Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук

Кафедра Системотехніки

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Розробка модульної архітектури ігрового застосунку з використанням архітектурного шаблону Scriptable Object Architecture
(тема)

Виконав:

Студент 6 курсу, групи ІТПм-24-2

Солодкий Д.В.
(прізвище, ініціали)

Спеціальність F3 Комп’ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Інформаційні технології проектування
(повна назва освітньої програми)

Керівник проф. каф. СТ Ситніков Д.Е.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри ________ Єрохін А. Л.
(підпис) (Прізвище, ініціали)

2025 р.

Я, як студент ХНУРЕ розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

__________ ___02.12.2025_Солодкий___________
(дата, підпис, прізвище студента)

Кваліфікаційна робота не містить відомостей заборонених до відкритого публікування.

Керівник кваліфікаційної роботи _______________________

Кваліфікаційна робота виконана у відповідності до стандартів, що діють в Україні.

Керівник кваліфікаційної роботи _______________________

Попередній захист проведено ____________

Керівник кваліфікаційної роботи _______________________

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук

Кафедра Системотехніки

Рівень вищої освіти другий (магістерський)

Спеціальність F3 Комп’ютерні науки

(код і повна назва)

Тип програми освітньо-професійна

(освітньо-професійна або освітньо-наукова)

Освітня програма інформаційні технології проектування

(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри____________

(підпис)

«____»__________ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові Солодкому Денису Віталійовичу

(прізвище, ім’я, по батькові)

1. Тема роботи Розробка модульної архітектури ігрового застосунку з використанням архітектурного шаблону Scriptable Object Architecture

Затверджена наказом університету від ______________________2025р. №__

2. Термін подання студентом роботи до екзаменаційної комісії _________2025 р.

3. Вихідні дані до роботи тип використовуваного архітектурного шаблону, вимоги щодо функціоналу демонстраційного застосунку, вимоги щодо розробки елементів застосунку

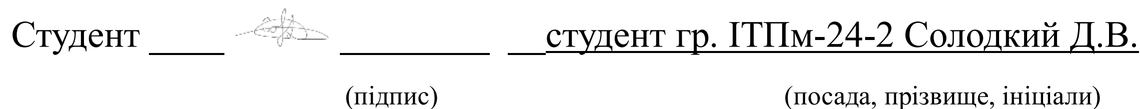
4. Перелік питань, що потрібно опрацювати в роботі розробити архітектуру ігрового застосунку за допомогою проведення аналізу обраної предметної області, формування вимог до демонстраційного ігрового застосунку; спроектувати архітектуру застосунку; реалізувати елементи системи

5. Перелік графічного матеріалу із зазначенням клеслеників, схем, плакатів, комп’ютерних ілюстрацій (слайдів) діаграма потоків даних (DFD), діаграма IDEF0 та її декомпозиція, діаграма класів, діаграма варіантів використання ігрового застосунку, діаграма послідовності дій

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів	Примітка
1	Аналіз предметної області	22.11.2025	Виконано
2	Дослідження існуючих аналогів	22.11.2025	Виконано
3	Розробка плану розробки архітектури ігрового застосунку	22.11.2025	Виконано
4	Розробка функціональних вимог	23.11.2025	Виконано
5	Розробка моделі потоків даних системи	24.11.2025	Виконано
6	Опис архітектури (структури) демонстраційного ігрового застосунку	24.11.2025	Виконано
7	Розробка ігрового меню	25.11.2025	Виконано
8	Розробка поведінкової моделі артефактів ігрового світу	27.11.2025	Виконано
9	Розробка поведінкової моделі некерованих та керованих персонажів	28.11.2025	Виконано
10	Оформлення пояснювальної записки до кваліфікаційної роботи	30.11.2025	Виконано
11	Захист кваліфікаційної роботи	23.12.2025	Виконано

Дата видачі завдання ________________2025 р.



Студент ____ ________ __студент гр. ІТПм-24-2 Солодкий Д.В.

(підпис) (посада, прізвище, ініціали)

Керівник роботи_____________ ___проф. каф. СТ Ситніков Д.Е.____

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Робота містить: 109 сторінок, 6 рисунків, 25 джерел, 2 додатки

СКРИПТОВІ ОБ’ЄКТИ (SCRIPTABLE OBJECTS), АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ІГРОВИЙ РУШІЙ UNITY, ІГРОВІ ЗАСТОСУНКИ, МОДУЛЬНІ СИСТЕМИ, ТОП-DOWN RPG

Предметом дослідження є архітектурні підходи, методи структуризації даних та принципи побудови модульних програмних систем, що застосовуються під час розробки інтерактивних ігрових застосунків у середовищі Unity.

Об’єктом дослідження є процес проєктування та реалізації архітектури ігрового застосунку, зокрема принципи організації його модулів, способи обміну даними між компонентами та підходи до побудови взаємодіючих підсистем у рамках сучасного ігрового рушія.

Метою роботи є розробка модульної архітектури ігрового застосунку з використанням архітектурного шаблону Scriptable Object Architecture (SOA) та демонстрація її ефективності на прикладі створення 2D гри жанру Top-Down RPG.

У роботі проведено аналіз предметної області розробки архітектур ігрових застосунків, виконано огляд сучасних архітектурних шаблонів, визначено їх переваги та доцільність застосування у контексті модульної побудови ігрових систем. На основі проведеного дослідження сформовано вимоги до архітектури, розроблено її концептуальну модель та функціональну структуру.

Сфера застосування результатів – створення модульних, масштабованих та легко підтримуваних ігрових застосунків, а також розробка архітектурних рішень для ігрової індустрії.

ABSTRACT

The work contains: 109 pages, 6 figures, 25 sources, 2 appendencies

SCRIPTABLE OBJECTS, SOFTWARE ARCHITECTURE, UNITY GAME ENGINE, GAME APPLICATIONS, MODULAR SYSTEMS, TOP-DOWN RPG

The subject of the study is architectural approaches, data structuring methods and principles of building modular software systems used in the development of interactive game applications in the Unity environment.

The object of the study is the process of designing and implementing the architecture of a game application, in particular the principles of organizing its modules, methods of data exchange between components and approaches to building interacting subsystems within the framework of a modern game engine.

The purpose of the work is to develop a modular architecture of a game application using the Scriptable Object Architecture (SOA) architectural template and demonstrate its effectiveness on the example of creating a 2D game of the Top-Down RPG genre.

The work analyzes the subject area of developing game application architectures, reviews modern architectural templates, determines their advantages and feasibility of application in the context of modular construction of game systems. Based on the research, requirements for the architecture were formed, its conceptual model and functional structure were developed.

The scope of application of the results is the creation of modular, scalable and easily maintainable game applications, as well as the development of architectural solutions for the game industry.

ЗМІСТ

ВСТУП... 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДОСЛІДЖЕННЯ.................................... 10
1.1 Аналіз предметної області розробки архітектури ігрових застосунків...10
1.2 Дослідження існуючих підходів, методів, засобів, технологій, інструментарію для вирішення подібних задач.. 13
1.3 Вибір середовища розробки (ігрового рушія) для реалізації ігрового застосунку.. 21
1.4 Постановка задачі дослідження.. 23
2 РОЗРОБКА АРХІТЕКТУРИ ІГРОВОГО ЗАСТОСУНКУ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ……………………………………………………………………….. 26
2.1 Розробка функціональних вимог………………………………………… 26
2.2 Розробка моделі потоків даних системи... 32
2.3 Розробка діаграми варіантів використання ігрового застосунку..... 37
2.4 Розробка діаграми класів ігрового застосунку................................. 40
2.5 Розробка послідовності дій ігрового застосунку……………………....47
3 ОПИС ПРИЙНЯТИХ ПРОЄКТНИХ РІШЕНЬ ПІД ЧАС РОЗРОБКИ МОДУЛЬНОЇ АРХІТЕКТУРИ ІГРОВОГО ЗАСТОСУНКУ.............................53
3.1 Опис архітектури (структури) ігрового застосунку....................................53
3.2 Розробка ігрового меню...57
3.3 Розробка поведінкової моделі керованого персонажа................................61
3.4 Розробка поведінкової моделі некерованих персонажів……………………..64
3.5 Розробка бойових механік ігрового застосунку...68
3.6 Розробка артефактів ігрового світу...73
ВИСНОВКИ...78
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ...80
ДОДАТОК А.. 82
ДОДАТОК Б... 91

ВСТУП

У сучасному світі цифрові технології дедалі глибше інтегруються у повсякденне життя людини, виконуючи не лише утилітарні, але й соціальні, культурні та психологічні функції. Однією з найбільш динамічних та впливових сфер цифрової індустрії є індустрія відеоігор. Ігрові застосунки давно перестали бути виключно засобом розваги – вони стали формою творчості, способом самовираження, засобом комунікації, тренажерами, навчальними платформами та інструментом зниження психологічної напруги. В умовах тривалої війни, постійного стресу та невизначеності ігрові технології відіграють важливу роль у підтримці емоційної рівноваги, надаючи користувачам можливість тимчасово відволіктися, зосередитися на інтерактивній діяльності та отримати позитивний досвід.

Ігрова індустрія характеризується стрімким розвитком, збільшенням кількості платформ, жанрів та технічних рішень. Сучасні ігри є складними програмними системами, які включають десятки взаємодіючих підсистем: систему персонажів, модель ігрового світу, інтерфейс користувача, систему предметів, штучний інтелект, систему збережень тощо. Розробка таких застосунків вимагає не лише творчого бачення, а й глибокого розуміння принципів архітектурного проєктування, що дозволяють забезпечити масштабованість, гнучкість, продуктивність і легкість підтримки проєкту.

Однією з ключових проблем сучасної геймдев-індустрії є надмірна зв’язаність компонентів у традиційних підходах до розробки. Монолітні структури, щільна взаємодія між класами, дублювання логіки та низький рівень повторного використання коду ускладнюють розвиток проєктів, роблять їх менш придатними до модифікації та розширення. Особливо гостро ця проблема проявляється у великих RPG-проєктах, де кількість взаємозалежних елементів може сягати сотень. Тому питання побудови модульної архітектури ігрових застосунків є одним із найактуальніших у сучасній практиці розробки.

