.InspectorLayoutRegistry.DiagnosticsConfig : new DiagnosticsConfig();

            // Null node and missing generator are pre-generator failures — route
through
            // the same pipeline (Record + config-gated render) as any other issue.
            if (node == null)
```

```
                return RecordAndRender(
                    GenerationResult.Warn("Null node (unassigned polymorphic
type)."),
                    context, config, currentIndex,
                    BuildNodeContext(context, currentIndex, "?"));

            var nodeType = node.GetType();

            if (!nodeGenerators.TryGetValue(nodeType, out var generator))
                return RecordAndRender(
                    GenerationResult.Warn($"No [CustomNodeGenerator] for
'{nodeType.Name}'."),
                    context, config, currentIndex,
                    BuildNodeContext(context, currentIndex, nodeType.Name));

            GenerationResult result;
            try
            {
                result = generator.CreateElement(node, context, ref nodeIndex);
            }
            catch (Exception e)
            {
                result = GenerationResult.Error($"Generator '{nodeType.Name}' threw:
{e.Message}");
            }

            if (!result.IsOk)
                return RecordAndRender(
                    result, context, config, currentIndex,
                    BuildNodeContext(context, currentIndex, nodeType.Name));

            return ApplyModifiers(result.Element, node, context, config,
currentIndex);
        }

        private static VisualElement RecordAndRender(
            GenerationResult result, GenerationContext context,
            DiagnosticsConfig config, int nodeIndex, string contextString)
        {
            context.Diagnostics.Record(result.Severity, result.Message,
contextString, nodeIndex);

            return config.ShouldShow(result.Severity,
DiagnosticsConfig.OutputMode.PerElement)
                ? RenderIssue(result)
                : new VisualElement();
        }

        /// <summary>
        /// Applies all modifiers sorted by Order (ascending).
        /// Structural (Order=0) run first; decorator (Order=100) apply to the final
element.
        /// Also checks modifier constraints (moved here from the deleted
LayoutValidator).
        /// </summary>
        private static VisualElement ApplyModifiers(
            VisualElement element, LayoutNode node,
            GenerationContext context, DiagnosticsConfig config, int nodeIndex)
        {
            // Resolve field type once for constraint checking
            var generator  = nodeGenerators.GetValueOrDefault(node.GetType());
            Type fieldType = generator?.GetFieldType(node);

            var perElementIssues = new List<VisualElement>();
```

```
            // Build list of (applicator, modifier) pairs, skip unknown or
incompatible
            var pairs = new List<(IModifierApplicator applicator, LayoutModifier
modifier)>();

            foreach (var modWrapper in node.modifiers)
            {
                var modifier = modWrapper.Value;
                if (modifier == null) continue;

                var modType = modifier.GetType();

                // Constraint check (moved here from the deleted LayoutValidator)
                if (!ModifierConstraintValidator.IsCompatible(modType,
node.Capabilities, fieldType))
                {
                    var constraintMsg = $"'{modType.Name}' is incompatible with
'{node.Capabilities}'.";
                    context.Diagnostics.Record(
                        Severity.Warning, constraintMsg,
                        BuildModifierContext(context, nodeIndex, node.GetType().Name,
modType.Name),
                        nodeIndex);

                    if (config.ShouldShow(Severity.Warning,
DiagnosticsConfig.OutputMode.PerElement))
                        perElementIssues.Add(CreateWarningLabel(constraintMsg));

                    continue; // skip this modifier
                }

                if (modifierApplicators.TryGetValue(modType, out var applicator))
                {
                    pairs.Add((applicator, modifier));
                }
                else
                {
                    var msg = $"No [CustomModifierApplicator] for '{modType.Name}'.";
                    context.Diagnostics.Record(
                        Severity.Warning, msg,
                        BuildModifierContext(context, nodeIndex, node.GetType().Name,
modType.Name),
                        nodeIndex);

                    if (config.ShouldShow(Severity.Warning,
DiagnosticsConfig.OutputMode.PerElement))
                        perElementIssues.Add(CreateWarningLabel(msg));
                }
            }

            // Sort by Order — structural first, decorator after
            pairs.Sort((a, b) => a.applicator.Order.CompareTo(b.applicator.Order));

            foreach (var (applicator, modifier) in pairs)
            {
                GenerationResult result;
                try
                {
                    result = applicator.Apply(element, modifier, node, context,
nodeIndex);
                }
                catch (Exception e)
                {
```

```
                    result = GenerationResult.Warn(
                        $"Applicator '{modifier.GetType().Name}' threw:
{e.Message}");
                }

                if (!result.IsOk)
                {
                    context.Diagnostics.Record(
                        result.Severity, result.Message,
                        BuildModifierContext(context, nodeIndex, node.GetType().Name,
modifier.GetType().Name),
                        nodeIndex);

                    if (config.ShouldShow(result.Severity,
DiagnosticsConfig.OutputMode.PerElement))
                        perElementIssues.Add(RenderIssue(result));

                    // Keep existing element — do not replace it on applicator
failure
                }
                else
                {
                    element = result.Element;
                }
            }

            // No issues — return the element as-is, zero overhead
            if (perElementIssues.Count == 0)
                return element;

            var wrapper = new VisualElement();
            wrapper.AddToClassList("ve-modifier-issue-wrapper");
            foreach (var issue in perElementIssues)
                wrapper.Add(issue);
            wrapper.Add(element);
            return wrapper;
        }

        public static INodeGenerator FindGenerator(Type nodeType)
        {
            EnsureInitialized();
            return nodeGenerators.GetValueOrDefault(nodeType);
        }

        // ── Context builders ──

        private static string BuildNodeContext(GenerationContext ctx, int nodeIndex,
string nodeTypeName)
        {
            var sb = new System.Text.StringBuilder();
            sb.Append(ctx.LayoutChain);
            sb.Append($" → Node {nodeIndex} ({nodeTypeName})");
            if (ctx.CurrentProperty != null)
                sb.Append($" at '{ctx.CurrentProperty.propertyPath}'");
            return sb.ToString();
        }

        private static string BuildModifierContext(
            GenerationContext ctx, int nodeIndex, string nodeTypeName, string
modifierName)
        {
            var sb = new System.Text.StringBuilder();
            sb.Append(ctx.LayoutChain);
            sb.Append($" → Node {nodeIndex} ({nodeTypeName}) → {modifierName}");
```

```
            if (ctx.CurrentProperty != null)
                sb.Append($" at '{ctx.CurrentProperty.propertyPath}'");
            return sb.ToString();
        }

        // ── HelpBox helpers — used internally and by generators for non-diagnostic
UI ──

        public static VisualElement RenderIssue(GenerationResult result) =>
            result.Severity switch
            {
                Severity.Error   => CreateErrorLabel(result.Message),
                Severity.Warning => CreateWarningLabel(result.Message),
                _                => CreateInfoLabel(result.Message)
            };

        /// <summary>Info HelpBox.</summary>
        public static VisualElement CreateInfoLabel(string message)
        {
            var box = new HelpBox(message, HelpBoxMessageType.Info);
            box.AddToClassList("ve-info");
            return box;
        }

        /// <summary>Warning HelpBox.</summary>
        public static VisualElement CreateWarningLabel(string message)
        {
            var box = new HelpBox(message, HelpBoxMessageType.Warning);
            box.AddToClassList("ve-warning");
            return box;
        }

        /// <summary>Error HelpBox.</summary>
        public static VisualElement CreateErrorLabel(string message)
        {
            var box = new HelpBox(message, HelpBoxMessageType.Error);
            box.AddToClassList("ve-error");
            return box;
        }

        public static string GetBindWarningMessage(DataField df) =>
            df.mode switch
            {
                DataSourceMode.Graph       => "Graph mode can't be used to create
serialized property bindings.",
                DataSourceMode.ObjectBound => $"Cannot bind '{df.path.Path}' — not
serialized, no Virtualization.",
                DataSourceMode.Manual      => "Manual mode needs Virtualization.",
                _                          => $"Unknown mode: {df.mode}"
            };
    }
}
```

InspectorOrchestrator.cs:

```
using UnityEditor;
using UnityEditor.UIElements;
using UnityEngine;
using UnityEngine.UIElements;
```

```
namespace VisualEditor.Editor
{
    public static class InspectorOrchestrator
    {
        public static VisualElement GenerateInspector(
            LayoutAsset layoutAsset,
            SerializedObject serializedObject,
            Object targetObject = null,
            UnityEditor.Editor editor = null)
        {
            var root = Generate(layoutAsset, serializedObject, targetObject, null,
editor);
            root.name = "visual-editor-root";
            return root;
        }

        public static VisualElement GenerateProperty(
            LayoutAsset layoutAsset,
            SerializedProperty property,
            UnityEditor.Editor editor = null)
        {
            var root = Generate(layoutAsset, property.serializedObject,
property.serializedObject.targetObject, property, editor);
            root.AddToClassList("ve-property-root");
            root.name = "visual-editor-property-root";
            return root;
        }

        private static VisualElement Generate(
            LayoutAsset layoutAsset,
            SerializedObject serializedObject,
            Object targetObject,
            SerializedProperty initialProperty,
            UnityEditor.Editor editor)
        {
            // ── Null guards ──
            if (layoutAsset == null)
                return ElementFactory.CreateWarningLabel("[VisualEditor] LayoutAsset
is null.");
            if (serializedObject == null)
                return ElementFactory.CreateWarningLabel("[VisualEditor]
SerializedObject is null.");

            if (targetObject == null) targetObject = serializedObject.targetObject;
            if (targetObject == null)
                return ElementFactory.CreateWarningLabel("[VisualEditor] Target
object is null.");

            // ── Create context ──
            var diagnostics = new GenerationDiagnostics();
            var context = new GenerationContext(
                serializedObject, targetObject, InspectorLayoutRegistry.Instance,
layoutAsset,
                ModuleRegistry.Virtualization, ModuleRegistry.Graph,
                diagnostics,
                initialProperty: initialProperty,
                editor: editor);

            // ── Build element tree ──
            var root = new VisualElement();
            root.AddToClassList("ve-root");

            var uss = Resources.Load<StyleSheet>("VisualEditorGeneration");
            if (uss != null) root.styleSheets.Add(uss);
```

```
            int nodeIndex = 0;
            for (int i = 0; i < layoutAsset.nodes.Count; i++)
            {
                var node = layoutAsset.nodes[i].Value;
                root.Add(ElementFactory.CreateElement(node, context, ref nodeIndex));
            }

            // ── Bind ──
            root.Bind(serializedObject);

            // Post-process property fields to fix label/input alignment
            root.RegisterCallback<GeometryChangedEvent>(_ =>
PostProcessPropertyFields(root));

            // ── Diagnostics output (after tree is fully built) ──
            root.userData = diagnostics;

            var config = context.InspectorLayoutRegistry != null
                ? context.InspectorLayoutRegistry.DiagnosticsConfig
                : new DiagnosticsConfig();

            diagnostics.EmitConsoleLogs(config, layoutAsset.name);

            var summaryElement = DiagnosticsSummaryBuilder.Build(diagnostics,
config);
            if (summaryElement != null)
                root.Insert(0, summaryElement);

            return root;
        }

        private static void PostProcessPropertyFields(VisualElement root)
        {
            // Remove the aligned-field class so our own alignment helper can take
over.
            root.Query(className: "unity-base-
field__aligned").ForEach(FieldAlignmentHelper.Attach);
        }
    }
}
```

InterceptorLogic.cs:

```
using System;
using UnityEditor;
using UnityEditor.UIElements;
using UnityEngine;
using UnityEngine.UIElements;
using VisualEditor;

namespace VisualEditor.Editor
{
    /// <summary>
    /// Shared interception logic.
    /// </summary>
    public static class InterceptorLogic
    {
        public static VisualElement CreateInspectorGUI(UnityEditor.Editor
interceptor)
```

```
        {
            var root = CreateInspectorGUI(interceptor, out var usedLayoutAsset);
            if (InspectorLayoutRegistry.Instance != null &&
InspectorLayoutRegistry.Instance.DynamicReloadEnabled)
            {
                // Creates a wrapper root element that can be swapped out when the
layout asset is reloaded, and registers it with the reload manager
                root = InspectorReloadManager.RegisterInterceptor(interceptor, root,
usedLayoutAsset);
            }
            return root;
        }

        private static VisualElement CreateInspectorGUI(UnityEditor.Editor
interceptor, out LayoutAsset usedLayoutAsset)
        {
            usedLayoutAsset = null;

            if (interceptor.target == null) return
CreateDefaultInspector(interceptor);

            var registry = InspectorLayoutRegistry.Instance;
            if (registry == null) return CreateDefaultInspector(interceptor);

            var type = interceptor.target.GetType();
            var entry = registry.FindInspectorEntry(type);

            if (entry == null) return CreateDefaultInspector(interceptor);

            if (entry.strategy == InspectorLayoutRegistry.RenderStrategy.Layout)
            {
                if (entry.layoutAsset != null)
                {
                    usedLayoutAsset = entry.layoutAsset;
                    return InspectorOrchestrator.GenerateInspector(entry.layoutAsset,
interceptor.serializedObject, interceptor.target, interceptor);
                }
                return ElementFactory.CreateWarningLabel($"[VisualEditor] LayoutAsset
missing for {type.Name}");
            }
            else if (entry.strategy ==
InspectorLayoutRegistry.RenderStrategy.CustomEditor)
            {
                if (entry.customEditorType != null && entry.customEditorType.Type !=
null)
                {
                    Type customEditorType = entry.customEditorType.Type;

                    var customEditor =
UnityEditor.Editor.CreateEditor(interceptor.targets, customEditorType);

                    VisualElement root = customEditor.CreateInspectorGUI() ??
                        new IMGUIContainer(() =>
                        {
                            if (customEditor != null && customEditor.target != null)
                            {
                                customEditor.OnInspectorGUI();
                            }
                        });

                    root.RegisterCallback<DetachFromPanelEvent>(e =>
                    {
                        if (customEditor != null)
                        {
```

```
                            UnityEngine.Object.DestroyImmediate(customEditor);
                        }
                    });

                    return root;
                }

                return ElementFactory.CreateWarningLabel($"[VisualEditor] No valid
CustomEditor selected for {type.Name}");
            }

            return CreateDefaultInspector(interceptor);
        }

        private static VisualElement CreateDefaultInspector(UnityEditor.Editor
interceptor)
        {
            var root = new VisualElement();
            InspectorElement.FillDefaultInspector(root, interceptor.serializedObject,
interceptor);
            return root;
        }
    }
}
```

VisualEditorInterceptor.cs:

```
using UnityEditor;
using UnityEngine;
using UnityEngine.UIElements;

namespace VisualEditor.Editor
{
    [CustomEditor(typeof(MonoBehaviour), true)]
    [CanEditMultipleObjects]
    public class VisualEditorMonoBehaviourInterceptor : UnityEditor.Editor
    {
        public override VisualElement CreateInspectorGUI()
        {
            return InterceptorLogic.CreateInspectorGUI(this);
        }
    }

    [CustomEditor(typeof(ScriptableObject), true)]
    [CanEditMultipleObjects]
    public class VisualEditorScriptableObjectInterceptor : UnityEditor.Editor
    {
        public override VisualElement CreateInspectorGUI()
        {
            return InterceptorLogic.CreateInspectorGUI(this);
        }
    }
}
```

VisualEditorPropertyDrawerInterceptor.cs:

```
using System;
```

```
using System.Reflection;
using UnityEditor;
using UnityEditor.UIElements;
using UnityEngine;
using UnityEngine.UIElements;

namespace VisualEditor.Editor
{
    [CustomPropertyDrawer(typeof(object), true)]
    public class VisualEditorPropertyDrawerInterceptor : PropertyDrawer
    {
        private static readonly FieldInfo internalFieldInfo =
typeof(PropertyDrawer).GetField("m_FieldInfo", BindingFlags.Instance |
BindingFlags.NonPublic);
        private static readonly FieldInfo internalAttribute =
typeof(PropertyDrawer).GetField("m_Attribute", BindingFlags.Instance |
BindingFlags.NonPublic);
        private static readonly FieldInfo internalPreferredLabel =
typeof(PropertyDrawer).GetField("m_PreferredLabel", BindingFlags.Instance |
BindingFlags.NonPublic);

        // ── UI Toolkit path ──────────────────────────────────────────────

        public override VisualElement CreatePropertyGUI(SerializedProperty property)
        {
            Type targetType = GetTargetType(property);
            if (targetType == null)
                return DefaultPropertyFieldHelper.CreateNativePropertyField(property,
preferredLabel);

            var registry = InspectorLayoutRegistry.Instance;
            var entry = registry != null ?
registry.FindPropertyDrawerEntry(targetType) : null;

            if (entry != null)
            {
                switch (entry.strategy)
                {
                    case InspectorLayoutRegistry.PropertyDrawerStrategy.Layout:
                        if (entry.layoutAsset != null)
                            // Runs diagnostics internally
                            return
InspectorOrchestrator.GenerateProperty(entry.layoutAsset, property);

                        return RunIssue(
                            GenerationResult.Error($"No layout asset assigned for
'{entry.targetType.Type?.Name}' in the InspectorLayoutRegistry."),
                            property);

                    case InspectorLayoutRegistry.PropertyDrawerStrategy.CustomDrawer:
                        var drawerType = entry.customDrawerType?.Type;
                        if (drawerType != null && drawerType !=
typeof(VisualEditorPropertyDrawerInterceptor))
                            return CreateCustomDrawer(property, drawerType);

                        return RunIssue(
                            GenerationResult.Error($"No valid CustomPropertyDrawer
selected for '{entry.targetType.Type?.Name}' in the InspectorLayoutRegistry."),
                            property);

                    case InspectorLayoutRegistry.PropertyDrawerStrategy.Native:
                        return
DefaultPropertyFieldHelper.CreateNativePropertyField(property, preferredLabel);
```

```
                    case InspectorLayoutRegistry.PropertyDrawerStrategy.Standard:
                        return
DefaultPropertyFieldHelper.CreateNativePropertyField(property, preferredLabel);
                }
            }

            // Fallback: check for user-defined PropertyDrawers
            var fallbackDrawerType =
CustomEditorResolver.FindCustomDrawerType(targetType);
            if (fallbackDrawerType != null && fallbackDrawerType !=
typeof(VisualEditorPropertyDrawerInterceptor))
                return CreateCustomDrawer(property, fallbackDrawerType);

            return DefaultPropertyFieldHelper.CreateNativePropertyField(property,
preferredLabel);
        }

        // ── IMGUI fallback ──────────────────────────────────────────────────

        public override void OnGUI(Rect position, SerializedProperty property,
GUIContent label)
        {
            EditorGUI.PropertyField(position, property, label, true);
        }

        public override float GetPropertyHeight(SerializedProperty property,
GUIContent label)
        {
            return EditorGUI.GetPropertyHeight(property, label, true);
        }

        // ── Type resolution ─────────────────────────────────────────────────

        private Type GetTargetType(SerializedProperty property)
        {
            // For [SerializeReference] polymorphic fields, resolve the runtime type
            // TODO: consider caching this type info on the property to avoid
repeated reflection and assembly loading
            if (property.propertyType == SerializedPropertyType.ManagedReference)
            {
                string typeName = property.managedReferenceFullTypename;
                if (!string.IsNullOrEmpty(typeName))
                {
                    var parts = typeName.Split(' ');
                    if (parts.Length == 2)
                    {
                        try
                        {
                            var assembly = Assembly.Load(parts[0]);
                            var runtimeType = assembly?.GetType(parts[1]);
                            if (runtimeType != null) return runtimeType;
                        }
                        catch { /* Fall through to declared type */ }
                    }
                }
            }

            // Default: declared field type from Unity-injected fieldInfo
            return fieldInfo?.FieldType;
        }

        // ── Custom drawer instantiation ─────────────────────────────────────
```

```
        private VisualElement CreateCustomDrawer(SerializedProperty prop, Type
drawerType)
        {
            try
            {
                var drawer = (PropertyDrawer)Activator.CreateInstance(drawerType);

                internalFieldInfo?.SetValue(drawer, fieldInfo);
                internalAttribute?.SetValue(drawer, attribute);
                internalPreferredLabel?.SetValue(drawer, preferredLabel);

                var element = drawer.CreatePropertyGUI(prop);
                if (element != null) return element;

                var guiLabel = new GUIContent(prop.displayName);
                return new IMGUIContainer(() =>
                {
                    if (prop.serializedObject.targetObject == null) return;
                    var rect = EditorGUILayout.GetControlRect(
                        true, drawer.GetPropertyHeight(prop, guiLabel));
                    drawer.OnGUI(rect, prop, guiLabel);
                });
            }
            catch (Exception e)
            {
                return RunIssue(
                    GenerationResult.Error($"CustomDrawer '{drawerType?.Name}' threw:
{e.Message}"),
                    prop);
            }
        }

        // ── Diagnostics pipeline for single-issue interceptor failures ──────────

        private static VisualElement RunIssue(GenerationResult result,
SerializedProperty property)
        {
            var config = InspectorLayoutRegistry.Instance != null
                ? InspectorLayoutRegistry.Instance.DiagnosticsConfig
                : new DiagnosticsConfig();

            var diagnostics = new GenerationDiagnostics();

            string context = property != null
                ? $"[VisualEditor/PropertyDrawer] at '{property.propertyPath}'"
                : "[VisualEditor/PropertyDrawer]";

            diagnostics.Record(result.Severity, result.Message, context, nodeIndex:
0);
            diagnostics.EmitConsoleLogs(config, "[VisualEditor/PropertyDrawer]");

            return config.ShouldShow(result.Severity,
DiagnosticsConfig.OutputMode.PerElement)
                ? ElementFactory.RenderIssue(result)
                : new VisualElement();
        }
    }
}
```

InspectorReloadManager.cs:

```
using System.Collections.Generic;
using UnityEditor;
using UnityEditor.UIElements;
using UnityEngine.UIElements;

namespace VisualEditor.Editor
{
    [InitializeOnLoad]
    public static class InspectorReloadManager
    {
        private static readonly Dictionary<UnityEditor.Editor, VisualElement>
activeInterceptors = new();
        private static readonly Dictionary<UnityEditor.Editor, LayoutAsset>
activeLayouts = new();
        private static readonly List<UnityEditor.Editor> editorsToClean = new();

        // Timer state
        private static double nextReloadTime = -1;
        private static bool reloadAll = false;
        private static readonly HashSet<LayoutAsset> pendingLayoutReloads = new();

        static InspectorReloadManager()
        {
            // Listeners
            LayoutAsset.OnLayoutAssetValidated += RequestReload;
            InspectorLayoutRegistry.OnRegistryValidated += RequestReloadAll;

            // Timer update hook
            EditorApplication.update += CheckReload;
        }

        public static void RequestReload(LayoutAsset layoutAsset)
        {
            if (layoutAsset == null) return;
            reloadAll = true;
            StartTimer();
        }

        public static void RequestReloadAll()
        {
            reloadAll = true;
            StartTimer();
        }

        private static void StartTimer()
        {
            // If registry is not ready, we aren't ready to do any reloads
            if (InspectorLayoutRegistry.Instance == null) return;
            if (nextReloadTime < 0)
            {
                nextReloadTime = EditorApplication.timeSinceStartup +
InspectorLayoutRegistry.Instance.DynamicReloadInterval / 1000d;
            }
        }

        private static void CheckReload()
        {
            if (nextReloadTime >= 0 && EditorApplication.timeSinceStartup >=
nextReloadTime)
            {
                nextReloadTime = -1; // Reset timer
                PerformReload();
            }
        }
```

```
        public static VisualElement RegisterInterceptor(UnityEditor.Editor editor,
VisualElement root, LayoutAsset layout)
        {
            CleanDisposedEditors();

            if (activeInterceptors.ContainsKey(editor))
            {
                // Update layout tracking for existing interceptor
                if (layout == null) activeLayouts.Remove(editor);
                else activeLayouts[editor] = layout;

                var existingWrapper = activeInterceptors[editor];
                existingWrapper.Unbind();
                existingWrapper.Clear();
                existingWrapper.Add(root);
                if (editor.serializedObject != null)
                {
                    existingWrapper.Bind(editor.serializedObject);
                }
                return existingWrapper;
            }

            VisualElement reloadWrapper = new();
            reloadWrapper.AddToClassList("ve-inspector-reload-wrapper");
            reloadWrapper.Add(root);

            activeInterceptors[editor] = reloadWrapper;
            activeLayouts[editor] = layout;

            reloadWrapper.RegisterCallback<DetachFromPanelEvent>(e =>
            {
                activeInterceptors.Remove(editor);
                activeLayouts.Remove(editor);
            });

            return reloadWrapper;
        }

        private static void CleanDisposedEditors()
        {
            editorsToClean.Clear();
            foreach (var key in activeInterceptors.Keys)
            {
                if (key == null) editorsToClean.Add(key);
            }
            foreach (var key in editorsToClean)
            {
                activeInterceptors.Remove(key);
                activeLayouts.Remove(key);
            }
        }

        private static void PerformReload()
        {
            CleanDisposedEditors();

            foreach (var kvp in activeInterceptors)
            {
                var editor = kvp.Key;

                bool shouldReload = reloadAll;
                if (!shouldReload && activeLayouts.TryGetValue(editor, out var
layout))
```

```
                {
                    if (layout != null && pendingLayoutReloads.Contains(layout))
                    {
                        shouldReload = true;
                    }
                }

                if (shouldReload)
                {
                    // Update layout asset tracking if strategy changed in registry
                    var registry = InspectorLayoutRegistry.Instance;
                    var type = editor.target != null ? editor.target.GetType() :
null;
                    if (registry != null && type != null)
                    {
                        var entry = registry.FindInspectorEntry(type);
                        if (entry != null && entry.strategy ==
InspectorLayoutRegistry.RenderStrategy.Layout && entry.layoutAsset != null)
                        {
                            activeLayouts[editor] = entry.layoutAsset;
                        }
                        else
                        {
                            activeLayouts.Remove(editor);
                        }
                    }

                    // As the wrapper is already registered it would automatically
updated inside the RegisterInterceptor method on InterceptorLogic.CreateInspectorGUI
call
                    // We don't external bind here, as the wrapper already was binded
                    _ = InterceptorLogic.CreateInspectorGUI(editor);
                }
            }

            reloadAll = false;
            pendingLayoutReloads.Clear();
        }
    }
}
```

ДОДАТОК Г

(обов’язковий)

ТЕСТОВІ КЛАСИ ТА РЕЗУЛЬТАТИ ТЕСТІВ

ValidTestComponent.cs:

```
[System.Serializable]
public struct CharacterStats
{
    public int health;
    public float speed;
    public bool isAlive;
}

public class ValidTestComponent : MonoBehaviour
{
    [Header("Basic Settings")]
    public string characterName = "Hero";

    [Space(10)]
    [Range(0f, 100f)]
    public float stamina = 100f;

    [Min(0)]
    public int goldCoins = 0;

    [TextArea(2, 5)]
    public string description = "A brave hero.";

    public CharacterStats stats;
}
```

CyclicTestComponent.cs:

```
[System.Serializable]
public class CyclicNode
{
    public string nodeName;
    [SerializeReference] public CyclicNode nextNode;
}

public class CyclicTestComponent : MonoBehaviour
{
    public CyclicNode rootNode;
}
```

Скрипт FallbackTestComponent.cs:

```
public class FallbackTestComponent : MonoBehaviour
{
    public Bounds objectBounds;
    public Gradient colorGradient;
}
```

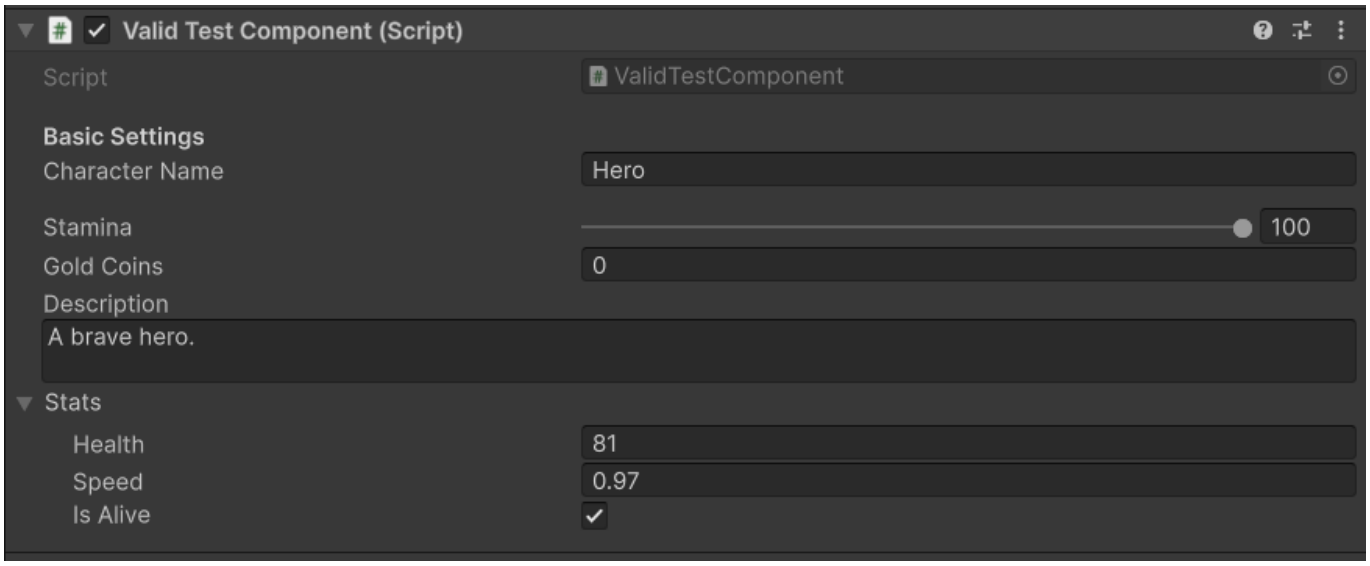


Рисунок Г.1 – Вигляд інспектору ValidTestComponent до розбирання

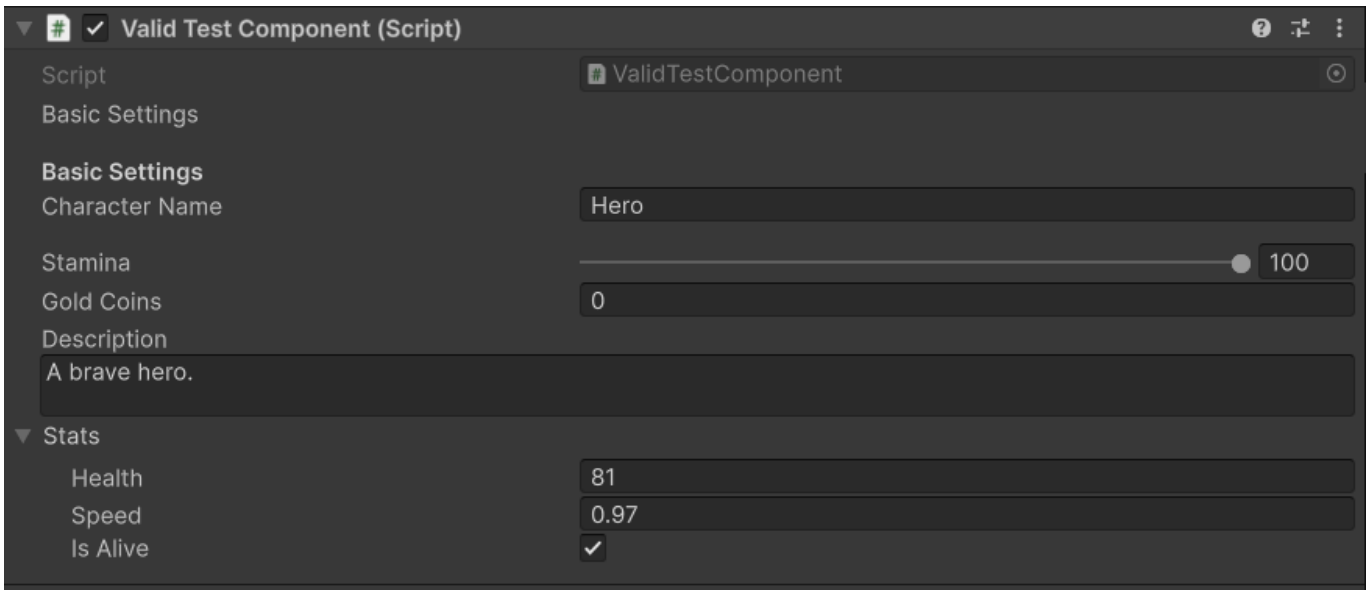


Рисунок Г.2 – Вигляд інспектору ValidTestComponent після розбирання, візуальне дублювання заголовку

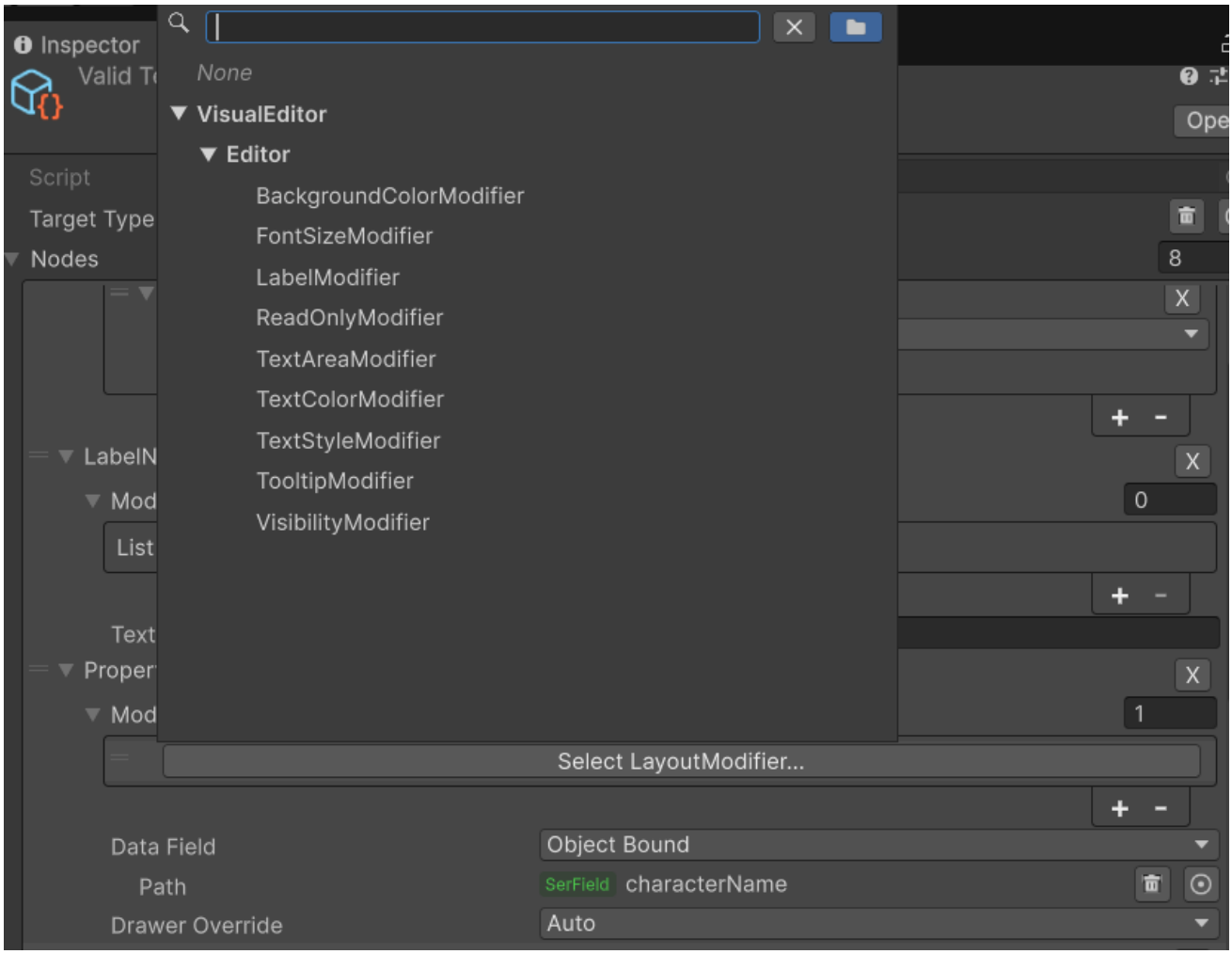


Рисунок Г.3 – Система фільтрує та не пропонує несумісні модифікатори (в цьому випадку Min, Max, Slider відсутні для рядкового типу)

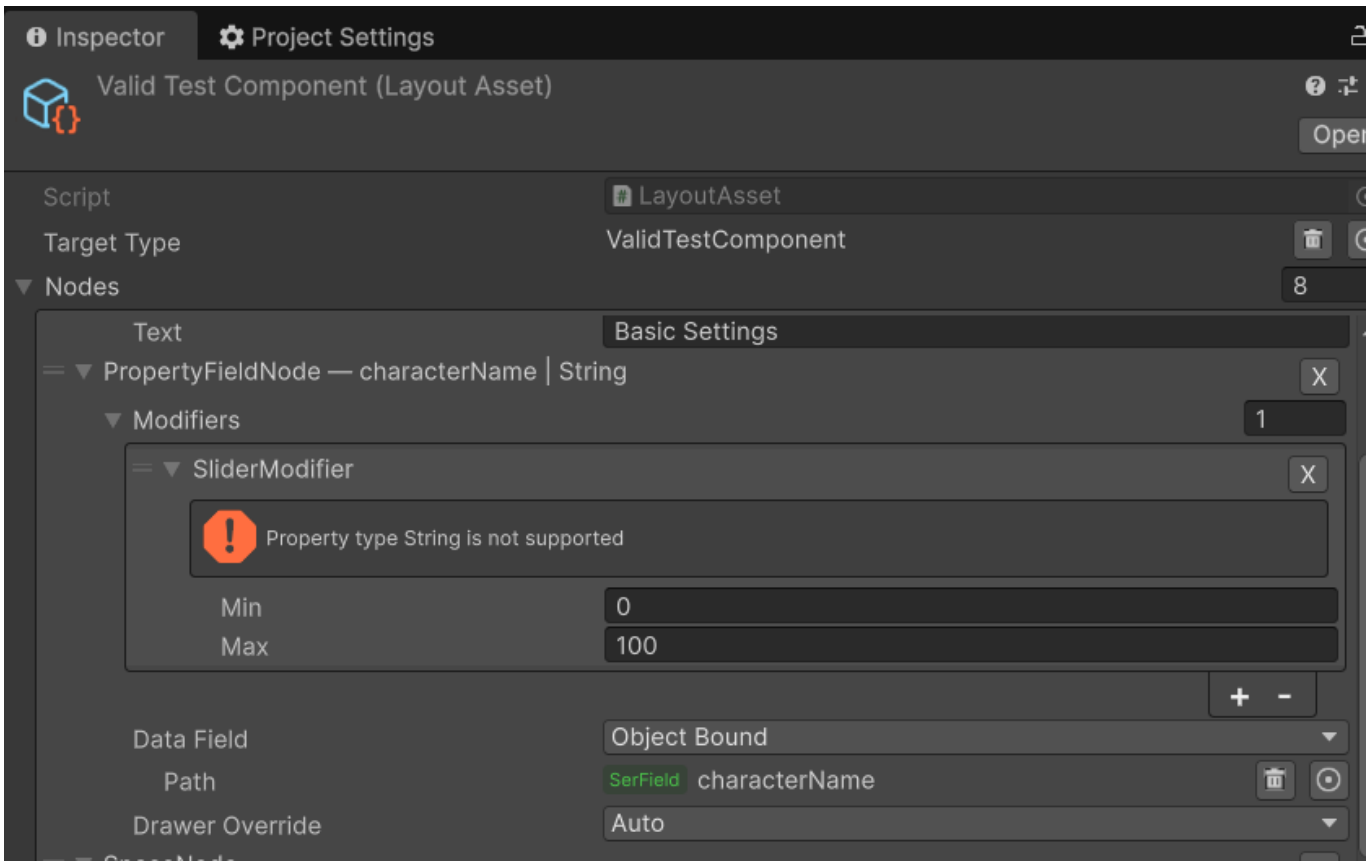


Рисунок Г.4 – Вигляд повідомлення про конфлікт модифікатору з вузлом (наприклад, при динамічній зміні цільової властивості)

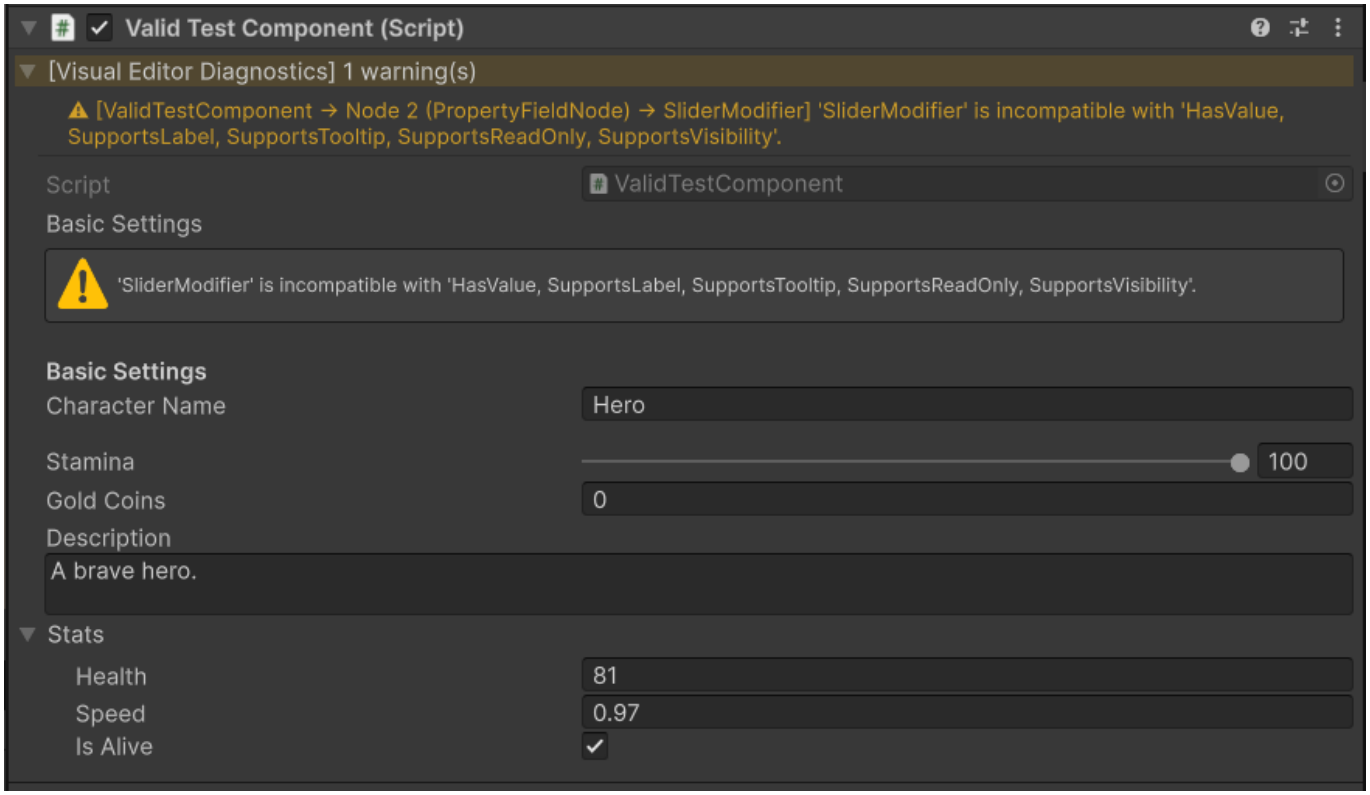


Рисунок Г.5 – Вигляд повідомлень системи діагностики при несумісних типах

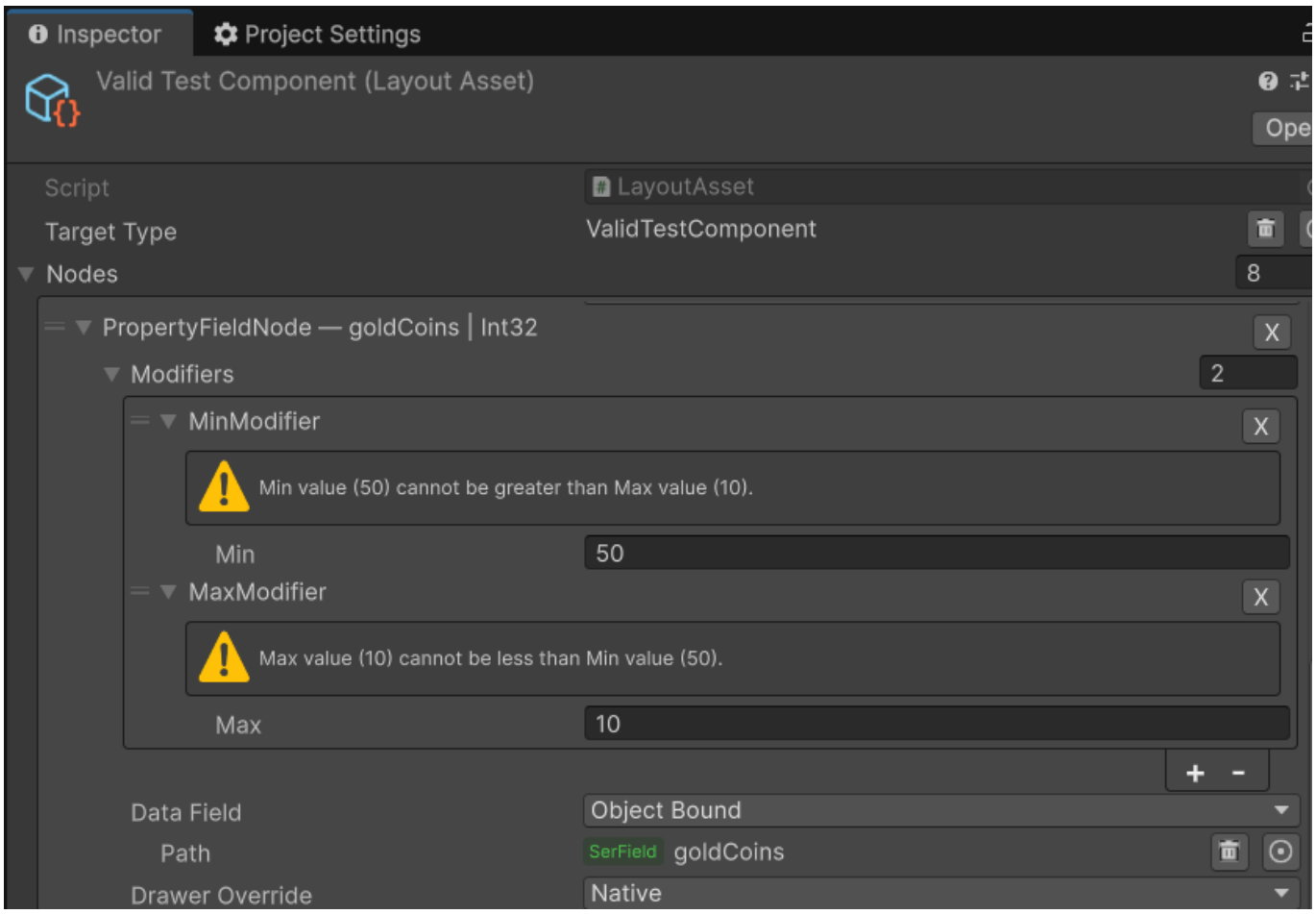


Рисунок Г.6 – Вигляд повідомлення про конфлікт суміжних модифікаторів

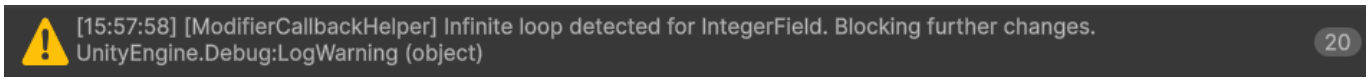


Рисунок Г.7 – Вигляд повідомлення в консолі в результаті фіксації та зупинення нескінченного циклу при зміні значення властивості

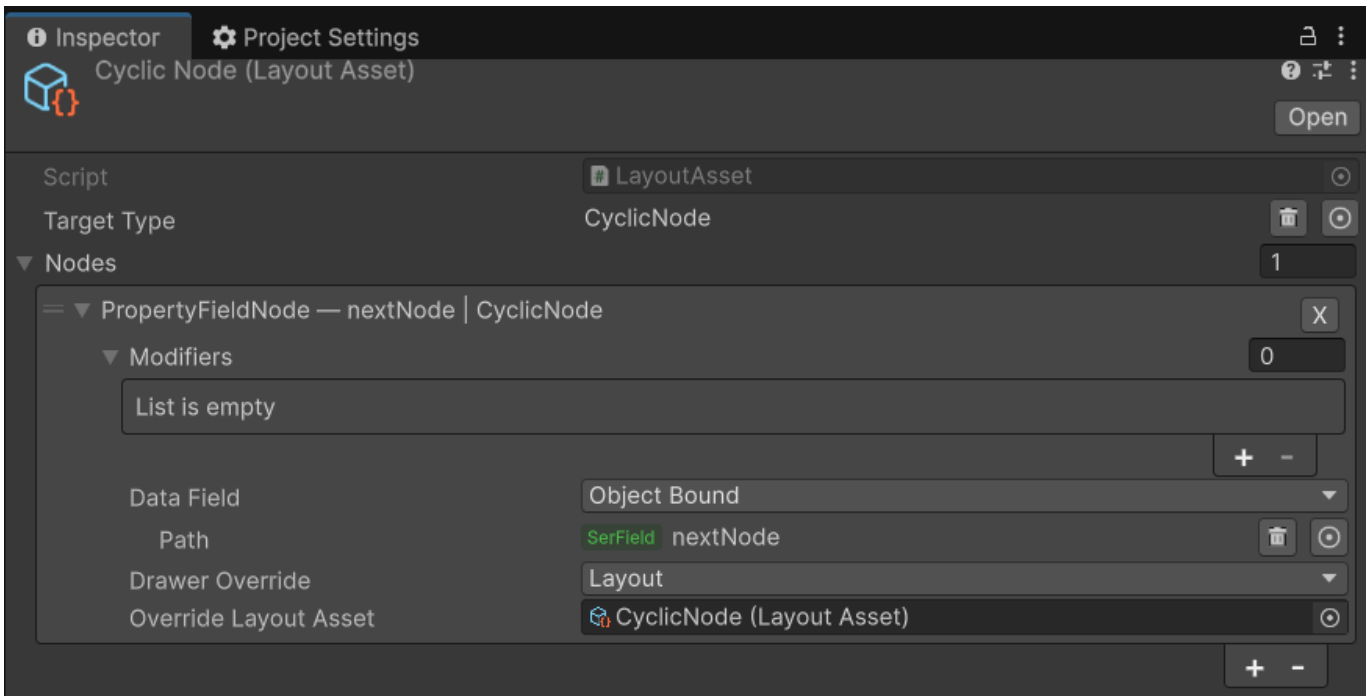


Рисунок Г.8 – Конфігурація, що рекурсивно передає виконання собі ж

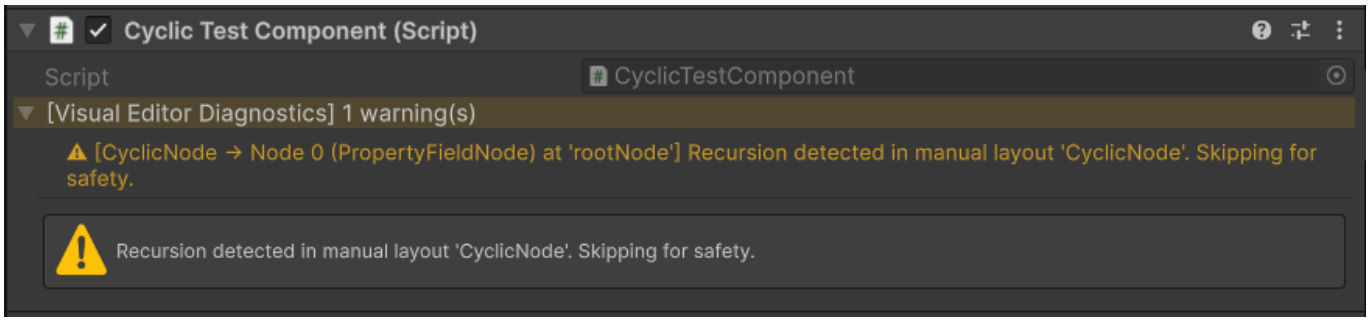


Рисунок Г.9 – Вигляд повідомлення системи діагностики при спрацюванні захисту від рекурсивних конфігурацій

ДОДАТОК Д

(обов’язковий)

КЕРІВНИЦТВО КОРИСТУВАЧА

Інструмент візуального конструювання інтерфейсів

Даний інструмент для Unity дозволяє створювати власні користувацькі інспектори та відмальовувачі для скриптів без необхідності написання коду (CustomEditor, CustomPropertyDrawer). Уся робота виконується у візуальному редакторі з підтримкою оновлення в реальному часі.

Ключові можливості:

- візуальне конструювання ієрархії інспектора за допомогою структурних вузлів;
- автоматичне перетворення стандартних атрибутів скрипта (наприклад, Header, Range) на готові макети;
- зміна поведінки та стилю полів за допомогою системи модифікаторів;
- миттєве оновлення користувацького інтерфейсу в інспекторі при зміні макета без очікування рекомпіляції коду;
- інтегрована система діагностики, що попереджає про помилки конфігурації або несумісність елементів.

Покрокові дії для створення макету інспектора та його реєстрації

Крок 1. Створення файлу макета (Layout Asset)

Для початку роботи необхідно створити файл, який зберігатиме конфігурацію вашого інтерфейсу:

1. у вікні Project натисніть праву кнопку миші у потрібній папці;
2. перейдіть до меню Create -> Visual Editor -> Layout Asset (рисунок Д.1);
3. вкажіть ім'я для новоствореного файлу.

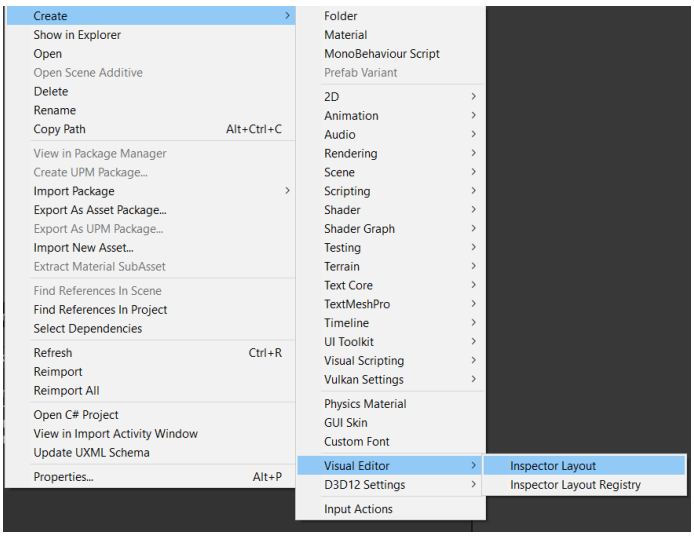


Рисунок Д.1 – Шлях у контекстному меню для створення макету

Крок 2. Налаштування макета та автоматичне розбирання

Ви можете дозволити системі автоматично згенерувати базовий макет на основі вже існуючого коду:

1. виділіть створений файл Layout Asset;
2. у панелі інспектора знайдіть поле Target Type та оберіть цільовий тип вашого C#-скрипта за допомогою вікна пошуку (рисунок Д.2);
3. натисніть кнопку Disassemble, підтвердіть дію (рисунок Д.3);
4. система автоматично згенерує вузли відповідно до серіалізованих полів та атрибутів вашого скрипта.

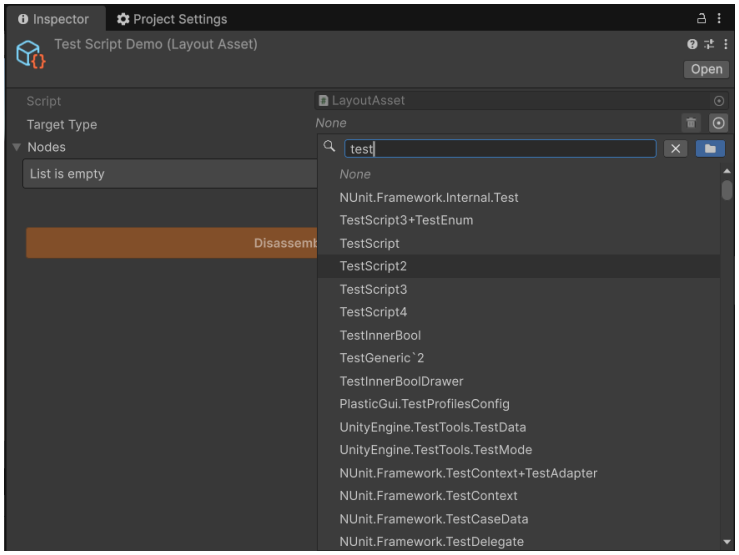


Рисунок Д.2 – Пошук цільового типу через TypeSelectionWindow

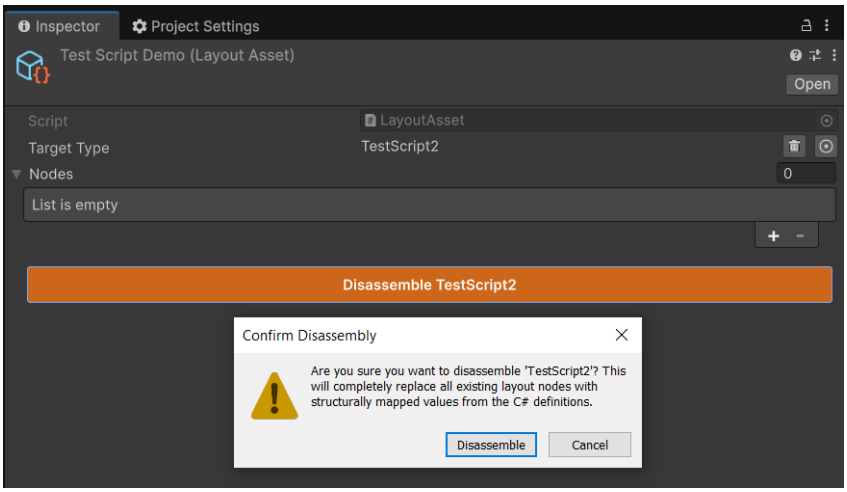


Рисунок Д.3 – Процес автоматичного розбирання

Крок 3. Додавання та налаштування вузлів (Nodes)

Якщо ви конструюєте інтерфейс з нуля або доповнюєте згенерований, використовуйте систему вузлів:

1. у переліку вузлів натисніть «+», щоб додати новий вузол;
2. натисніть кнопку «Select LayoutNode…» для нового вузла;
3. у спливаючому вікні вибору типу оберіть необхідний вузол, наприклад, PropertyFieldNode для показу змінної або FoldoutNode для створення групи, що може згортатися (рисунок Д.4);
4. розгорніть доданий вузол та налаштуйте його параметри (наприклад, вкажіть цільову змінну для візуалізації DataField.Path у PropertyFieldNode (рисунок Д.5) або напрямок групи для GroupNode);
5. за потреби змініть режим відмальовування поля у випадаючому списку (рисунок Д.6), рекомендується використовувати Native для полів, що у своєму скрипті користуються стандартними атрибутами Unity.

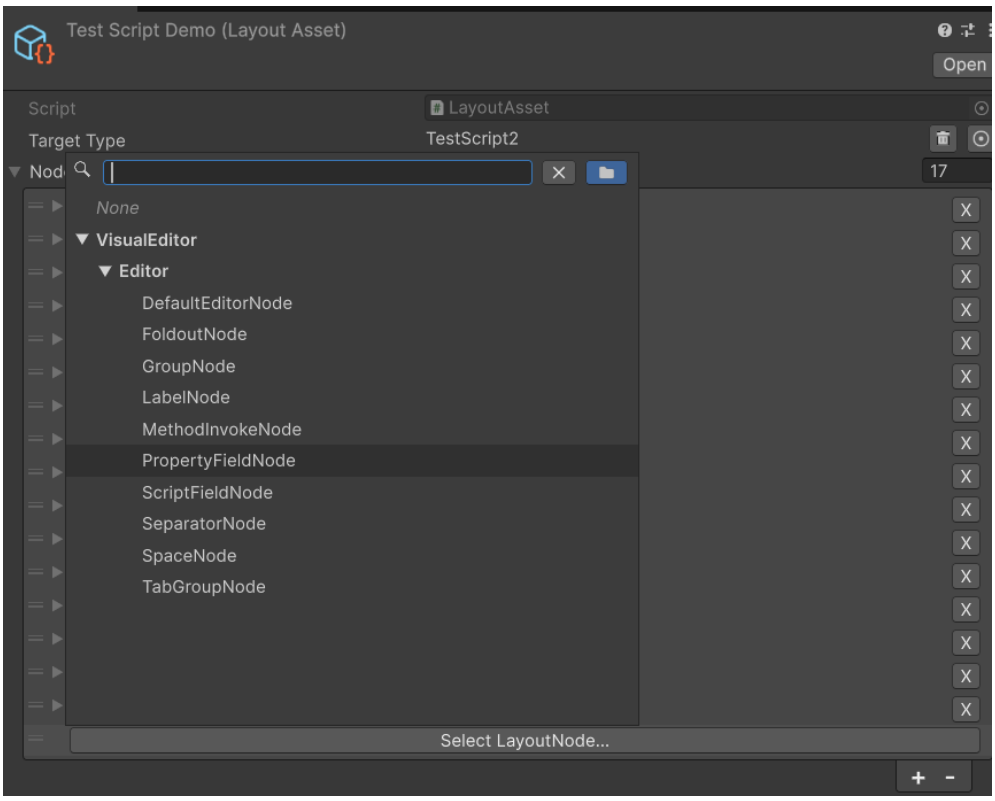


Рисунок Д.4 – Панель вибору типу вузла

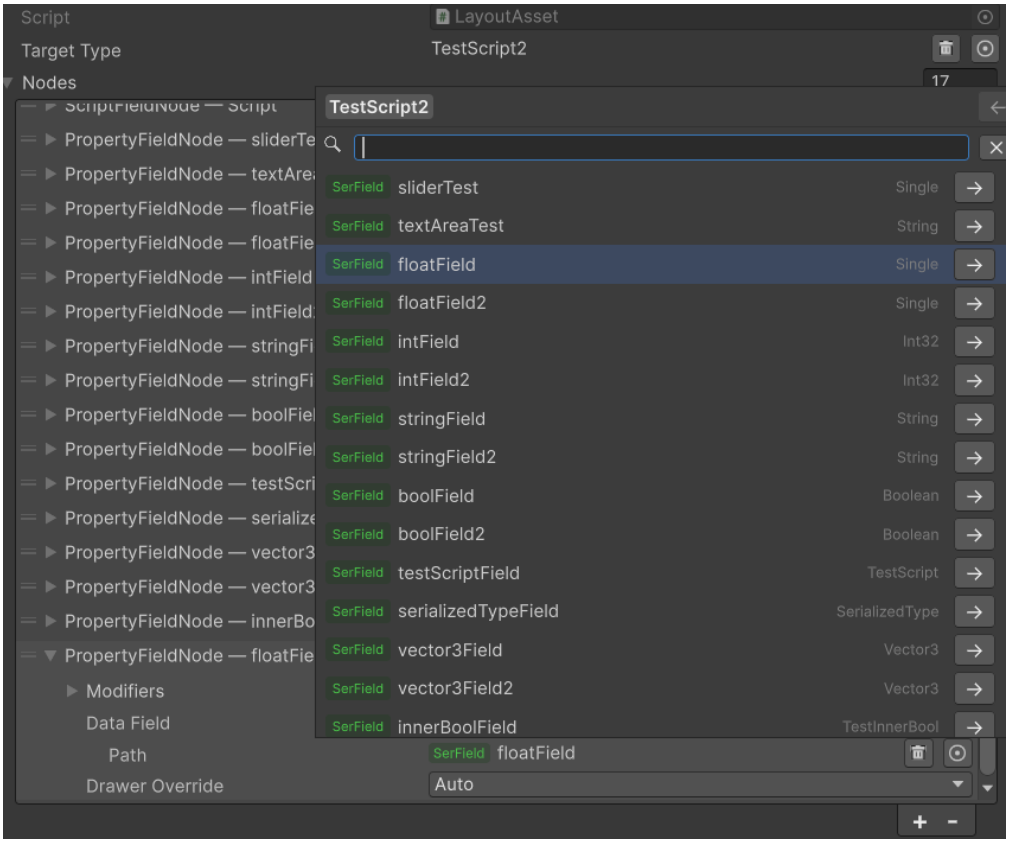


Рисунок Д.5 – Пошук члену для візуалізації через MemberSelectionWindow

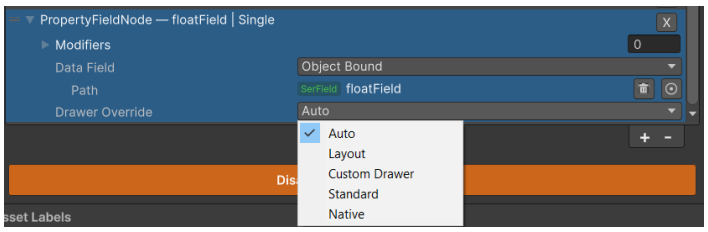


Рисунок Д.6 – Зміна типу відмальовки для конкретного поля

Крок 4. Застосування модифікаторів (Modifiers)

Для зміни візуального вигляду або додавання логіки до властивості використовуйте модифікатори:

1. виділіть налаштований структурний вузол у списку;
2. у нижній частині налаштувань вузла знайдіть перелік Modifiers та натисніть «+», щоб додати новий модифікатор;
3. натисніть кнопку «Select LayoutModifier…» для нового модифікатора;
4. оберіть потрібний модифікатор, наприклад, SliderModifier, щоб перетворити числове поле на повзунок, або VisibilityModifier для динамічного приховування поля (рисунок Д.7);
5. задайте параметри обраного модифікатора, наприклад, вкажіть мінімальне та максимальне значення для повзунка, (рисунок Д.8).

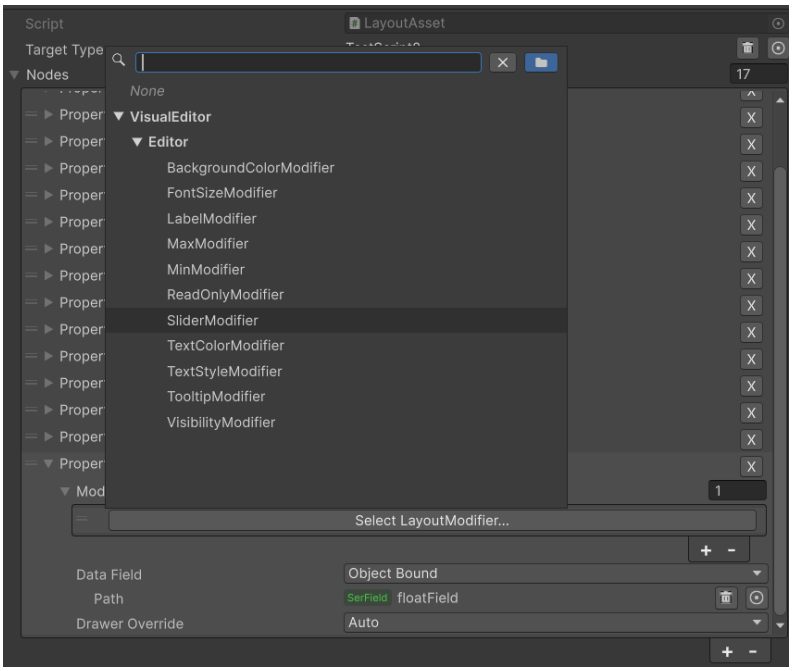


Рисунок Д.7 – Панель вибору типу модифікатора

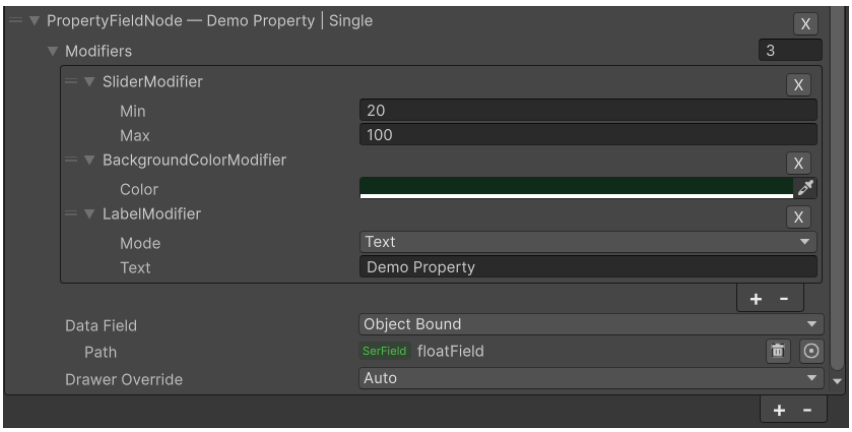