Серед підходів, що спрямовані на пом’якшення цих проблем, особливе місце займає Scriptable Object Architecture (SOA) – архітектурний шаблон, що використовує ScriptableObject у Unity для створення незалежних модулів та передачі даних через реактивні структури. Цей підхід дозволяє відокремити дані від логіки, зменшити кількість прямий залежностей, спростити тестування, а також забезпечити високу гнучкість та повторне використання компонентів.

Unity, який широко застосовується в індустрії, надає інструменти для реалізації подібних архітектурних рішень, зокрема ScriptableObjects, подієві канали, систему відокремлених контейнерів даних та можливість побудови редакторських інструментів. Завдяки цьому разом із класичними підходами (MVC, MVVM, ECS, EDA, Clean Architecture) SOA формує сучасні засоби організації внутрішньої структури ігор. Застосування цього підходу є особливо доцільним для розробки ігор у жанрі Top-Down RPG, де існує велика кількість взаємодій між персонажами, предметами, підсистемами та оточенням.

У даній кваліфікаційній роботі розглядається проблема побудови модульної архітектури ігрового застосунку з використанням Scriptable Object Architecture. Метою роботи є створення архітектурного рішення, що дозволяє структурувати логіку гри, забезпечити слабку зв’язаність між компонентами, підвищити гнучкість системи та спростити масштабування проєкту. Для демонстрації принципів та ефективності розробленої архітектури реалізовано демонстраційний ігровий застосунок жанру Top-Down RPG, у якому архітектурні рішення відіграють ключову роль.

У роботі розглянуто архітектурні підходи, проведено аналіз предметної області, побудовано функціональні моделі (IDEF0), діаграми потоків даних (DFD), UML-діаграми, а також виконано практичну реалізацію архітектури засобами Unity. Запропоноване рішення орієнтоване на подальше використання у більш складних ігрових проєктах та інформаційних системах інтерактивного типу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДОСЛІДЖЕННЯ

1.1 Аналіз предметної області розробки архітектури ігрових застосунків

Ігрова індустрія є однією з найдинамічніших сфер інформаційних технологій, яка поєднує в собі програмну інженерію, системний аналіз, графіку, штучний інтелект, моделювання та взаємодію людини з комп'ютером. Ігрові застосунки – це складні програмні системи, які реалізують інтерактивну взаємодію між користувачем і віртуальним середовищем у реальному часі. З технічної точки зору, гра – це інформаційна система реального часу, що обробляє потоки подій, станів та ресурсів.

Процес створення ігрового застосунку охоплює не лише художнє проєктування чи геймдизайн, а й розробку архітектури програмного продукту, яка визначає взаємозв'язки між усіма складовими гри – від системи вводу до логіки поведінки персонажів, системи збереження даних, UI, фізичного рушія, штучного інтелекту тощо. Саме архітектурне рішення визначає, наскільки ефективною, стабільною та розширюваною буде гра.

Ігровий застосунок можна розглядати як різновид інтерактивної інформаційної системи, де джерелом даних є дії користувача, а результати обчислень – зміна стану об'єктів у віртуальному середовищі. Основні компоненти такої системи включають:

– інформаційну базу (дані про персонажів, предмети, рівні, параметри тощо);

– механізм обробки подій (система вводу, фізика, колізії, логіка AI);

– підсистему візуалізації (рендеринг 2D або 3D сцени);

– інтерфейс користувача;

– систему збереження даних (локально або через сервер).

У контексті ігрового застосунку жанру «Top-Down RPG», який використовується як демонстраційний приклад у цій роботі, інформаційна

система керує станом персонажа, його інвентарем, прогресом квестів, статистикою боїв тощо. Така система повинна бути гнучкою й модульною, щоб забезпечити можливість швидкої зміни механік, додавання нових сутностей або систем (наприклад, бойових здібностей, діалогів, інвентарю), не порушуючи при цьому цілісності проєкту. Це вимагає структурованого підходу до проєктування, що й зумовлює потребу у модульній архітектурі.

Модульна архітектура – це підхід до проєктування програмних систем, який передбачає поділ програми на незалежні частини (модулі), що мають чітко визначені функції та інтерфейси. Для ігрового застосунку це дозволяє:

– спростити підтримку та розширення гри;

– реалізувати повторне використання компонентів;

– підвищити стабільність та масштабованість коду;

– ізолювати логіку від візуальної частини проєкту.

У середовищі Unity модульна архітектура досягається за допомогою гнучких компонентних систем та шаблонів проєктування, серед яких особливу роль відіграє Scriptable Object Architecture.

Scriptable Object Architecture – це архітектурний підхід, заснований на використанні об’єктів типу ScriptableObject у Unity як незалежних контейнерів даних. Основна ідея – розділити дані, логіку та події у незалежні модулі, які легко налаштовуються й повторно використовуються у різних проєктах. Основні переваги:

– можливість створювати глобальні змінні та події без жорстких зв’язків між компонентами;

– зниження залежностей і покращення інкапсуляції;

– спрощення налагодження та повторного використання ресурсів;

– зручна інтеграція у редактор Unity для швидкого налаштування поведінки об’єктів.

Scriptable Object Architecture забезпечує:

– гнучкість під час розширення системи (додавання нових механік без модифікації існуючих скриптів);

– декларативність – усі параметри можна редагувати в інспекторі Unity без зміни коду;

– реактивність – зміни у ScriptableObject автоматично оновлюють пов'язані компоненти;

– низьку зв'язаність – логіка об'єктів не містить жорстких посилань на інші елементи гри.

Scriptable Object Architecture активно використовується для реалізації систем інвентарю, бойових механік, керування станом гри, обробки подій і навіть побудови Event-Driven логіки.

У межах цієї магістерської роботи Scriptable Object Architecture використовується для створення модульної архітектури Top-Down RPG-застосунку, де окремі підсистеми (рух, бойова система, інвентар, UI, керування станами гри) функціонують як незалежні модулі, що взаємодіють через події й глобальні змінні, реалізовані на базі ScriptableObject.

Хоча більшість 2D-RPG є локальними застосунками, сучасні ігрові платформи передбачають можливість розширення до розподілених систем, наприклад, для мультиплеєра чи хмарного збереження даних. У цьому контексті важливо забезпечити:

– синхронізацію станів між клієнтами та сервером;

– балансування навантаження при великій кількості користувачів;

– використання баз даних для зберігання інформації про користувачів та їхній прогрес;

– побудову безпечних каналів обміну даними.

Розробка модульної архітектури ігрового застосунку є складовою галузі програмної інженерії та інформаційних систем, орієнтованих на інтерактивні середовища. Вона поєднує принципи:

– архітектурного моделювання;

– об'єктно-орієнтованого програмування;

– системного аналізу та декомпозиції;

– візуального проєктування та оптимізації.

Розробка Top-Down RPG із використанням Scriptable Object Architecture дозволяє створити стійку, розширювану і добре структуровану систему, що може служити основою для масштабних ігор або навчальних симуляторів.

Розробка ефективної архітектури є критичною задачею в сучасному геймдеві. Від її якості залежить не лише продуктивність гри, а й можливість масштабування, підтримки, інтеграції з зовнішніми системами (аналітика, оновлення, мережеві сервіси тощо).

Тому тема цієї роботи – побудова модульної архітектури ігрового застосунку з використанням Scriptable Object Architecture – є актуальною в контексті розвитку архітектурного проєктування ігрових систем. Вона спрямована на оптимізацію процесу розробки, підвищення стабільності проєкту та покращення повторного використання компонентів у майбутніх розробках.

2.2 Дослідження існуючих підходів, методів, засобів, технологій, інструментарію для вирішення подібних задач

Розробка сучасних ігрових застосунків є складним процесом, що поєднує творчий та інженерний підходи. Ігри – це не лише засіб розваги, а й складні програмні системи, які потребують чіткої структури, стабільності, повторного використання компонентів та легкості розширення [4], [5]. Саме тому під час створення ігрового програмного забезпечення ключову роль відіграє архітектура проєкту, тобто спосіб організації взаємодії між різними елементами системи – логікою, даними, об’єктами, користувацьким інтерфейсом та механізмами збереження стану [1], [3].

Архітектурне проєктування виступає фундаментом будь-якої розробки, оскільки саме від нього залежить не лише якість і продуктивність гри, але й подальша можливість її масштабування, підтримки та розвитку [1], [2]. Погано спроєктована архітектура може призвести до надмірної складності коду, зниження продуктивності, появи взаємних залежностей між компонентами та

труднощів у додаванні нових функцій. Навпаки, добре структурована архітектура дозволяє розробникам швидко вносити зміни, тестувати окремі частини системи та забезпечувати стійкість проєкту на всіх етапах життєвого циклу гри [2], [3].

Особливе значення архітектурні рішення мають у розробці ігрових рушіїв, таких як Unity, Unreal Engine, Godot тощо, які надають гнучкі можливості для організації ігрової логіки, обробки подій, відображення графіки та керування ресурсами [7], [8]. Оскільки сучасні ігри часто поєднують десятки взаємопов'язаних підсистем (бойова система, інвентар, штучний інтелект, користувацький інтерфейс, збереження прогресу), правильний вибір архітектурного шаблону стає визначальним для їх ефективного функціонування [4], [5].

З цієї причини у процесі створення ігор розробники активно застосовують різні архітектурні шаблони (architectural patterns) – перевірені часом підходи до організації коду, що допомагають забезпечити зрозумілу структуру, логічну ієрархію та взаємодію між компонентами програми [4]. Кожен із цих шаблонів має власні переваги, недоліки та сфери доцільного використання залежно від складності проєкту, вимог до продуктивності, гнучкості чи масштабування [5]. Аналіз існуючих архітектурних шаблонів дає можливість виявити сильні сторони кожного підходу, оцінити їхню ефективність у контексті розробки ігрових систем та визначити, який із них найбільш відповідає цілям створюваного застосунку [1], [2].

Аналіз існуючих архітектурних шаблонів дає можливість виявити сильні сторони кожного підходу, оцінити їхню ефективність у контексті розробки ігрових систем та визначити, який із них найбільш відповідає цілям створюваного застосунку.