Рисунок Д.8 – Налаштування модифікаторів вузла

Крок 5. Реєстрація макета у глобальній системі

Щоб Unity почав використовувати створений макет для відмальовування об'єктів на сцені, його потрібно зареєструвати:

1. знайдіть у проєкті файл InspectorLayoutRegistry (або створіть його через контекстне меню);
2. у блоці Inspector Entries додайте новий елемент до списку;
3. оберіть стратегію Layout, а у поле конфігурації перетягніть створений файл Layout Asset (рисунок Д.9);
4. виділіть будь-який ігровий об'єкт на сцені, який містить ваш скрипт, і переконайтеся, що його інспектор тепер відмальовується за новим макетом (рисунок Д.10).

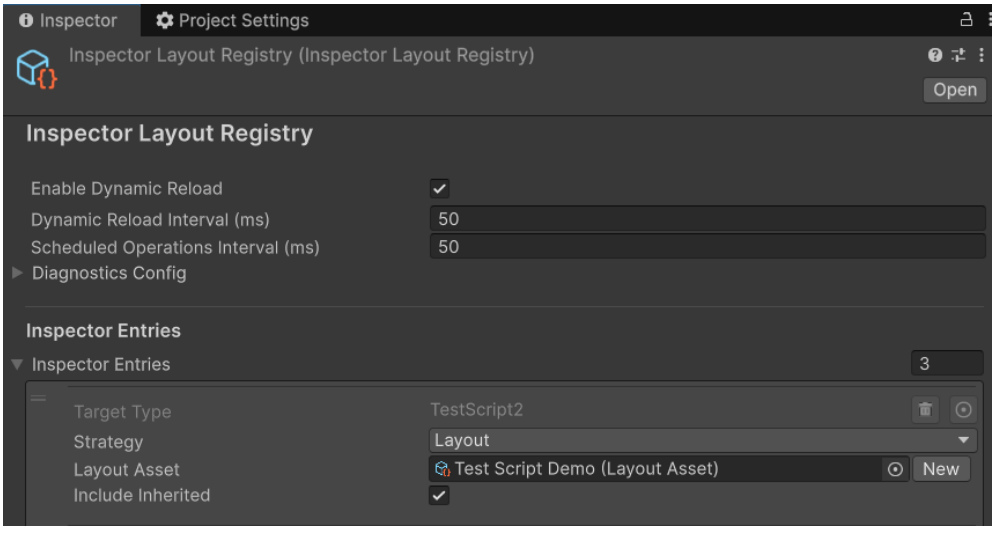


Рисунок Д.9 – Внесення нової конфігурації до реєстру

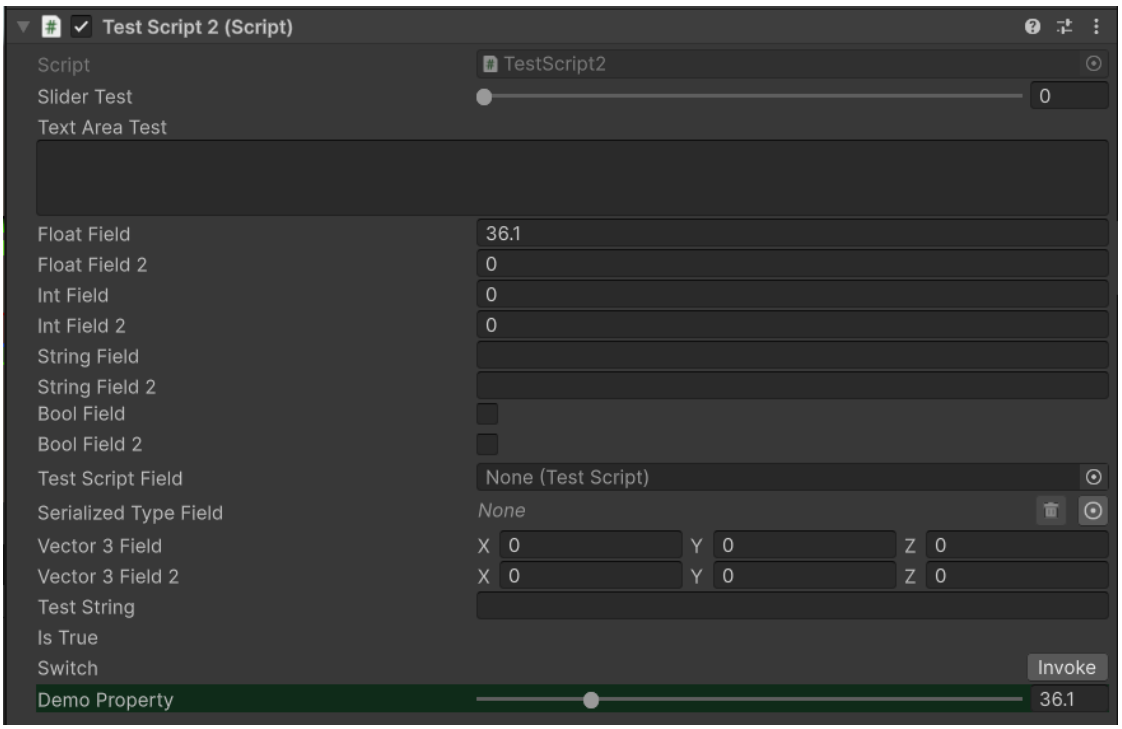


Рисунок Д.10 – Результат макету (новий вузол «Demo Property»)

Важливі примітки

- зміни, які ви вносите в Layout Asset, миттєво відображатимуться в інспекторах цільових об'єктів (якщо увімкнений режим динамічного перезавантаження), тому ви можете налаштовувати інтерфейс, тримаючи обидва вікна відкритими;
- система має вбудований захист: при спробі додати модифікатор до несумісного типу даних (наприклад, повзунок до тексту) або при створенні логічного зациклення, плагін виведе інформаційне повідомлення червоного або жовтого кольору безпосередньо у вікні інспектора, не допускаючи помилок редактора;
- при виникненні будь-яких помилок конфігурації, система виведе відповідні повідомлення в збудованому інспекторі та/або в консолі;
- для створення макету для властивості виконуються аналогічні дії, проте сам макет додається до переліку Property Drawer Entries в реєстрі;
- перелік усіх вузлів та модифікаторів можна переглянути у таблицях додатку Б.

Налаштування реєстру

Реєстр є місцем зберігання усіх актуальних конфігурацій для інспекторів та властивостей (рисунок Д.11), окрім цього він також зберігає глобальні налаштування системи, такі як:

– Enable Dynamic Reload визначає чи увімкнена система динамічного перезавантаження при зміні макетів;
– Dynamic Reload Interval (ms) визначає затримку оновлення представлень при зміні макетів у мілісекундах, працює лише при увімкненому Enable Dynamic Reload, занадто низькі значення можуть сповільнити роботу редактора;
– Scheduled Operations Interval (ms) визначає інтервал між динамічними перевірками станів вузлів та модифікаторів, менші значення збільшують відчуття плавності, проте можуть сповільнювати роботу редактора, якщо обрані обчислення занадто складні;
– Diagnostics Config визначає шляхи виводу різних типів повідомлень системи (Info, Warning, Error), з шляхів є Summary (показ у агрегованому списку повідомлень зверху побудованого інспектора чи елемента), PerElement (показ прямо над цільовим елементом) та Console (вивід агрегованих помилок у консоль), можна використовувати декілька шляхів одночасно.

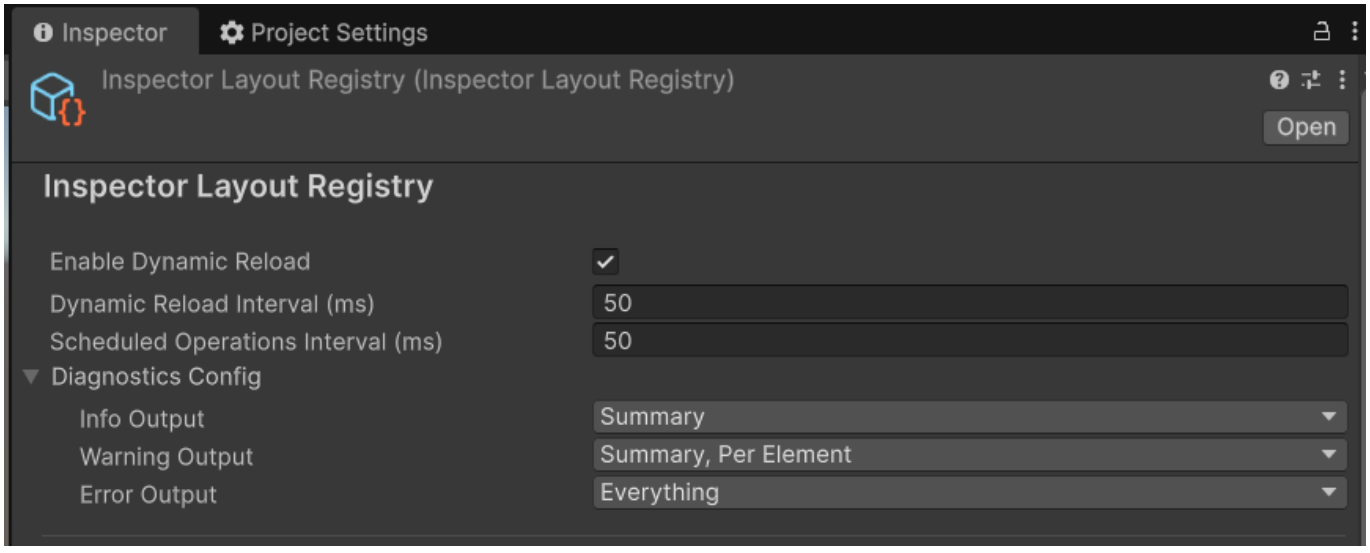


Рисунок Д.11 – Глобальні налаштування реєстру

Приклади результатів, яких можна досягнути (рисунки Д.12-Д.15)

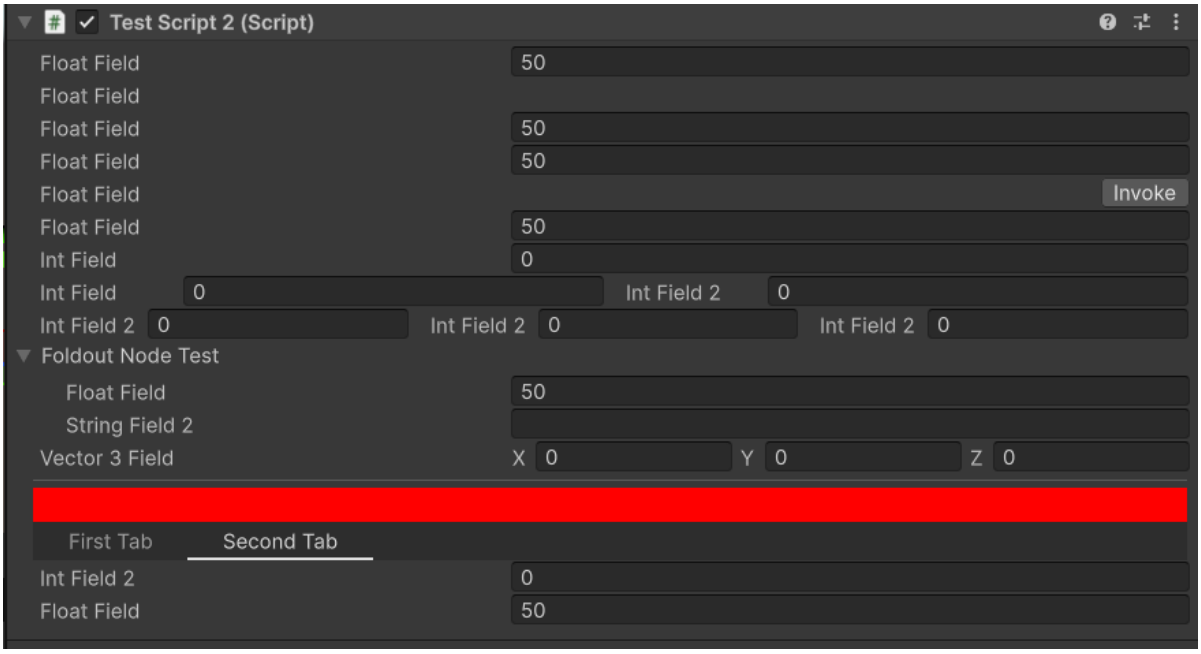


Рисунок Д.12 – Результат макету з багатьма типами групування

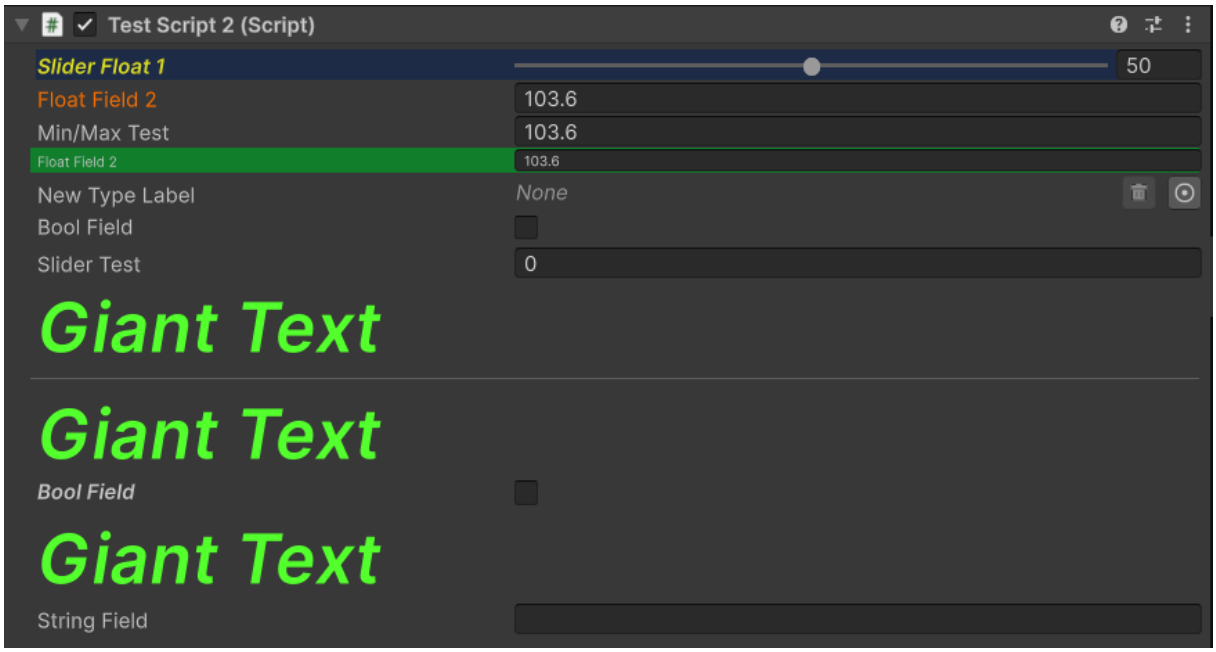


Рисунок Д.13 – Результат макету з багатьма типами стилізації

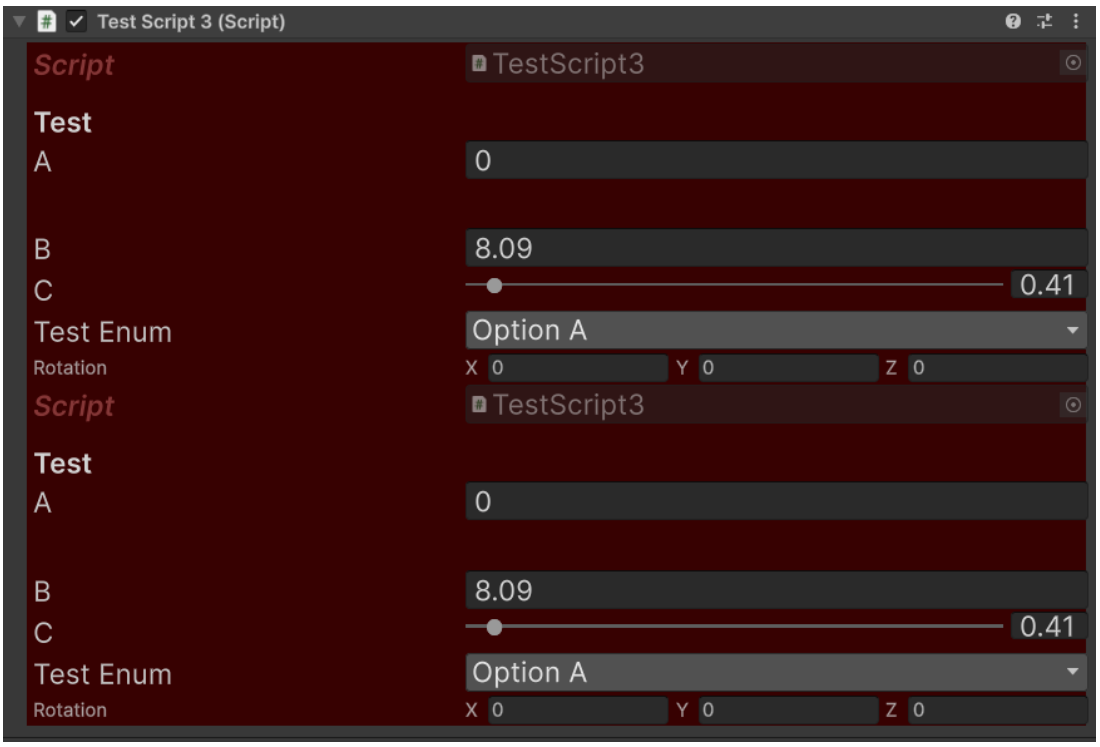


Рисунок Д.14 – Результат макету, що використовує DefaultEditorNode для роботи із стандартним інспектором у вигляді одного вузла

Рисунок Д.15 – Результат макету, що відмальовує одне поле тричі, використовуючи різні способи в межах одного макету

ДОДАТОК Е

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кафедра інженерії програмного забезпечення

Кваліфікаційна робота на тему: «Інструмент візуального створення редакторів користувача для рушія Unity»

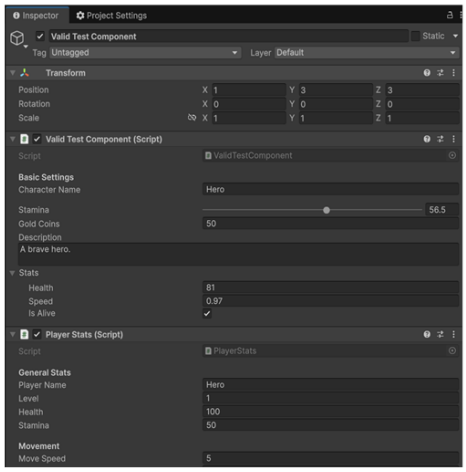
Виконав:
Студент IV курсу, групи ІПЗ-22-1 Юрчук Роман Володимирович

Керівник:
канд. пед. наук, доцент Праворська Наталія Іванівна

Рисунок Е.1 – Слайд 1

Актуальність теми

- Використання ігрових рушіїв проводиться не тільки для розробки ігор, а ще й для створення відео, фільмів, анімацій, інтерактивних програм, симуляцій та тренажерів.
- Зростання обсягів інформації, зумовлює потребу у покращенні інструментів для роботи із нею.
- Наразі Unity є домінуючим рішенням на ринку, через свою універсальність та велику кількість ресурсів.
- Структура даних задається виключно через код, автоматичне представлення строго відповідає цій структурі.
- Багато спеціалістів у сфері не володіють навіть базовими навичками програмування.
- Загальна тенденція розвитку low-code та no-code.



Приклад інспектора Unity

Рисунок Е.2 – Слайд 2

Мета та Завдання

Метою кваліфікаційної роботи є розроблення інструментального засобу для візуального створення користувацьких редакторів у середовищі Unity, що дозволяє підвищити ефективність вказаного процесу шляхом автоматизації генерації інтерфейсів та усунення необхідності рекомпіляції коду під час їхнього налаштування, а також знижує поріг необхідних знань для початку роботи.

Завдання:

- виконати аналіз предметної області, вивчити сучасні тенденції та методики створення розширень редактора Unity;
- проаналізувати наявні програмні аналоги та інструментальні рішення, виявити їхні переваги та недоліки;
- встановити особливості та сформулювати перелік функціональних і нефункціональних вимог;
- розробити архітектуру та структуру інструменту, що дозволить інтегрувати систему в середовище Unity;