Для ігор, побудованих на рушії Unity, особливу популярність отримали шаблони, орієнтовані на модульність, слабку зв'язаність компонентів і можливість конфігурації поведінки через дані, серед яких найпоширенішими є Model– View– Controller (MVC), Model– View– ViewModel (MVVM) та

Entity– Component– System (ECS) [4], [5]. Саме ці архітектурні підходи формують основу сучасного бачення структури ігрових систем, забезпечуючи гнучкість і стабільність під час розробки [6].

Одним із найвідоміших і найстаріших архітектурних підходів у програмній інженерії є шаблон Model-View-Controller (MVC), що виник ще у 1970-х роках у межах Smalltalk, але згодом набув великої популярності в розробці настільних, веб- та мобільних додатків [3]. Його основна ідея полягає у розділенні структури програмного забезпечення на три взаємопов’язані частини: Model (модель), View (представлення) і Controller (контролер). Такий поділ дозволяє розмежувати відповідальність компонентів системи, зменшити зв’язаність коду та спростити подальший розвиток і підтримку проєкту [2], [3].

У контексті ігрової розробки модель (Model) описує структуру даних та логіку гри. Вона містить усі відомості про стан світу, об’єкти, характеристики персонажів, ресурси, очки досвіду, предмети тощо. Представлення (View) відповідає за відображення інформації користувачеві – це графічна частина гри: сцена, спрайти, анімації, інтерфейс користувача, ефекти. Контролер (Controller) виступає посередником між моделлю та представленням, обробляючи події, що виникають під час взаємодії гравця із системою: натискання клавіш, кліки, рух миші, зіткнення тощо [4].

Застосування MVC у геймдеві має низку переваг. По-перше, воно забезпечує чітке розділення логіки і візуалізації, що спрощує командну роботу – програмісти можуть зосереджуватись на моделі та контролері, тоді як художники або UI-дизайнери працюють із View [6], [7]. По-друге, така структура сприяє підвищенню тестованості та гнучкості: зміни у відображенні не впливають на логіку гри, а оновлення моделі не потребує переробки графічної частини [2], [3].

Втім, при розробці складних ігор MVC має й обмеження. У динамічних системах, де об’єкти постійно змінюють стан, а взаємодій дуже багато, Controller може швидко перетворитися на перевантажений елемент, який складно підтримувати. Крім того, у традиційній реалізації MVC модель не

«знає» про представлення, тому для постійної синхронізації між ними потрібен додатковий механізм сповіщень або подій, що може ускладнювати розробку. Через це в ігровій індустрії MVC часто використовується локально – наприклад, для реалізації інтерфейсів користувача (меню, інвентарю, діалогових вікон), а не як глобальна архітектура всього проєкту.

Подальший розвиток ідей MVC привів до появи низки похідних шаблонів – насамперед Model-View-Presenter (MVP) та Model-View-ViewModel (MVVM). Обидва підходи прагнуть вирішити основну проблему класичного MVC – надмірну концентрацію логіки у контролері – шляхом введення проміжного шару, який бере на себе функції зв'язку між даними та представленням [11], [12].

У шаблоні MVP центральним елементом є Presenter – об'єкт, який отримує дані від моделі, обробляє їх та передає у вигляді, придатному для відображення у View. Presenter безпосередньо не взаємодіє з графічними елементами, а лише повідомляє View, що саме потрібно відобразити. Такий підхід дозволяє повністю ізолювати бізнес-логіку від інтерфейсу, що значно підвищує зручність тестування – Presenter можна перевіряти незалежно від UI.

У свою чергу, шаблон MVVM, який набув популярності у середовищах, орієнтованих на декларативний UI (наприклад, WPF, Xamarin, Unity UI Toolkit), вводить поняття ViewModel – представлення моделі, що містить дані, готові до безпосереднього зв'язування з інтерфейсом користувача (data binding) [11], [12]. На відміну від Presenter, ViewModel не керує View напряму, а лише надає властивості та команди, на які View може «підписатися». Це робить систему реактивною: зміни у даних автоматично відображаються у UI, і навпаки [11], [12].

В ігровій розробці шаблони MVP та MVVM найчастіше застосовуються для побудови складних інтерфейсів користувача: інвентарів, налаштувань, меню, вікон квестів або систем діалогів. У таких випадках UI має велику кількість взаємопов'язаних елементів, які потрібно синхронізувати з поточним

станом гри. Використання Presenter або ViewModel дозволяє мінімізувати кількість ручного коду та полегшує підтримку.

Перевагою цих підходів є низька зв’язаність між даними та інтерфейсом, що робить код гнучким та легким для розширення. Крім того, MVVM чудово поєднується з архітектурою Scriptable Object Architecture, де роль ViewModel можуть виконувати ScriptableObject-змінні та подієві канали. Таким чином, у сучасних ігрових застосунках ці шаблони часто виступають як допоміжний рівень, що інтегрує архітектуру даних (SOA) з візуальною частиною проєкту.

Одним із найвпливовіших підходів у сучасному геймдеві є Entity–Component– System (ECS) – архітектура, що виникла як еволюція класичної об’єктно-орієнтованої моделі, але з фокусом на дано-орієнтований дизайн (Data-Oriented Design) [4], [16]. На відміну від традиційного підходу, де поведінка прив’язана до об’єктів через наслідування, ECS пропонує розділити гру на три ключові сутності: Entity (сутність), Component (компонент) та System (система) [4], [16].

Entity – це абстрактна одиниця, що представляє об’єкт у грі (персонаж, снаряд, ворог, предмет тощо), але не має власних даних чи логіки. Усі характеристики зберігаються у компонентах – структурах даних, які описують певний аспект поведінки або стану (позиція, швидкість, здоров’я, колізія, графіка). Логіка ж реалізується у системах – незалежних процесорах, які обробляють усі сутності, що мають певну комбінацію компонентів. Наприклад, система руху опрацьовує всі об’єкти з компонентами Position і Velocity, а система бою – ті, що мають Health і Attack.

Основною перевагою ECS є висока продуктивність і масштабованість. Завдяки чіткому відокремленню даних від логіки та компактному зберіганню компонентів у пам’яті, ECS дозволяє максимально використовувати кеш процесора, оптимізує обробку великої кількості об’єктів і забезпечує можливість багатопотокового виконання систем. Це робить підхід особливо ефективним для ігор із великою кількістю активних елементів – наприклад, стратегій, симуляторів або масових RPG.

Водночас ECS має й певні складнощі. Його впровадження вимагає ретельного планування структури даних та зміни мислення розробника – від об’єктно-орієнтованої логіки до дано-орієнтованої. Для невеликих 2D-проєктів це може бути надмірно складним рішенням. Проте навіть у таких іграх можна використовувати гібридний підхід, де ECS застосовується лише до підсистем, що вимагають високої продуктивності (наприклад, обробка ворогів або снарядів), тоді як решта логіки реалізована за допомогою більш декларативних архітектур – таких як Scriptable Object Architecture.

Наступним шаблоном є подієво-орієнтоана архітектура, або Event-Driven Architecture. Її сутність полягає в тому, що взаємодія між окремими компонентами програми відбувається не через прямі виклики методів чи обмін повідомленнями «зверху вниз», а шляхом генерації та обробки подій. Подія у цьому контексті – це повідомлення, яке інформує систему про певну зміну стану: натискання клавіші, завершення анімації, отримання шкоди персонажем, відкриття дверей тощо.

На відміну від традиційних монолітних структур, у яких модулі безпосередньо взаємодіють між собою, подієво-орієнтований підхід базується на принципі публікації-підписки (publish– subscribe) [10]. Один компонент, який є джерелом події, «публікує» повідомлення, не знаючи, хто саме на нього відреагує, тоді як інші модулі підписуються на ці події та реагують на них відповідно до власної логіки. Така схема суттєво знижує зв’язаність системи, оскільки компоненти стають незалежними і можуть змінюватися або замінюватися без потреби змінювати інші частини програми [4], [10].

В ігровій розробці подієво-орієнтована архітектура має особливе значення, адже гра – це система, що постійно реагує на події, створювані гравцем або середовищем. Кожна дія користувача – натискання клавіші, клік миші, вибух снаряда, зіткнення об’єктів – породжує подію, яку мають обробити відповідні системи. Завдяки EDA можливо організувати складні сценарії взаємодії без хаотичних залежностей між об’єктами сцени. Наприклад, коли персонаж підбирає предмет, подія ItemCollected може

сповістити підсистему інвентарю, користувацький інтерфейс і систему звуку – без прямих викликів між ними.

Використання подій спрощує процес масштабування і тестування: можна легко додати нову реакцію на існуючу подію (наприклад, візуальний ефект чи запис у журнал) без втручання у вже реалізовану логіку. Крім того, події можуть передаватися між різними рівнями архітектури – від геймплейної логіки до UI – утворюючи єдину систему реактивної взаємодії.

Однак подієво-орієнтована архітектура вимагає суворої організації. Якщо кількість подій надмірна, а їхні імена або структури не стандартизовані, система може стати важкою для налагодження. Тому важливо визначати чіткі правила обробки подій, їхній життєвий цикл і типізацію.

У середовищі Unity EDA зазвичай реалізується через подієві канали (Event Channels) на базі ScriptableObject, що забезпечують незалежну взаємодію між об’єктами без прямих посилань. Такий підхід ідеально поєднується з архітектурним шаблоном Scriptable Object Architecture, який буде детальніше розглянуто далі, і виступає одним із основних способів побудови модульних та масштабованих ігрових систем. Подієво-орієнтована взаємодія дає можливість створювати не просто набір класів, а динамічну екосистему модулів, які реагують на зміни середовища, що особливо важливо для RPG-ігор, де геймплей постійно породжує нові стани та події.

Ще одним фундаментальним підходом, що набув популярності у професійній розробці програмного забезпечення, є концепція «Чистої архітектури» (Clean Architecture), запропонована Робертом Мартіном [1]. Вона базується на принципі інверсії залежностей та чіткому розділенні рівнів системи за ступенем їхньої абстракції. Метою цього підходу є створення програмної системи, де доменна логіка (правила гри) ізольована від деталей реалізації (інтерфейсу, рушія, бази даних, файлів тощо) [1].