- розробити моделі та алгоритми функціонування візуального конструктора;
- виконати програмну реалізацію засобу з використанням мови програмування C# та API Unity;
- провести тестування для підтвердження працездатності та відповідності описаним вимогам.

Рисунок Е.3 – Слайд 3

Дослідження предметної області

Жорстка прив'язка до коду. Стандартний механізм серіалізації Unity прив'язує представлення до коду. Будь-яка зміна інтерфейсу призводить до компіляції (3-60 сек).

Перевизначення представлень. Використання атрибутів CustomEditor та CustomPropertyDrawer для визначення спеціалізованих скриптів, що відповідають за відмальовку.

Засоби представлення. IMGUI – старіше рішення, що відмальовує все представлення кожного кадру. UI Toolkit використовує збереження стану, аналогічно до підходу сучасних вебсайтів.

Серіалізація. Unity використовує класи SerialiazedObject та SerializedProperty, щоб зберігати інформацію про дані. Ці ж класи використовуються для створення представлень.

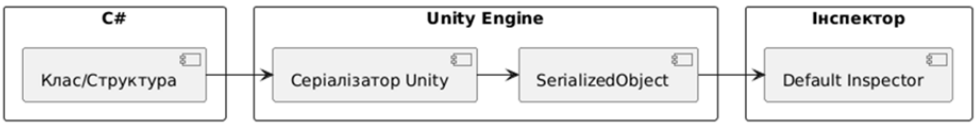


Схема стандартного процесу серіалізації

Рисунок Е.4 – Слайд 4

Аналіз наявного програмного забезпечення

Тип рішення / Назва	Метод взаємодії	Гнучкість логіки	Швидкість ітерації
Атрибутивні фреймворки (Alchemy, Artificetoolkit, Odin Inspector)	Лише через програмний код (C#)	Висока для програміста	Низька (через рекомпіляцію)
Odin Visual Designer	Візуальне компонування вузлів	Середня	Висока
Vibe: No-Code Inspector	Ієрархічне дерево елементів	Низька	Висока
Unity UI Builder	Візуальна верстка (подібна до вебсайтів)	Висока для програміста	Висока (при відсутності логіки)

Рисунок Е.5 – Слайд 5

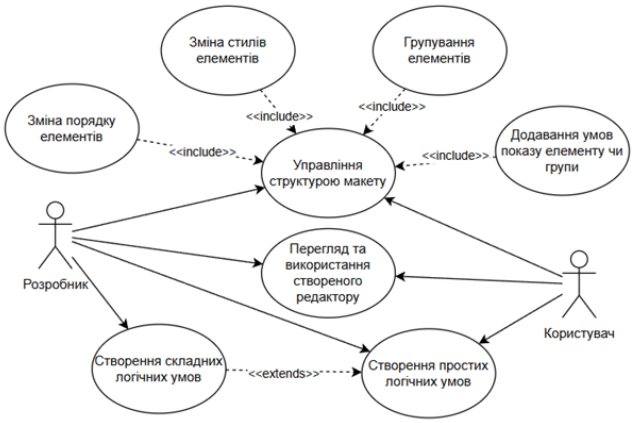
Функціональні та нефункціональні вимоги

Функціональні вимоги:

- механізм макетів, що керуються даними;
- візуальний конструктор для побудови макетів;
- автоматична побудова інтерфейсів з макету;
- гнучке керування джерелами даних для параметрів;
- візуалізація несеріалізованих даних;
- гнучка побудова логіки;
- підтримка інших рішень та відмальовувачів.

Нефункціональні вимоги:

- підтримка Unity 6000.3 LTS та вище;
- затримка оновлення не більше 300 мс;
- використання збереження стану;
- безпечне відловлювання помилок;
- автоматична валідація типів;
- підтримка стандартної системи скасування та повторення дій.



Діаграма варіантів використання

Рисунок Е.6 – Слайд 6

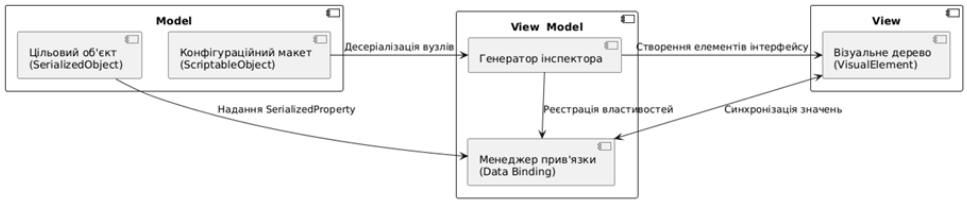
Вибір архітектури та шаблонів проєктування

Архітектурний шаблон MVVM

Використовується для ізолювання логіки представлення та моделей одне від одного, збільшуючи гнучкість та масштабованість системи. Відповідає засадам роботи UI Toolkit.

Шаблони проєктування

- **«Фабричний метод»** – створення відповідних візуалізацій на основі вузлів;
- **«Стратегія»** – динамічне перемикання логіки між режимами джерел даних;
- **«Проксі»** – заміщення несеріалізованих даних;
- **«Службовий локатор»** – інкапсуляція отримання модулів системи;
- **«Одинак»** – створення глобального реєстру конфігурацій.



Діаграма компонентів «MVVM у контексті системи»

Рисунок Е.7 – Слайд 7

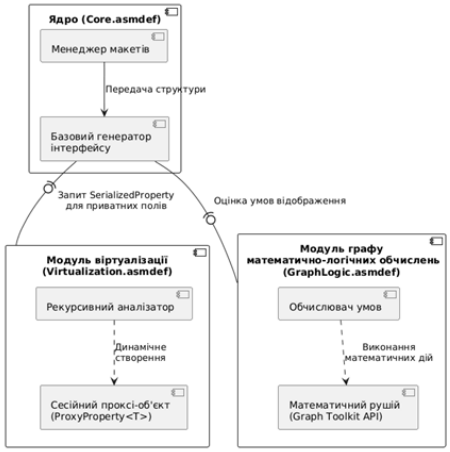
Декомпозиція підсистем, залежностей та інтерфейсів

Ядро. Центральний модуль, що виконує базові функції, включаючи роботу із макетами, їхнє читання та перетворення на представлення.

Віртуалізація. Модуль-міст, що створює проксі-об'єкти для несеріалізованих даних.

Граф обчислень. Модуль для обробки математичних та логічних виразів.

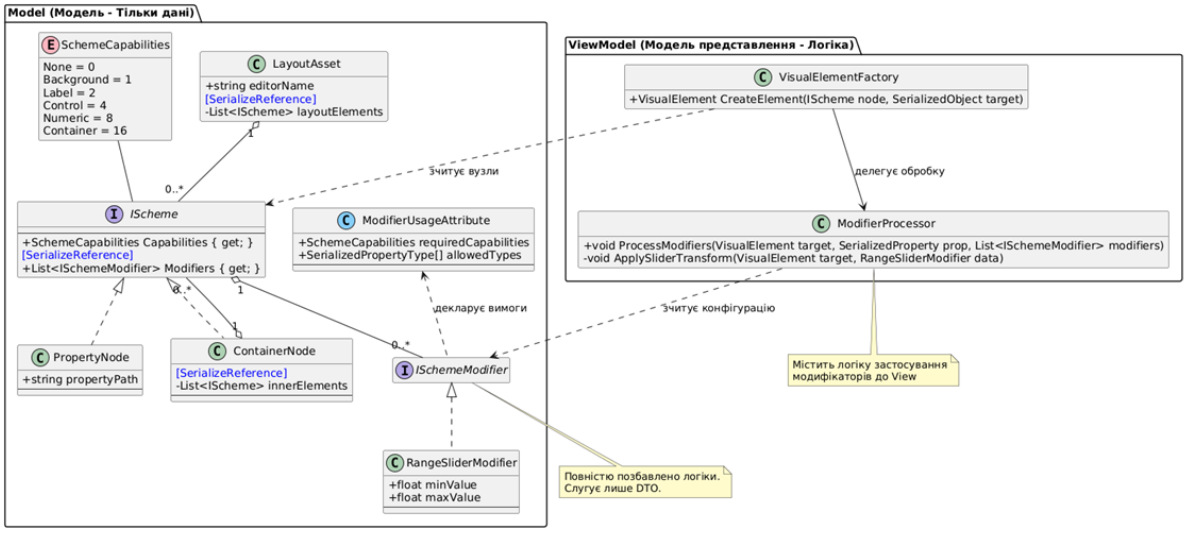
Діаграма класів «Стратегія при виборі типу відмальовувача»



Діаграма компонентів «Модулі системи»

Рисунок Е.8 – Слайд 8

Проєктування структур даних



Діаграма класів «Структура даних макету»

Рисунок Е.9 – Слайд 9

Технологічний стек проєкту

Мова програмування C#

Сучасна розвинута мова програмування, що підтримується рушієм Unity

C# Reflection

Механізм, для динамічного аналізу структур класів та доступу до приватних членів

Unity 6000.3 LTS

Актуальна версія цільового ігрового рушія із довготривалою підтримкою

UI Toolkit (Unity)

Декларативна система побудови інтерфейсів із підтримкою механізму збереження стану

Graph Toolkit API (Unity)

Основа для реалізації вузлового редактора математичних та логічних операцій

Project Auditor (Unity)

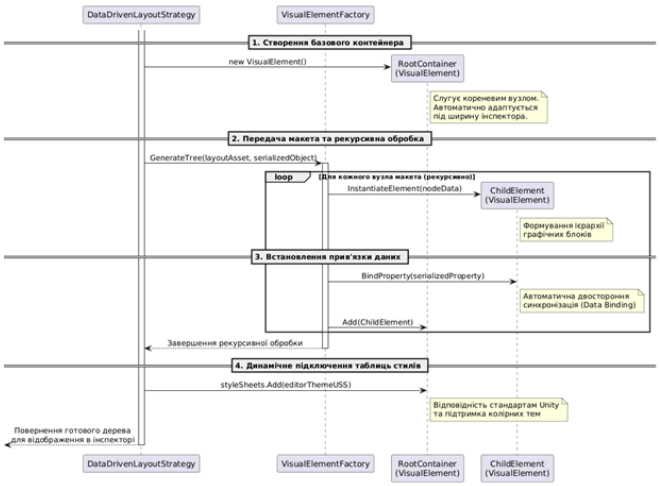
Інструмент статичного аналізу коду та проєкту, з метою пошуку вузьких місць у продукивності

Рисунок Е.10 – Слайд 10

Реалізація модулів системи

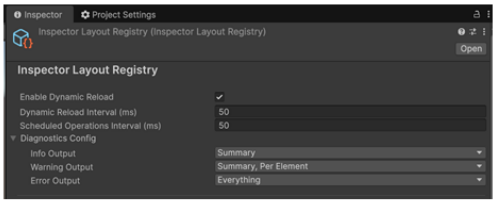
Діаграма комунікації «Взаємодія у Ядрі»

Діаграма станів «Перезавантаження у реальному часі»

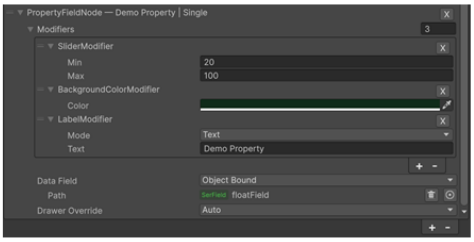


Діаграма послідовності «Формування візуального дерева»

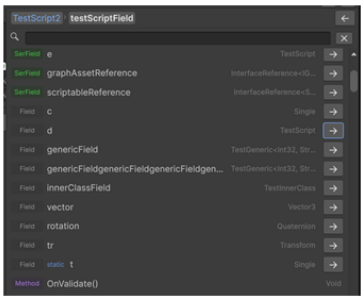
Реалізація інтерфейсу



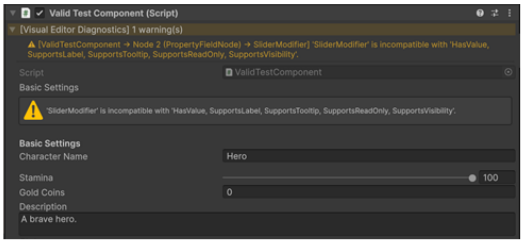
Глобальні налаштування реєстру



Налаштування вузла для поля



Вікно для вибору члену цільового об'єкту



Вигляд системи діагностики

Рисунок Е.11 – Слайд 11

Рисунок Е.12 – Слайд 12

Вимоги до технічного та програмного забезпечення

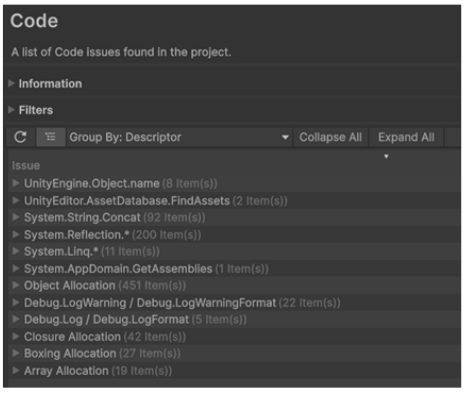
Характеристика	Вимога або рекомендація
Базове програмне забезпечення	редактор Unity версії 6000.3 LTS або новішої
Операційна система (ОС)	Windows 10 21H1 (64-bit), Windows 11 21H2 (64-bit), macOS 13 або новіші; підтримувані дистрибутиви Linux (64-bit), Ubuntu 22.04, Ubuntu 24.04
Процесор (CPU)	сумісний з архітектурою x64 та набором інструкцій SSE2, Arm64 або ж Apple Silicon
Оперативна пам'ять (RAM)	мінімально 8 ГБ, рекомендовано 16 ГБ та більше для комфортної роботи, особливо для великих проєктів
Зовнішня пам'ять (Storage)	SSD-накопичувач, до 10 МБ вільного простору для розширення та 10–20 ГБ для рушія Unity
Спеціальні пристрої	не вимагаються (достатньо звичайної клавіатури, миші та монітора)

Рисунок Е.13 – Слайд 13

Тестування та валідація результатів

Project Auditor виявив 892 рекомендації (249 помірних, решта – інформаційні).

Більшість із них стосуються рефлексії, інтерполяції рядків та виділення пам’яті.



Project Auditor – Статичний аналіз коду

Тестування «Чорна скринька»:

- перехоплення відмальовки інспектора (успішно);
- автоматичне генерація представлення із макета (успішно);
- автоматичне розбирання макета (частково успішно);
- валідація несумісних модифікаторів (успішно);
- захист від зациклення подій (успішно);
- захист від рекурсії у макетах (успішно);
- підтримка зворотньої сумісності (успішно).

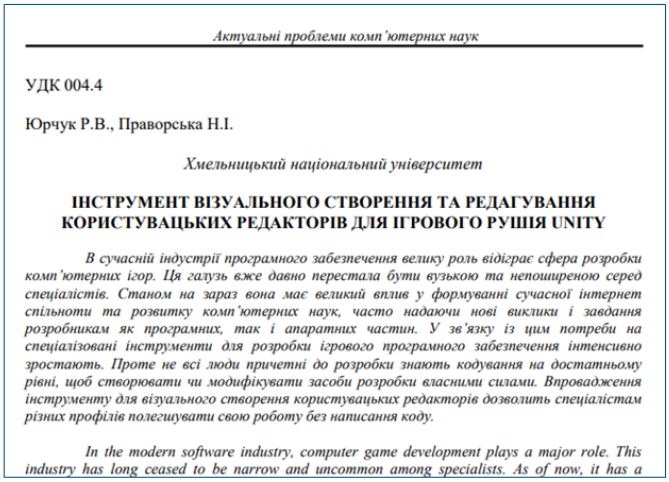
Тестування підтвердило правильне функціонування інструменту відповідно до визначених вимог, навіть у нетипових ситуаціях та під час критичних навантажень.

Рисунок Е.14 – Слайд 14

Участь у конференції

Роботу було також опубліковано в:

ЗБІРНИК НАУКОВИХ ПРАЦЬ за матеріалами XVII Всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2025»



Актуальні проблеми комп'ютерних наук

УДК 004.4

Юрчук Р.В., Праворська Н.І.

Хмельницький національний університет

ІНСТРУМЕНТ ВІЗУАЛЬНОГО СТВОРЕННЯ ТА РЕДАГУВАННЯ КОРИСТУВАЦЬКИХ РЕДАКТОРІВ ДЛЯ ІГРОВОГО РУШІЯ UNITY

В сучасній індустрії програмного забезпечення велику роль відіграє сфера розробки комп'ютерних ігор. Ця галузь вже давно перестала бути вузькою та непоширеною серед спеціалістів. Станом на зараз вона має великий вплив у формуванні сучасної інтернет спільноти та розвитку комп'ютерних наук, часто надаючи нові виклики і завдання розробникам як програмних, так і апаратних частин. У зв'язку із цим потреби на спеціалізовані інструменти для розробки ігрового програмного забезпечення інтенсивно зростають. Проте не всі люди причетні до розробки знають кодування на достатньому рівні, щоб створювати чи модифікувати засоби розробки власними силами. Впровадження інструменту для візуального створення користувацьких редакторів дозволить спеціалістам різних профілів полегшувати свою роботу без написання коду.

In the modern software industry, computer game development plays a major role. This industry has long ceased to be narrow and uncommon among specialists. As of now, it has a

Юрчук Р. В., Праворська Н. І. Інструмент візуального створення та редагування користувацьких редакторів для ігрового рушія Unity. *Актуальні проблеми комп'ютерних наук АПКН.* 2025. С. 494–497.

Рисунок Е.15 – Слайд 15

Висновки та результати роботи

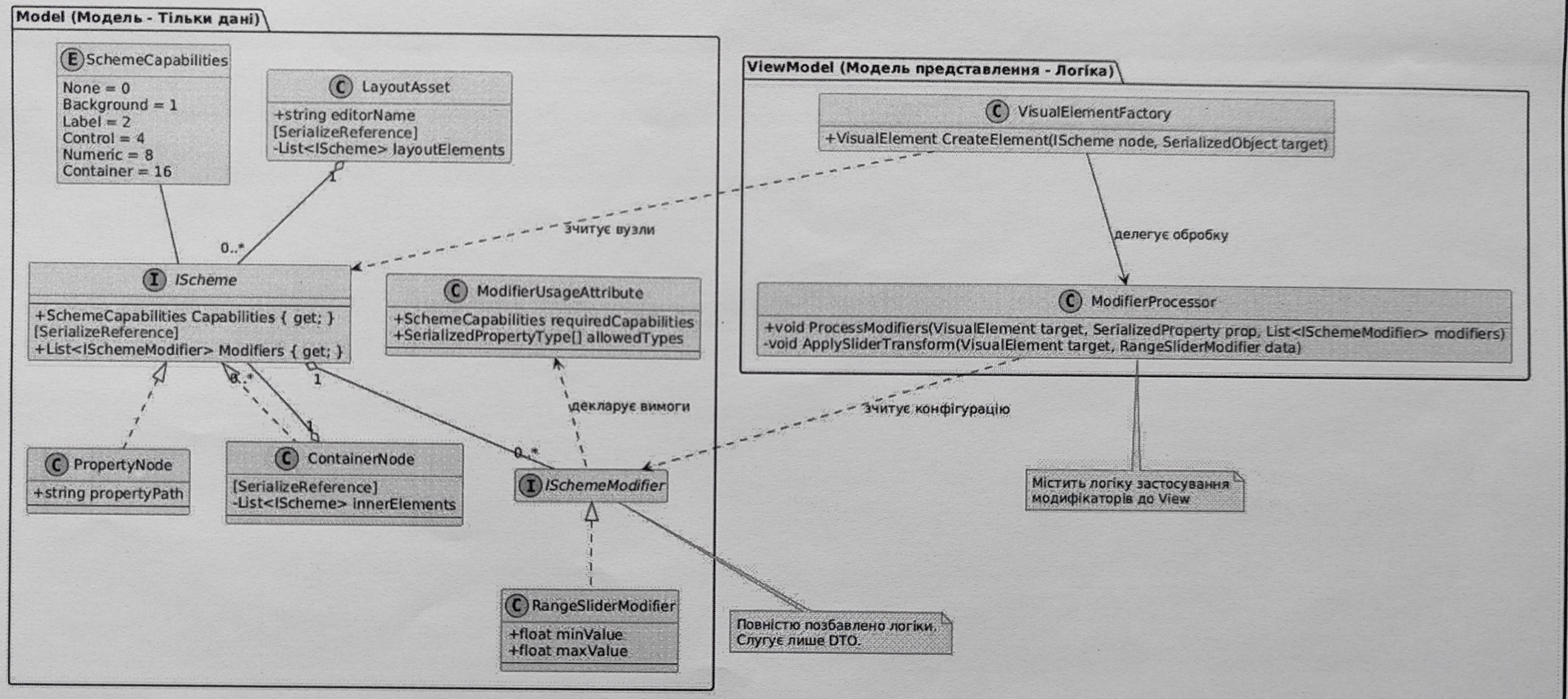
Завдання	Результат виконання
Виконати аналіз предметної області	Досліджено механізми серіалізації Unity, виявлено проблему жорсткої прив'язки інтерфейсу до коду та обґрунтовано необхідність no-code/low-code підходів.
Проаналізувати наявні програмні аналоги	Проведено порівняння систем. Підтверджено потребу в інструменті, що надає гнучкість у виборі стратегій відмальовування та зручний механізм налаштування динамічних обчислень.
Встановити функціональні та нефункціональні вимоги	Визначено вимоги до продуктивності, надійності, безпеки та ключового функціоналу: візуальний конструктор, автоматична збірка, графова логіка, підтримка інших відмальовувачів.
Розробити архітектуру та структуру інструменту	Спроєктовано модульну систему на базі шаблону MVVM, яка забезпечує слабку зв'язність Ядра з підсистемами віртуалізації та графів.
Розробити моделі та алгоритми функціонування візуального конструктора	Створено структури даних поділені на макети, вузли та модифікатори. Використано алгоритми генерації інтерфейсу та вибору джерел параметрів за шаблоном Стратегія.
Виконати програмну реалізацію інструменту	Написано ядро системи: службовий локатор підсистем, візуальний редактор, перехоплювач інспекторів, модуль розбирання, система діагностики, динамічне перезавантаження.
Провести тестування	Успішно пройдено статичний аналіз коду та перевірку усього інструменту методом «Чорної скриньки», що включала всі функціональні та нефункціональні вимоги до ядра.

Мета роботи досягнута: розроблено ефективний інструментарій для візуального створення користувацьких редакторів.

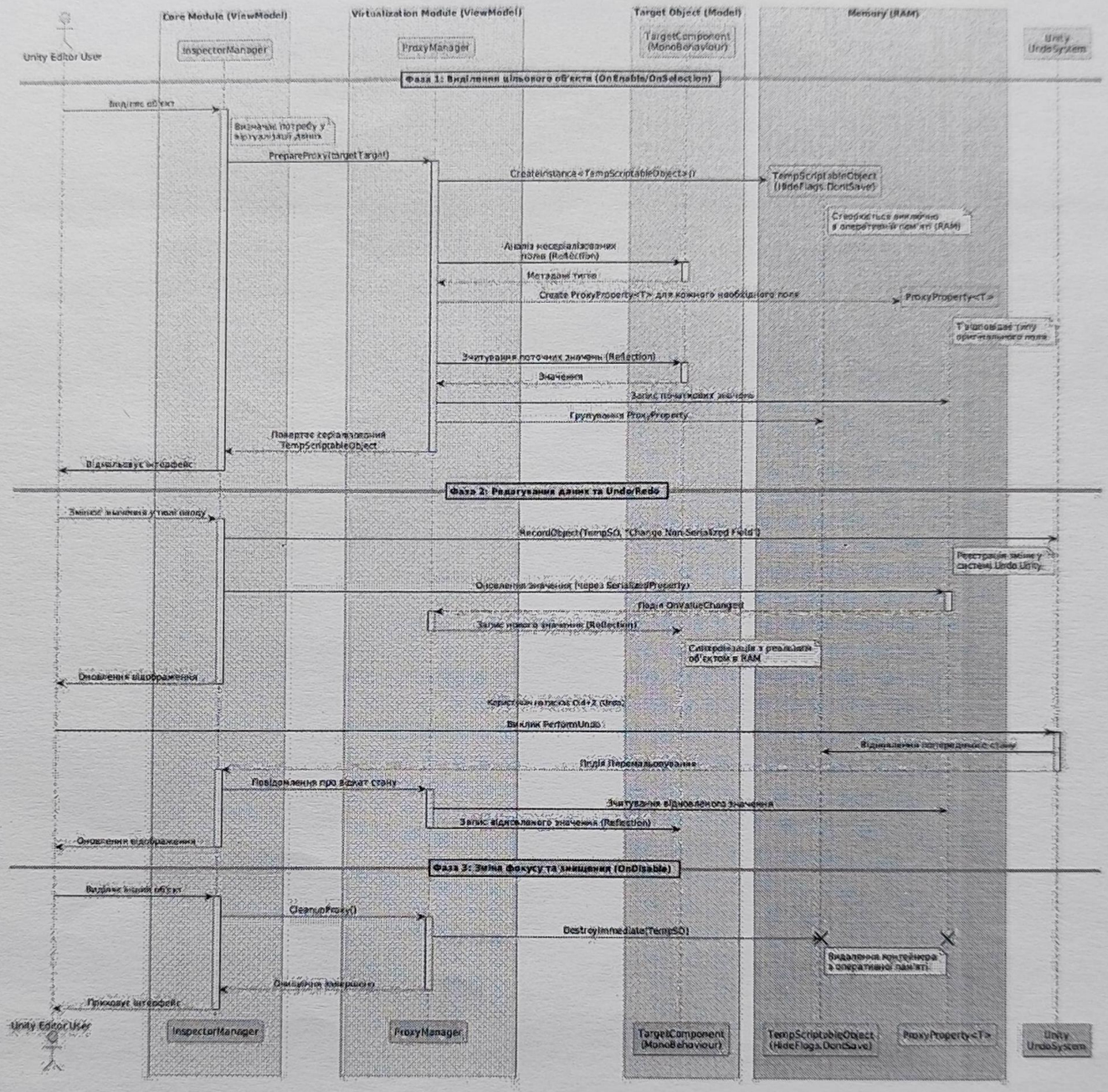
Рисунок Е.16 – Слайд 16

Рисунок Е.17 – Слайд 17

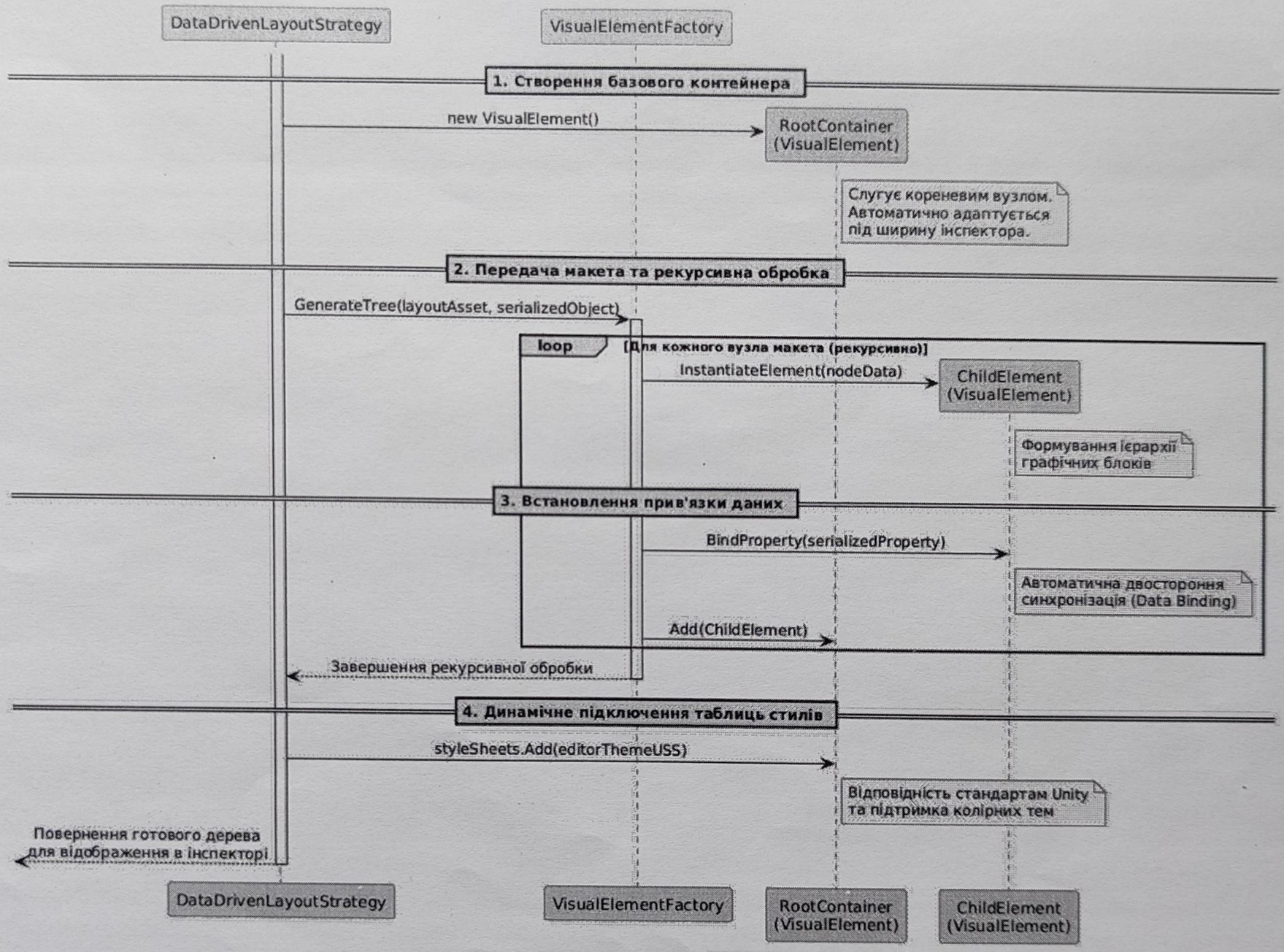
ГРАФІЧНІ МАТЕРІАЛИ



					КвРІПЗ.2201119.01.24.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Діаграма класів схем даних та модифікаторів макету	Літера / Маса / Масштаб
Розробив		Юрчук Р.В.		27.05		
Керівник		Праворська Н.І.		27.05		Аркуш 1 / Аркушів 3
Консульт.						
Н. Контр.		Яшина О.М.		27.05		ХНУ, ІПЗ-22-1
Зав. каф.		Бедратюк Л.П.		27.05		



					КвРІПЗ.2201119.01.24.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Діаграми послідовностей системи віртуалізації	Літера / Маса / Масштаб
Розробив		Юрчук Р.В.		27.05		
Керівник		Праворська Н.І.		27.05		Аркуш 2 / Аркушів 3
Консульт.						
Н. Контр.		Яшина О.М.		27.05		ХНУ, ІПЗ-22-1
Зав. каф.		Бедратюк Л.П.		27.05		



					КвРІПЗ.2201119.01.24.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Діаграма послідовності формування візуального дерева	Літера / Маса / Масштаб