Основна ідея полягає у поділі системи на концентричні шари, де найвнутрішній рівень – це бізнес-логіка або доменна модель, яка визначає головні сутності та правила їхньої взаємодії. Зовнішні шари – це

інфраструктура, користувацький інтерфейс, бази даних і зовнішні сервіси. Всі залежності мають бути спрямовані всередину, тобто зовнішні елементи можуть використовувати внутрішні, але не навпаки. Це забезпечує стабільність логіки навіть при зміні технологій чи рушія.

У контексті ігрової розробки застосування Clean Architecture дозволяє відокремити «мозок» гри – її механіку, правила та стан – від усіх допоміжних систем, таких як графіка, фізика, ввід чи інтерфейс користувача. Наприклад, бойова система RPG може бути реалізована як незалежний доменний модуль, що не має жодних посилань на класи Unity. У цьому випадку логіка обчислення шкоди, перевірки умов перемоги чи взаємодії предметів існує окремо від того, як ці події відображаються на екрані. Такий підхід суттєво підвищує тестованість ігрової логіки, оскільки її можна перевіряти за допомогою звичайних модульних тестів без запуску самої сцени.

Перевагою «чистої» архітектури є також гнучкість і довговічність проєкту. Якщо в майбутньому потрібно перейти з одного рушія на інший або змінити технологію візуалізації, доменна логіка залишається незмінною – достатньо реалізувати новий зовнішній шар. Це робить архітектуру стійкою до технологічних змін і полегшує підтримку протягом багатьох років.

Водночас цей підхід має і певні складнощі. Для його успішного впровадження необхідно витримувати сувору дисципліну проєктування, дотримуватись принципів SOLID і ретельно визначати межі між шарами. Надмірна кількість абстракцій може ускладнити розробку невеликих проєктів, де глибока шарова структура не є виправданою. Тому Clean Architecture найкраще підходить для середніх і великих проєктів, які передбачають тривалу підтримку, оновлення та можливість розширення.

Для Unity Clean Architecture часто реалізується у поєднанні з Scriptable Object Architecture або подієво-орієнтованими системами, що забезпечують передачу даних між шарами без порушення принципу інверсії залежностей. У такій комбінації внутрішня логіка гри описується звичайними C#-класами, а зовнішній шар Unity виступає лише як засіб візуалізації, що реагує на події або

зміни даних. Таким чином, «чиста» архітектура стає архітектурним каркасом, який гарантує структурну цілісність і стабільність ігрового застосунку в довгостроковій перспективі.

1.3 Вибір середовища розробки (ігрового рушія) для реалізації ігрового застосунку

У процесі створення ігрового застосунку одним із ключових етапів є вибір середовища розробки, або ж ігрового рушія (game engine), який визначає технологічну базу, архітектурні можливості, продуктивність та гнучкість майбутнього проєкту. Від правильності цього вибору залежить не лише якість кінцевого продукту, а й ефективність самого процесу розробки, можливості тестування, підтримки та масштабування системи.

Ігровий рушій – це комплексне програмне середовище, яке об’єднує набір інструментів, бібліотек та технологій, призначених для створення інтерактивних графічних застосунків і відеоігор. Він забезпечує роботу таких основних підсистем, як:

– графічний рушій (rendering engine) – відповідає за відображення 2D та 3D-графіки, анімацію, освітлення й тіні;

– фізичний рушій (physics engine) – реалізує симуляцію руху, колізій, гравітації та інших фізичних взаємодій;

– аудіосистема – обробляє музику, звукові ефекти, просторове звучання;

– система введення/виведення – забезпечує обробку дій користувача (клавіатура, миша, геймпад);

– скриптингова підсистема – надає інструменти для програмування логіки гри;

– редактор сцен і активів – дозволяє створювати ігрові об’єкти, середовища, рівні;

– система побудови проєкту (build system) – здійснює експорт гри під різні платформи.

Завдяки комплексності цих інструментів ігровий рушій виконує роль інтегрованого середовища розробки ігрового процесу, значно спрощуючи реалізацію навіть складних сценаріїв, фізичних симуляцій і візуальних ефектів.

Unity – це один із найпопулярніших у світі ігрових рушіїв, розроблений компанією Unity Technologies і вперше представлений у 2005 році. Його головна особливість полягає у компонентно-орієнтованій архітектурі та потужному візуальному редакторі, які дозволяють створювати інтерактивні ігри та симуляції практично будь-якої складності. Unity підтримує розробку для більш ніж 25 платформ, серед яких Windows, macOS, Linux, Android, iOS, WebGL, PlayStation, Xbox, Nintendo Switch та інші, що робить його універсальним середовищем для кросплатформних проєктів.

Рушій надає розробнику повний набір інструментів для створення як 2D, так і 3D-ігор, зокрема:

– редактор сцен і префабів;

– інтегровану підтримку фізики (Box2D, PhysX);

– систему частинок і візуальних ефектів;

– засоби анімації (Animator, Animation Graph);

– модуль освітлення, камери та постобробки;

– інтеграцію зі скриптовою мовою C#, яка є основною мовою розробки логіки проєктів;

– можливість підключення зовнішніх бібліотек, пакетів і власних модулів через Unity Package Manager;

– інструменти для профілювання, дебагу та оптимізації продуктивності.

Unity активно підтримує концепцію швидкої ітеративної розробки, тобто зміни у проєкті можна миттєво тестувати без повного перекомпілювання, що значно пришвидшує цикл створення і тестування ігор.

Вибір саме Unity як середовища реалізації проєкту обумовлений низкою технічних, архітектурних і практичних переваг, що роблять його оптимальним

рішенням для створення модульної архітектури ігрового застосунку та реалізації демонстраційної гри жанру Top-Down RPG.

Unity базується на гнучкій системі GameObject– Component, що дозволяє будувати складні ієрархії об'єктів і додавати до них функціональність через незалежні компоненти. Це повністю узгоджується з ідеєю модульності архітектури, яка лежить в основі теми роботи. Також Unity нативно підтримує використання ScriptableObject, що дозволяє реалізувати низько зв'язану модульну архітектуру, де дані, події й конфігурації існують окремо від ігрових об'єктів. Це забезпечує простоту масштабування, повторне використання модулів і легке налагодження системи. Використання C# як основної мови програмування забезпечує об'єктно-орієнтований підхід до проєктування, доступ до широкої екосистеми бібліотек, високу читабельність коду та можливість легкої інтеграції з зовнішніми інструментами. Крім цього, Вбудований редактор Unity дозволяє не лише розміщувати об'єкти у сцені, але й взаємодіяти з архітектурними елементами, налаштовуючи логіку гри на рівні інспектора без зміни коду. Це спрощує реалізацію підходів, де поведінка визначається даними.

Рушій дозволяє експортувати готовий продукт на різні операційні системи та пристрої без необхідності суттєвої переробки коду. Це відкриває можливість подальшого розвитку проєкту в напрямі мобільних або веб-версій.

Таким чином, вибір Unity як середовища розробки є логічним і технічно обґрунтованим рішенням. Його можливості повністю відповідають цілям роботи – реалізації модульної архітектури ігрового застосунку з використанням Scriptable Object Architecture, яка забезпечує низьку зв'язаність компонентів, високу гнучкість та простоту подальшої підтримки.

1.4 Постановка задачі дослідження

Враховуючи результати аналізу предметної області, дослідження існуючих архітектурних підходів та вибір середовища розробки, можна

сформулювати постановку задачі даного дослідження. Метою роботи є розробка модульної архітектури ігрового застосунку на базі архітектурного шаблону Scriptable Object Architecture (SOA), що забезпечить гнучкість, масштабованість та зниження зв'язаності компонентів у межах рушія Unity.

Сучасні ігрові системи складаються з великої кількості підсистем – управління персонажем, бойових механік, інвентарю, користувацького інтерфейсу, системи збереження, діалогів тощо. Традиційна архітектура, побудована на прямих зв'язках між об'єктами (reference-based architecture), призводить до сильної зв'язаності коду, що ускладнює його тестування, повторне використання та розширення.

Для усунення цих проблем у дослідженні ставиться завдання розробити модульну архітектуру, в якій окремі компоненти гри функціонують автономно, обмінюючись інформацією через подієві канали, змінні даних і конфігураційні об'єкти. Основна ідея полягає у використанні ScriptableObject як центрального елемента управління даними, подіями та взаємодією між системами гри.

Загальна постановка задачі полягає у створенні такої архітектурної моделі, яка:

– забезпечить мінімальну зв'язаність між окремими модулями;

– дозволить гнучко масштабувати систему за рахунок додавання нових функціональних блоків без зміни існуючих;

– підтримуватиме подієво-орієнтовану взаємодію між компонентами;

– забезпечить зручність налагодження і повторного використання архітектурних рішень у майбутніх проєктах.

У ході роботи передбачається реалізація низки етапів, кожен з яких спрямований на поступове досягнення поставленої мети:

– дослідження теоретичних засад Scriptable Object Architecture (SOA): визначення основних принципів та моделей її застосування у Unity. Аналіз існуючих практик розділення даних і логіки, зокрема використання ScriptableObject для управління подіями, станами, параметрами ігрового світу;

– проєктування модульної архітектури ігрового застосунку: Формування структурної схеми, що відображає взаємодію основних модулів системи – управління персонажем, інтерфейсу, інвентарю, бойової системи, логіки рівнів. Визначення каналів комунікації між модулями на основі ScriptableObject-змінних та подієвих каналів;

– вибір інструментів і середовища розробки: для реалізації архітектури використовується ігровий рушій Unity, що підтримує компонентно-орієнтований підхід і має вбудований механізм роботи зі ScriptableObject. Основна мова програмування – C#, оскільки вона забезпечує підтримку об’єктно-орієнтованих і подієвих парадигм програмування.

– реалізація демонстраційного ігрового застосунку: на основі розробленої архітектури створюється приклад 2D-гри жанру Top-Down RPG, який виступає експериментальною платформою для перевірки працездатності архітектурного рішення. Гра має містити базові модулі – пересування, бойову систему, інтерфейс користувача, інвентар та механізм збереження прогресу;

– перевірка ефективності розробленої архітектури: на завершальному етапі здійснюється аналіз переваг розробленої системи у порівнянні з традиційними архітектурами, оцінюються параметри гнучкості, масштабованості, модульності, а також простота внесення змін і повторного використання компонентів.