Розробив		Юрчук Р.В.		27.05		
Керівник		Праворська Н.І.		27.05		Аркуш 3 / Аркушів 3
Консульт.						
Н. Контр.		Яшина О.М.		27.05		ХНУ, ІПЗ-22-1
Зав. каф.		Бедратюк Л.П.		27.05		

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Юрчука Романа Володимировича
факультет ІТ, IV курс, група ІПЗ-22-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

02 вересня 2025
дата

підпис

Anti-Plagiarism (http://ap.km.ua) v-16.718

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: UA, US, RU. **Помилок в документах: 13%**

	Документ		Сумарний збіг по Базі Даних	
ID: 271749 Назва: БКР_Інструмент візуального створення редакторів користувача для рушія Unity Додано в БД: 2026-05-20 Автора: Роман ЮРЧУК Керівники: канд. пед. наук, доцент Наталія ПРАВОРСЬКА Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	116608	690	3473 (3%)	43 (6%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Роман ЮРЧУК

Співавтор:

Назва: Інструмент візуального створення редакторів

Науковий керівник: канд. пед. наук, доцент Наталія ПРАВОРСЬКА

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:2.7%

Коефіцієнт подібності 2:0.88%

Мікропробіли: 22

Заміна букв: 0

Інтервали: 11

Білі знаки: 0

Дата створення звіту: 2026-05-20 01:12:57.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 25.05.2026

експерт

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Юрчук Роман Володимирович

Тема Інструмент візуального створення редакторів користувача для рушія Unity

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3; кількість сторінок записки 77

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі досліджено проблему тривалого ітераційного циклу під час створення розширень редактора Unity та проаналізовано наявні програмні рішення у цій предметній області. На основі цих даних було сформовано функціональні та нефункціональні вимоги до проєкту, спроєктовано архітектуру інструментального засобу для візуального конструювання користувацьких інтерфейсів на основі шаблону MVVM та фреймворку UI Toolkit і виконано програмну реалізацію. За результатами проведеного тестування підтверджено, що розроблене програмне забезпечення функціонує стабільно та успішно усуває визначені проблеми.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі обґрунтовано актуальність застосування no-code підходу для оптимізації процесів розробки внутрішнього ігрового інструментарію. У першому розділі проведено детальний аналіз предметної області, розглянуто існуючі аналоги та чітко визначено функціональні і нефункціональні вимоги до майбутнього продукту. У другому розділі спроєктовано модульну архітектуру системи, розроблено механізми збереження даних з використанням поліморфної серіалізації, а також детально спроєктовано підсистеми глобального перехоплення, віртуалізації та графів логіки. У третьому розділі виконано практичну програмну реалізацію, а також проведено статичний аналіз коду та тестування методом «чорної скриньки».

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки вирішує проблему складності та тривалості розробки ігрових інструментів. Перевагою є використання найновіших технологій рушія Unity. Окремо варто відзначити гнучку модульну архітектуру, що забезпечує високу відмовостійкість та легку масштабованість системи.

5. Негативні сторони роботи У роботі було повністю спроєктовано модулі віртуалізації та графу логіки, проте їх не було програмно реалізовано.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення виконано відповідно до теми кваліфікаційної роботи та подано у вигляді діаграм, рисунків і таблиць. Пояснювальна записка оформлена згідно вимог чинних стандартів.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки. Матеріал пояснювальної записки викладено з належною повнотою, послідовно та аргументовано, що дозволяє чітко та в повній мірі зрозуміти викладений матеріал у рамках тематики дипломної роботи. Графічні матеріали чітко відображають та пояснюють текст, висвітлений у роботі.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, повністю відповідає поставленому завданню та заслуговує на оцінку "відмінно".

РЕЦЕНЗЕНТ Мартинюк Валерій Володимирович,
д.т.н., проф., проф. кафедри АКІТпоП, ХНУ

"27" травня 2026 р. _____
(підпис)

SemanticAI for Education

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

першого освітнього рівня «Бакалавр»

Студента: Юрчука Романа
Група: ІП3-22-1

Тема: *«Інструмент візуального створення редакторів користувача для рушія Unity»*

Спеціальність: 121 – Інженерія програмного забезпечення

Короткий зміст пояснювальної записки

Кваліфікаційна робота присвячена розробці інструментального засобу для візуального створення користувацьких редакторів у середовищі Unity. Актуальність теми обумовлена потребою в модернізації процесів розробки інструментарію, що дозволить знизити витрати на розробку та підвищити ефективність роботи геймдизайнерів і програмістів. Мета роботи полягає у створенні програмного засобу, який автоматизує генерацію інтерфейсів, усуває необхідність рекомпіляції коду та знижує поріг знань для початку роботи. Для досягнення цієї мети визначено ряд завдань, включаючи аналіз предметної області, розробку архітектури та тестування програмного продукту.

Відповідність отриманих результатів роботи поставленим завданням

Завдання, сформульовані у вступі, включають аналіз предметної області, вивчення сучасних стандартів, визначення вимог до програмного забезпечення, розробку архітектури та реалізацію програмного продукту. Висновки роботи підтверджують,