У результаті виконання дослідження очікується створення універсальної модульної архітектури, придатної для подальшого використання при розробці різних типів ігрових застосунків на Unity. Вона повинна:

– забезпечувати низький рівень зв’язаності між модулями;

– підтримувати динамічну конфігурацію поведінки гри через дані;

– бути масштабованою та повторно використовуваною.

2 РОЗРОБКА АРХІТЕКТУРИ ІГРОВОГО ЗАСТОСУНКУ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ

2.1 Розробка функціональних вимог

Розробка модульної архітектури ігрового застосунку потребує створення чіткої методології, яка описує структуру системи, взаємозв’язки між її компонентами, а також засоби реалізації кожного елемента. Під інформаційною технологією в контексті цієї роботи розуміється сукупність методів, архітектурних рішень, програмних засобів і процесів взаємодії, що забезпечують ефективне функціонування модульної структури гри.

Основу технології складає Scriptable Object Architecture (SOA) – архітектурний шаблон, орієнтований на розділення даних, логіки та взаємодій між системами через подієво-орієнтовану модель. Впровадження цієї архітектури дозволяє мінімізувати прямі залежності між об’єктами та забезпечити масштабованість гри за рахунок використання змінних ScriptableObject, подієвих каналів (Event Channels) та модульних конфігурацій.

Метою цього етапу є чітке визначення функціональних обов’язків, зв’язків та взаємодій між компонентами системи, що забезпечують ефективне функціонування ігрового застосунку жанру Top-Down RPG, реалізованого у середовищі Unity з використанням архітектурного шаблону Scriptable Object Architecture (SOA).

Методологія IDEF0 використовується для моделювання структури складних систем та опису функціональних процесів у вигляді ієрархічних моделей, які відображають взаємозв’язки між вхідними даними, керуючими впливами, механізмами реалізації та результатами діяльності системи. Цей підхід дозволяє системно описати не лише програмну архітектуру, а й

функціональні ролі кожного компонента, що особливо актуально при побудові модульних ігрових систем, де важливо забезпечити слабку зв'язаність і чітке розмежування обов'язків між підсистемами.

У контексті розробки архітектури ігрового застосунку методологія IDEF0 дає змогу:

– формалізувати структуру функцій і взаємодій між ігровими підсистемами (керування персонажем, бойова система, інвентар, інтерфейс користувача, система збереження тощо);

– визначити потоки даних між компонентами, які реалізуються через подієві канали (Event Channels) і змінні ScriptableObject.

Методологія IDEF0 базується на представленні системи у вигляді функціональних блоків, кожен з яких відображає окрему функцію або процес. Між блоками встановлюються стрілки зв'язку, які позначають:

– вхідні потоки (Input) – дані, ресурси або події, що надходять у систему для обробки;

– вихідні потоки (Output) – результати функціонування системи або створювані нею дані;

– керуючі потоки (Control) – правила, обмеження або зовнішні впливи, що регулюють виконання функцій;

– механізми (Mechanism) – інструменти, засоби або ресурси, які забезпечують виконання функцій.

Використання IDEF0 у розробці ігрової архітектури дозволяє візуалізувати логічну структуру застосунку та продемонструвати, як різні підсистеми взаємодіють у межах єдиної модульної системи. Кожен блок на діаграмі репрезентує певний аспект роботи гри – від обробки дій користувача до оновлення станів і відображення результатів у графічному інтерфейсі.

На основі моделювання IDEF0 визначено функціональні вимоги до системи. Ці вимоги враховують основні функції гри, взаємодію з гравцем,

логіку геймплею та інші ключові аспекти розроблюваного ігрового застосунку. Розроблені діаграми зображено на рисунках 2.1-2.5 нижче.