що результати в цілому відповідають поставленим завданням, оскільки розроблений інструмент відповідає вимогам, визначеним на етапі аналізу. Оцінка відповідності: **в цілому відповідають**.

Оцінка розділів

Розділ 1: Аналіз предметної області

У розділі представлено детальний аналіз предметної області, що охоплює розробку інструментів для оптимізації робочих процесів у середовищі Unity. Описано основні об'єкти маніпуляції, обмеження стандартного механізму редагування та три парадигми для побудови інтерфейсів. Проте, відсутність конкретних прикладів використання інструментів та чіткої структури ускладнює сприйняття інформації. Рекомендується додати приклади та структурувати текст для покращення сприйняття. Загалом, розділ містить важливу інформацію про сучасні тенденції у розробці ігор.

Розділ 2: Проєктування програмного забезпечення

У цьому розділі розглянуто архітектуру, декомпозицію, залежності, інтерфейси та модулі. Описано вибір архітектури MVVM, що обґрунтовано наявністю в UI Toolkit вбудованої підсистеми прив'язки даних. Проте, відсутність чітких порівнянь між архітектурами та недостатня деталізація декомпозиції ускладнюють розуміння. Рекомендується додати порівняння архітектур та детальнішу декомпозицію. Загалом, розділ містить логічний підсумок проєктування та чітке визначення задач.

Розділ 3: Програмна реалізація та тестування

Розділ містить детальний опис реалізації базових контрактів, але відсутні фрагменти коду, що ускладнює розуміння ключових алгоритмів. Описано вимоги до технічних засобів, але не зазначено структуру бази даних та реалізацію тригерів. Тестування охоплює функціональні аспекти, але відсутні конкретні приклади тестів. Рекомендується доповнити розділ фрагментами коду та візуалізацією результатів тестування. Загалом, розділ містить важливу інформацію про реалізацію, але потребує доопрацювання.

Позитивні сторони

Кваліфікаційна робота демонструє оригінальність у підході до розробки інструментального засобу, що відповідає сучасним тенденціям no-code та visual scripting. Якість рішень, представлених у роботі, свідчить про глибоке розуміння предметної області та потреби користувачів. Зручність для користувача підкреслюється акцентом на автоматизацію процесів, що знижує поріг входження для нових користувачів.

Недоліки

Серед недоліків можна відзначити відсутність конкретних прикладів використання інструментів у розділі аналізу, що ускладнює сприйняття інформації. Також, недостатня деталізація в описі декомпозиції та міжмодульних залежностей у розділі

проєктування може призвести до неповного розуміння архітектури системи. В розділі реалізації відсутні фрагменти коду, що ускладнює розуміння ключових алгоритмів.

Відгук в цілому

Кваліфікаційна робота є актуальною та практично значущою, оскільки відповідає потребам індустрії розробки програмного забезпечення. Зміст роботи відповідає темі та завданню, а новизна ідей та рішень підкреслює інноваційний підхід до розробки інструментів. Однак, є певні недоліки у структурі та деталізації, які потребують доопрацювання. Загалом, робота демонструє високий рівень самостійності та аналітичного мислення здобувача.

Оцінка кваліфікаційної роботи

Кваліфікаційна робота заслуговує оцінки **добре**. Вона виконана в повному обсязі з дотриманням основних вимог, проте має деякі недоліки, які потребують доопрацювання. Здобувач володіє матеріалом, грамотно викладає суть роботи, хоча може припускатися дрібних неточностей.

Рекомендації

Рекомендується доопрацювати роботу, зокрема, уточнити формулювання вимог, додати приклади використання інструментів, структурувати текст для покращення сприйняття, а також включити фрагменти коду у розділі реалізації. Це підвищить якість роботи та її відповідність вимогам.

OpenAI API-асистент
Session ID: d67a8241-1b11-430a-a15f-935c335a6318
Підписано автоматично, модель gpt-4o-mini
Дата: 15.05.2026

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів.
Назва кваліфікаційної роботи: «Інструмент візуального створення редакторів користувача для рушія Unity»
Автор: Юрчук Роман Володимирович
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Праворська Наталія Іванівна, кандидат педагогічних наук, доцент
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована.	
3	Виявлені запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Виявлені запозичення частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:
1) у тексті кваліфікаційної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках, у структурі змісту, назвах розділів/підрозділів, рамках форм, у назвах та URL-адресах публікацій переліку джерел посилання;
2) запозичення, виявлені у тексті роботи, є фрагментарними.
Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 2.0% з одного джерела. Загальна сумарна подібність у базі даних складає 3% за символами та 6% за лексемами. Крім того, за результатами додаткового аналізу коефіцієнт подібності 1 становить 2.7%, коефіцієнт подібності 2 – 0.88%. Не виявлено мікропробілів, зайвих білих знаків або маніпуляцій з інтервалами у кількості, що класифікувалося б як навмисне спотворення тексту. З урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 27.05.2026

Завідувач кафедри _____ Леонід БЕДРАТЮК

Гарант освітньої програми _____ Леонід БЕДРАТЮК

Керівник кваліфікаційної роботи _____ Наталія ПРАВОРСЬКА