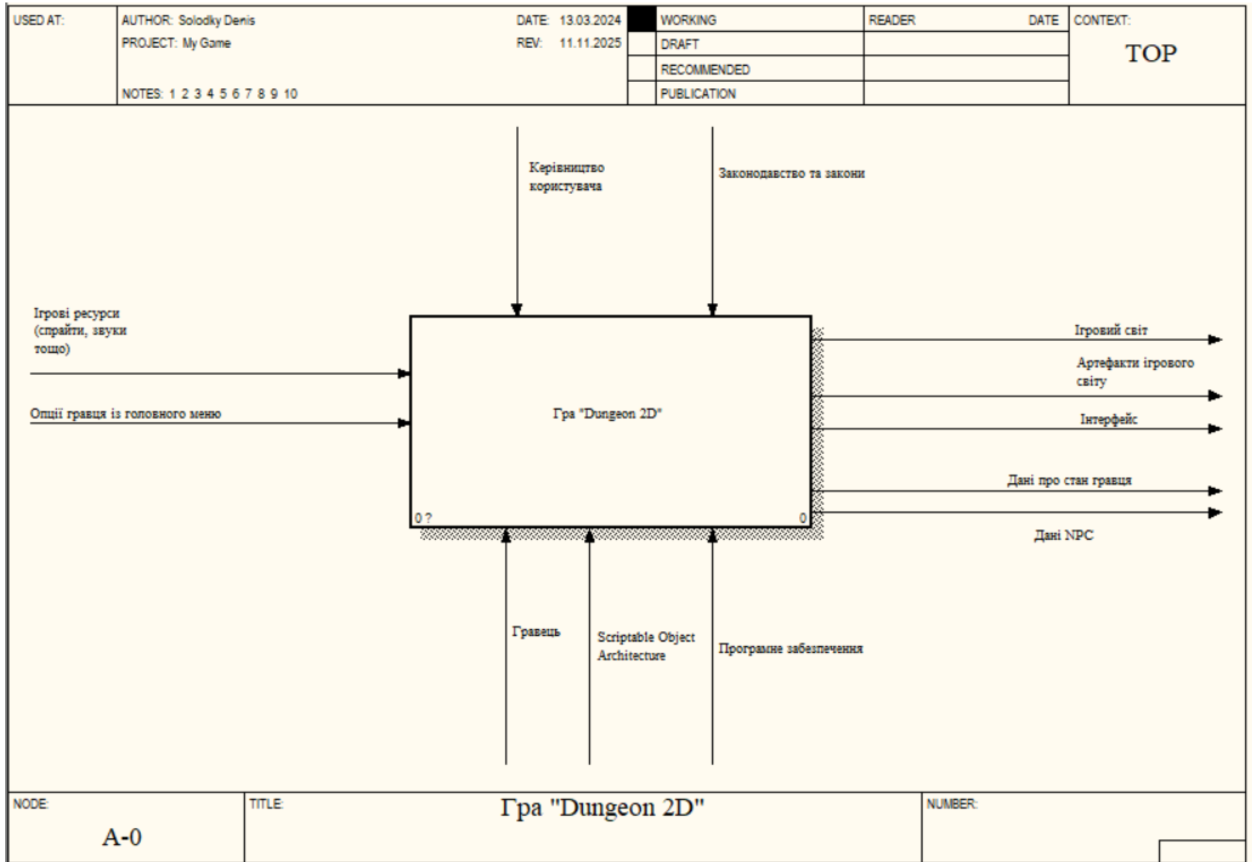


Рисунок 2.1 – основний блок діаграми IDEF0

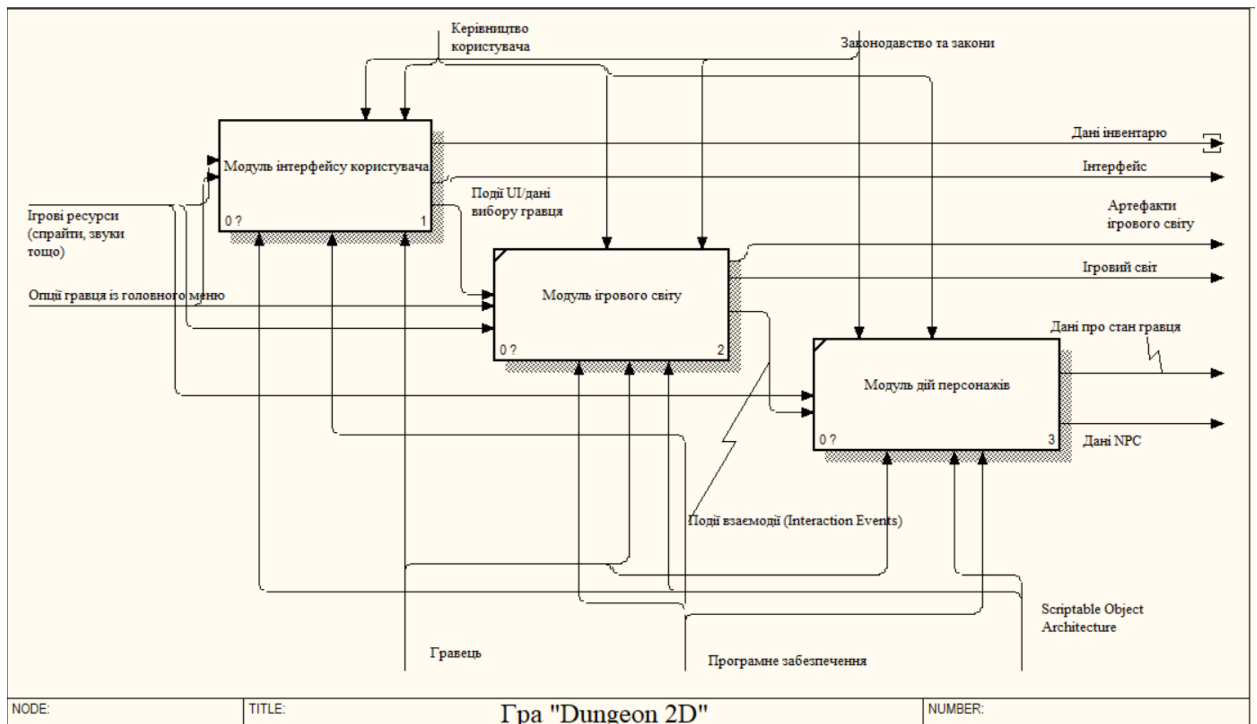


Рисунок 2.2 – декомпозиція основного блоку

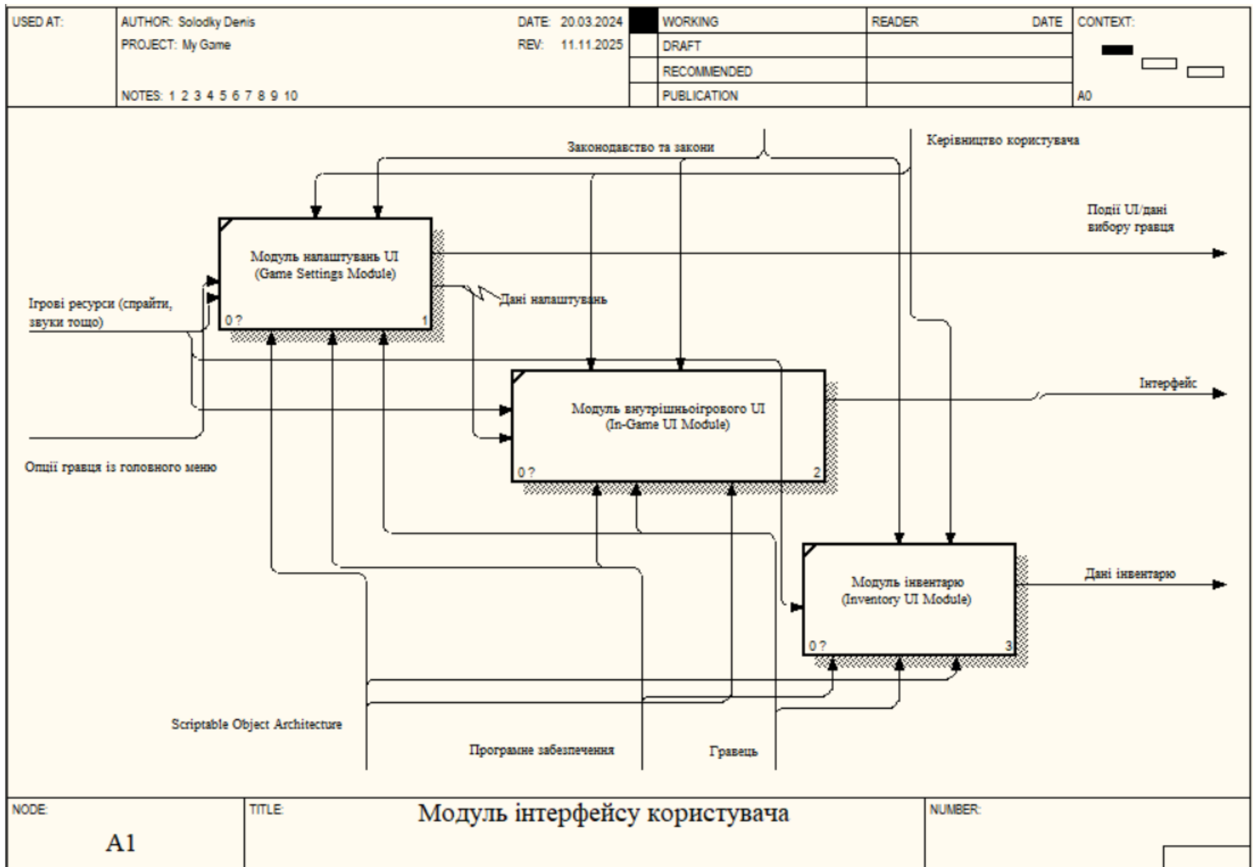


Рисунок 2.3 – декомпозиція блоку «Модуль інтерфейсу користувача»

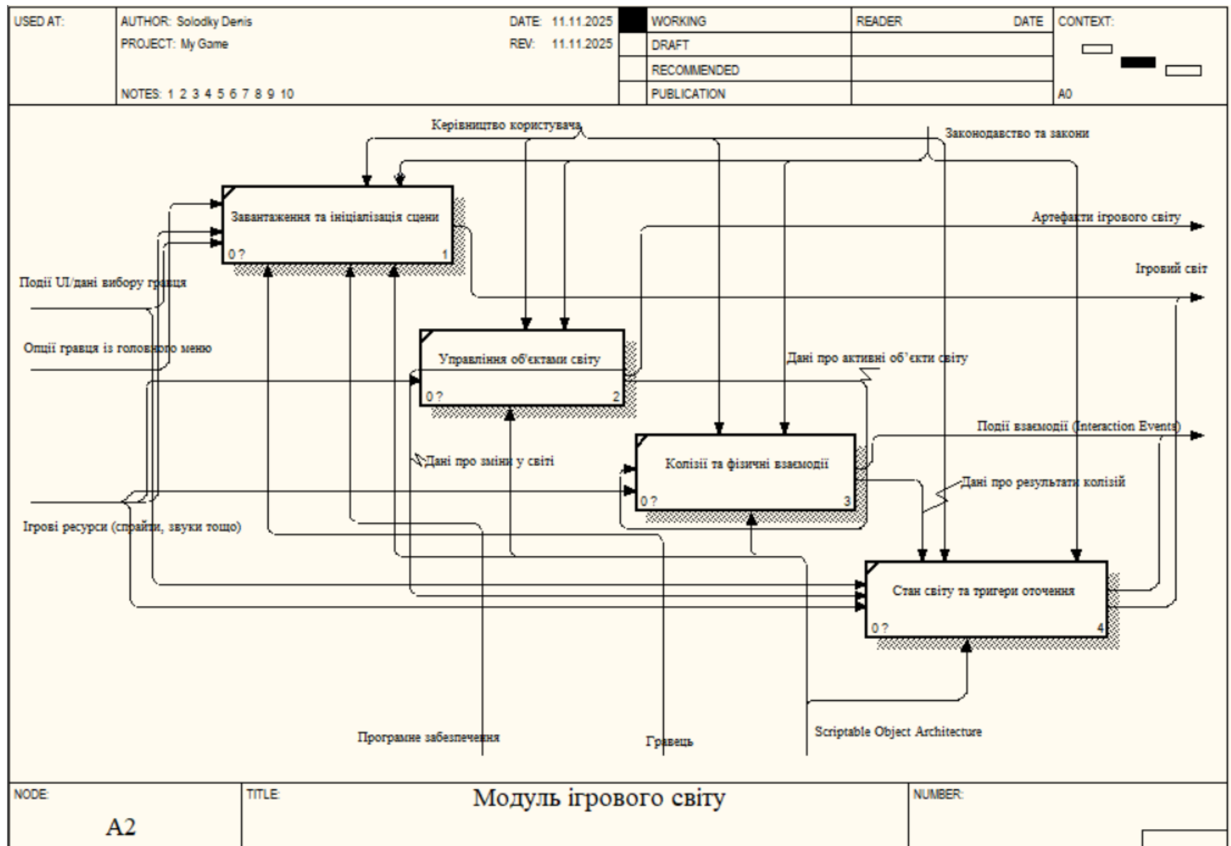


Рисунок 2.4 – декомпозиція блоку «Модуль ігрового світу»

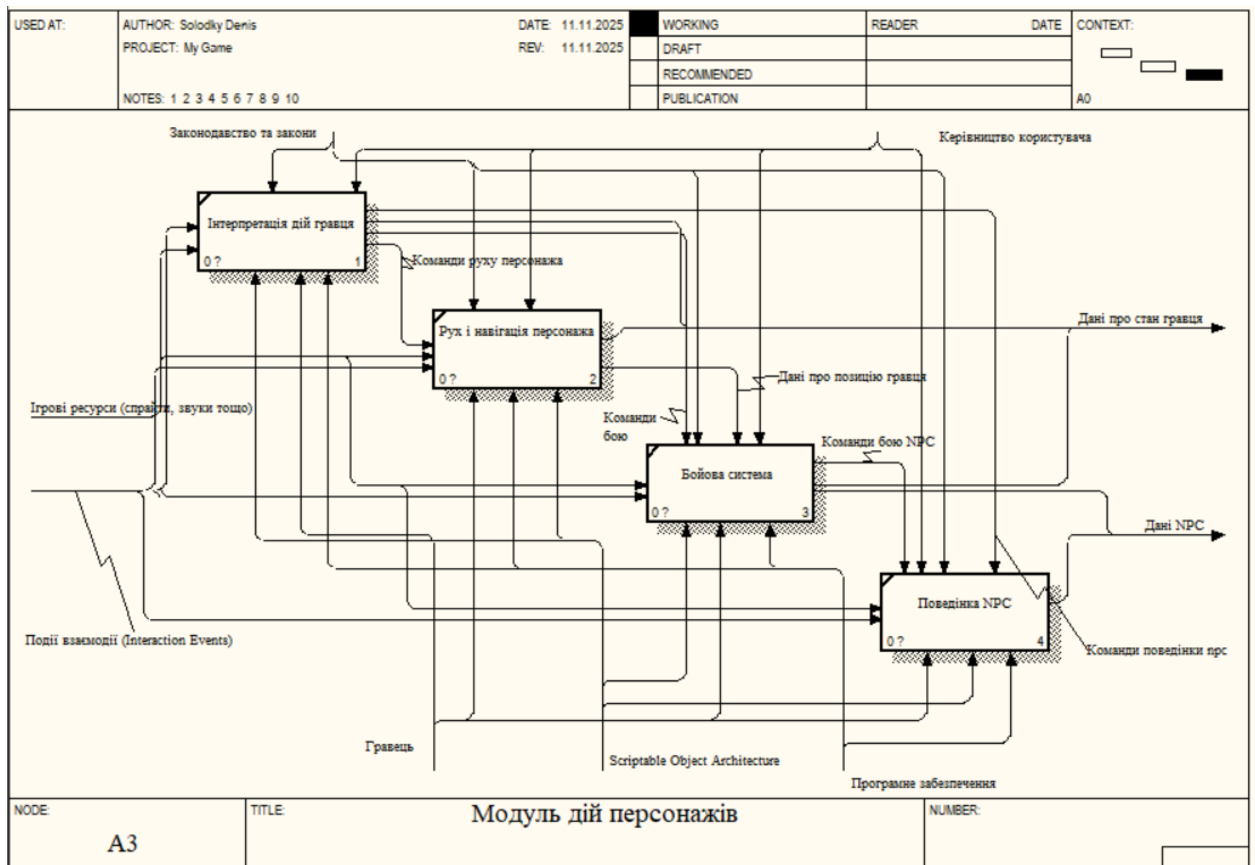


Рисунок 2.5 – декомпозиція блоку «Модуль дій персонажів»

На контекстній діаграмі (A-0) подано узагальнену модель функціонування ігрового застосунку жанру Top-Down RPG. Вхідними даними виступають дії користувача та ресурси гри (спрайти, сцени, скрипти, конфігураційні файли), що використовуються для формування ігрового світу.

Керування здійснюється згідно з правилами гри та інструкціями користувача. У ролі механізмів реалізації функцій виступає програмне забезпечення Unity та архітектурний шаблон Scriptable Object Architecture (SOA), який забезпечує модульну структуру та подієву взаємодію підсистем. Результатом роботи системи є сформований ігровий світ, артефакти, інтерфейс користувача та оновлені дані про ігрових і неігрових персонажів.

На діаграмі декомпозиції (A0) подано функціональну модель ігрового застосунку, побудованого з використанням архітектурного шаблону Scriptable Object Architecture (SOA).

Кожен блок діаграми відповідає окремому модулю архітектури: модулю інтерфейсу користувача, модулю ігрового світу та модулю дій персонажів. Взаємодія між модулями здійснюється через подієві канали (Event Channels) і змінні ScriptableObject, що забезпечує низьку зв'язаність компонентів і гнучкість системи.

Вхідними даними є дії користувача та ігрові ресурси (спрайти, сцени, звуки тощо), керування здійснюється на основі правил гри та інструкцій користувача.

Механізмами реалізації функцій виступають програмне забезпечення Unity та архітектурний шаблон SOA.

Вихідними даними є сформований ігровий світ, артефакти, інтерфейс, а також дані про стан ігрових та неігрових персонажів.

На діаграмі A1 представлено деталізацію роботи модуля інтерфейсу користувача, який реалізовано за допомогою архітектурного шаблону Scriptable Object Architecture (SOA).

До складу модуля входять три підмодулі: налаштування гри, внутрішньоігровий інтерфейс та інвентар гравця. Кожен з них реалізує окрему підфункцію UI-системи та взаємодіє з іншими елементами через подієві канали (Event Channels) та дані ScriptableObject (наприклад, SettingsSO, PlayerStateSO, InventorySO).

Такий підхід дозволяє досягти слабкої зв'язаності між модулями, спрощує оновлення окремих елементів інтерфейсу та забезпечує можливість повторного використання компонентів.

Вхідними даними для модуля є ресурси інтерфейсу (спрайти, аудіо, сцени) та опції користувача, а вихідними – сформований інтерфейс, події взаємодії та дані інвентарю.

На діаграмі A2 зображено декомпозицію модуля ігрового світу, який забезпечує завантаження, оновлення та реакцію середовища гри. Взаємодія між підмодулями реалізується за допомогою Scriptable Object

Architecture, де об'єкти гри, їхній стан та події представлені у вигляді ScriptableObject-змінних і подієвих каналів.

Підмодуль "Управління об'єктами світу" формує перелік активних сутностей і передає його до підмодуля "Колізії та фізичні взаємодії", який обробляє зіткнення й генерує події. Ці події надходять до підмодуля "Стан світу та тригери оточення", що оновлює глобальний стан середовища та ініціює подальші реакції гри (відкриття дверей, активацію пасток, зміну обстановки тощо).

Такий підхід дозволяє побудувати модульну архітектуру з низьким рівнем зв'язності, що спрощує розширення та тестування системи.

На діаграмі А3 показано структуру модуля дій персонажів, який реалізує взаємодію гравця та NPC з ігровим середовищем.

Блок «Інтерпретація дій гравця» відповідає за прийом і обробку подій від системи вводу та користувацького інтерфейсу, нормалізуючи їх до уніфікованого формату команд (Move, Attack, Interact). Ці команди передаються через Scriptable Object Event Channels до відповідних підмодулів – «Рух і навігація персонажа» та «Бойова система».

Модуль «Поведінка NPC» формує реакції неігрових персонажів, створюючи власні події дій (атака, ухилення, втеча) та надсилаючи їх до бойової системи. Таким чином, архітектура забезпечує спільний механізм взаємодії для всіх агентів гри – як керованих гравцем, так і автоматичних.

Події обробляються асинхронно, без прямої залежності між компонентами, що забезпечує низьку зв'язаність і модульність системи, відповідно до принципів Scriptable Object Architecture.

2.2 Розробка моделі потоків даних системи

Розробка архітектури сучасних ігрових застосунків передбачає не лише визначення структури програмних модулів чи вибір шаблонів проєктування, але й детальне моделювання потоків даних, які циркулюють у межах системи

під час роботи гри. У відеоіграх дані є основою усіх внутрішніх процесів – від ініціалізації сцени та завантаження рівня до обробки взаємодій персонажів, роботи інвентаря, NPC-логіки та збереження прогресу. Тому побудова діаграми потоків даних (DFD) є важливим етапом у формуванні цілісного розуміння взаємодії компонентів і підтвердження коректності розробленої архітектури.

DFD (Data Flow Diagram) – це графічний метод моделювання, який дозволяє візуально представити рух інформації між процесами, зовнішніми сутностями та сховищами даних у системі. На відміну від IDEF0, який описує функції та їх призначення, DFD концентрується саме на тому, які дані генеруються, обробляються та передаються між логічними компонентами. Такий підхід дозволяє чітко відобразити інформаційні залежності, оцінити коректність обміну даними між модулями, виявити дублювання або надмірні зв’язки, а також підтвердити відповідність архітектури принципам структурності, модульності та слабкої зв’язаності.

У контексті даної кваліфікаційної роботи DFD-діаграма відіграє роль інструменту, що демонструє логічну модель інформаційних потоків у демонстраційному ігровому застосунку, побудованому за принципами Scriptable Object Architecture. Оскільки SOA передбачає існування незалежних модулів, що взаємодіють між собою через дані та події, DFD дозволяє наочно показати, які саме дані циркулюють між підсистемами гри: інтерфейсом, ігровим світом, системами дій персонажів, базами збережень, файловими ресурсами та елементами оточення.

Основні елементи DFD включають:

– процеси – логічні дії чи операції, які виконують обробку даних (напр., завантаження гри, обробка дій гравця, оновлення NPC);

– потоки даних – стрілки, що показують напрям руху інформації між процесами, сховищами та зовнішніми сутностями;

– сховища даних (Data Stores) – внутрішні джерела або приймачі даних, такі як файли збережень, база NPC, структура рівня;

– зовнішні сутності – об’єкти поза системою, що ініціюють або отримують інформацію (наприклад, гравець як користувач системи).

Розроблена діаграма дозволяє представити логіку взаємодії між компонентами ігрового застосунку на різних етапах – від запуску гри та завантаження рівня до управління персонажами, NPC та збереженням прогресу. DFD використовується як інструмент перевірки коректності архітектури, оскільки дозволяє визначити, чи узгоджені структурні рішення з реальними потоками даних, які виникають під час роботи гри.

На рисунку 2.6 зображено розроблену DFD діаграму

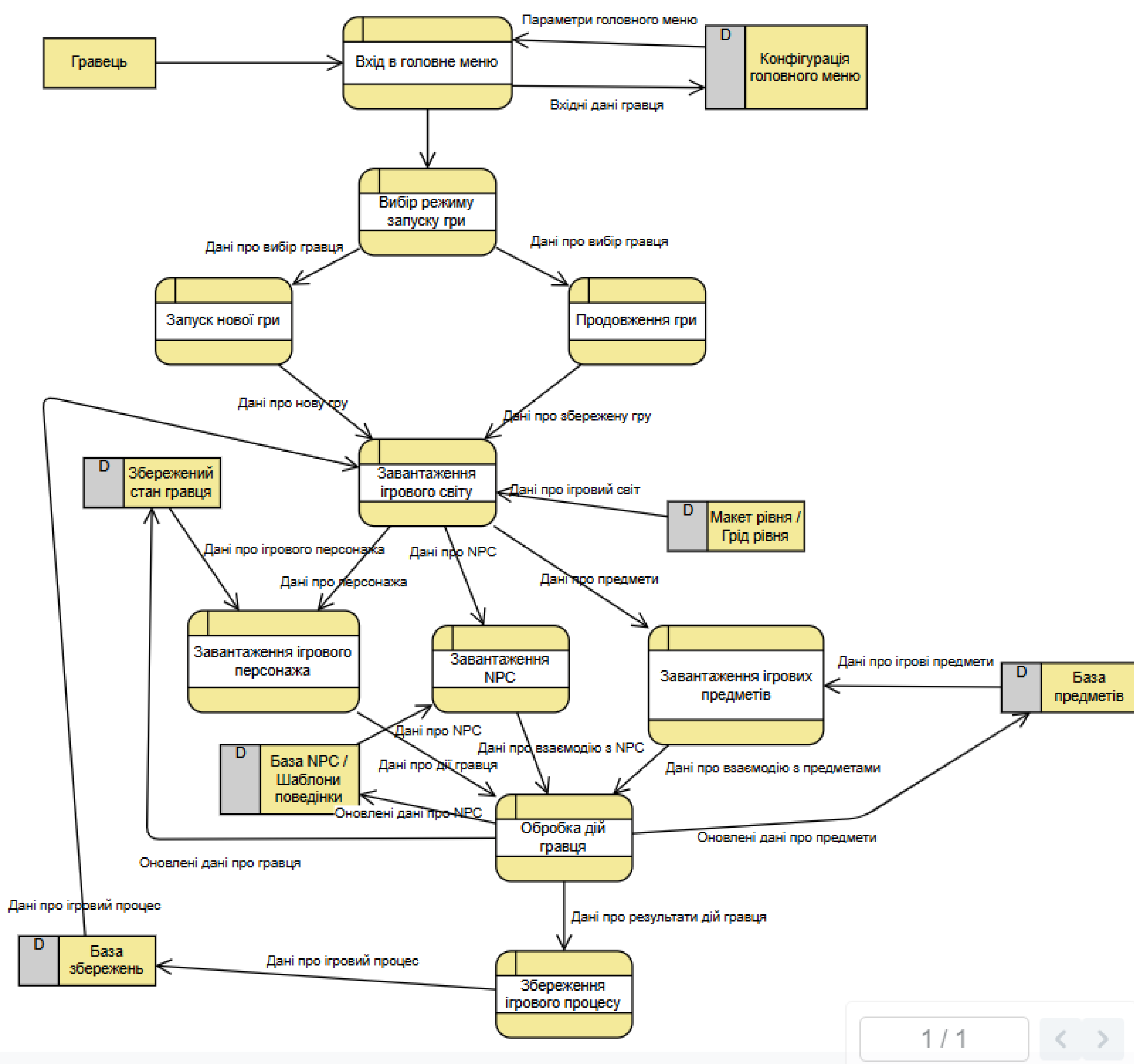


Рисунок 2.6 – діаграма DFD

Діаграма потоку даних (DFD), подана на рисунку, описує логічну структуру руху інформації в ігровому застосунку типу Top-Down RPG, який використовується для демонстрації розробленої архітектури на основі Scriptable Object Architecture. Діаграма відображає послідовність обробки користувацьких подій, завантаження ігрових ресурсів, ініціалізації компонентів і збереження прогресу гравця.

Першим кроком взаємодії гравця з системою є процес «Вхід в головне меню». Цей процес отримує:

– дані гравця (наприклад, налаштування, попередні параметри вибору);

– параметри конфігурації меню зі сховища Конфігурація головного меню.

Результатом обробки є відображення меню та передача «Вхідних даних гравця» у наступний процес – Вибір режиму запуску гри.

Далі йде вибір режиму запуску гри. На цьому етапі гравець обирає спосіб продовження гри – запуск нової гри, або продовження збереженої.

Процес формує вихідні дані – дані про вибір гравця, які спрямовуються в обидва можливі процеси – «Запуск нової гри» або «Продовження гри».

Наступними процесами є «Запуск нової гри» та «Продовження гри». Процес запуск нової гри формує початковий стан:

– створює за замовчуванням дані персонажа;

– встановлює початкові параметри світу;

– генерує базові значення інвентарю.

Також він передає «Дані про нову гру» до процесу «Завантаження ігрового світу».

Процес «продовження гри» отримує дані зі сховища «Збережений стан гравця», а саме:

– характеристики героя;

– стан NPC;

– прогрес квестів;

– стан рівня.

Далі він передає дані про збережену гру до процесу «Завантаження ігрового світу».

Процес завантаження ігрового світу є одним з ключових процесів усієї системи. На вхід він отримує:

– дані про нову або збережену гру;

– структуру рівня зі сховища «Макет рівня/Грід рівня»;

– стан збережених елементів світу.

Сам процес виконує ініціалізацію сцени, завантаженння компонентів оточення, передачу відповідних даних до наступних процесів. Результатом є синхронізований образ ігрового світу, необхідний для подальшого геймплею.

Наступним процесом є завантаження ігрового персонажа. Він отримує дані про гравця зі сховища. Сам процес формує структуру даних персонажа, включаючи характеристики, позицію у світі. Далі він передає дані у обробку дій гравця та повертає оновленні дані до системи збереження.

Наступний процес – завантаження NPC. Він отримує дані зі сховища та створює набір неігрових персонажів на рівні, ініціалізуючи поведінкові параметри, стани, позиції на мапі. Далі процес передає згенеровані дані в обробку дій гравця та готові моделі NPC у взаємодії з предметами та світом.

Процес завантаження ігрових предметів отримує дані зі сховища та формує предмети на рівні, розташовує їх відповідно до макету рівня, передає дані у обробку дій гравця. Предмети можуть бути як статичними, так і динамічними.

Наступний процес – обробка дій гравця. Цей процес відповідає за всю геймплейну взаємодію під час гри. На вхід він отримує дані про персонажа, дані про NPC, дані про предмети, команди гравця, події взаємодії. Функціями процесу є:

– обробка переміщень;

– взаємодія з предметами;

– бойові дії;

– відправлення подій у підсистему NPC;

– оновлення стану всіх активних компонентів.

Процес збереження ігрового процесу отримує дані від обробки дій гравця. Ці дані включають в себе змінений стан персонажа, прогрес рівнів, оновлені дані NPC, стан предметів та світу.

Передає дані до сховища «База збережень».

2.3 Розробка діаграми варіантів використання ігрового застосунку

Діаграма використання (Use Case Diagram) є одним з ключових інструментів UML, який застосовується для моделювання функціональної поведінки програмної системи на високому рівні абстракції. Основною метою такої діаграми є відображення взаємодії зовнішніх користувачів або акторів із системою, а також визначення основних сценаріїв, які система повинна підтримувати під час роботи.

На відміну від структурних або поведінкових діаграм, діаграма використання не деталізує внутрішню реалізацію функцій, структуру класів чи алгоритми. Вона концентрується саме на зовнішніх можливостях системи та тому, що може робити користувач, а не як це реалізовано програмно. Такий підхід дозволяє створити зрозумілу та доступну модель системи для різних категорій зацікавлених осіб – розробників, дизайнерів, тестувальників, аналітиків, а також кінцевих користувачів.

У контексті ігрового застосунку діаграма використання забезпечує:

– візуалізацію ігрового процесу: дозволяє представити ключові ігрові дії, доступні гравцю – пересування, атаки, взаємодію з об’єктами, перехід між рівнями тощо;

– опис поведінки NPC: дає можливість показати, які функції виконують неігрові персонажі (патрулювання, переслідування гравця, атака);

– визначення системних можливостей: збереження прогресу, завантаження гри, обробка подій, застосування логіки та конфігурацій;

– чітке розмежування відповідальності між суб’єктами: гравець, NPC та система мають власні набори сценаріїв.

Завдяки цьому, діаграма використання дозволяє структурувати вимоги до гри, визначити основні функціональні можливості кожного компонента та створити основу для подальшого проектування архітектури, діаграм послідовності та діаграм класів. Вона є невід’ємним етапом формування цілісної моделі ігрового застосунку, забезпечує цілісність дизайну та полегшує розуміння взаємодій у системі.

На рисунку 2.7 зображено розроблену use-case діаграму демонстраційного ігрового застосунку.

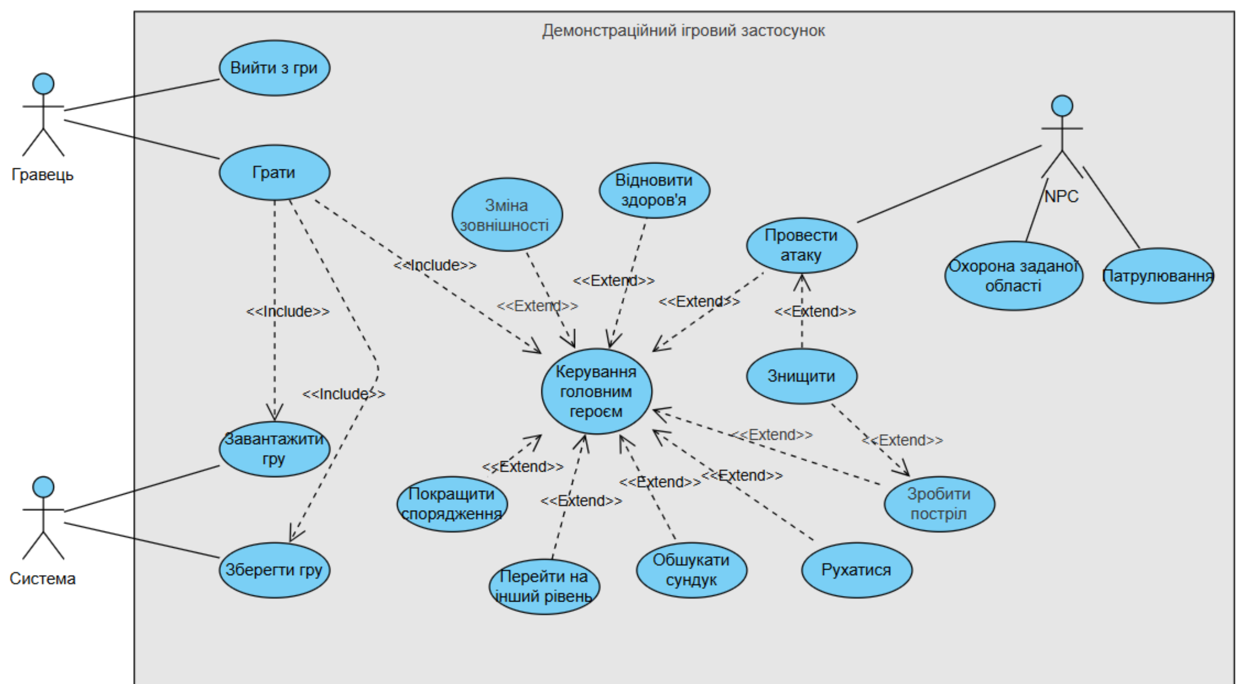


Рисунок 2.7 – use-case діаграма демонстраційного ігрового застосунку

На use-case діаграмі зображено основних акторів системи та ключові функціональні можливості, які надає розроблений ігровий застосунок. Діаграма відображає взаємодію між користувачем, системою та неігровими персонажами, що дозволяє чітко окреслити зовнішню поведінку програми на високому рівні абстракції.

У системі визначено три актори:

– гравець – основний користувач системи, який безпосередньо керує головним героєм, взаємодіє з предметами та впливає на перебіг ігрового процесу.

– система – внутрішній сервіс гри, відповідальний за збереження та завантаження прогресу.

– NPC (неігровий персонаж) – автономні сутності, керовані ігровою логікою. Вони можуть патрулювати територію або реагувати на появу гравця в певному радіусі.

Гравець має доступ до найбільшої кількості сценаріїв, пов’язаних з управлінням персонажем та взаємодією з ігровим світом:

а) грати – головний сценарій, що представляє увесь ігровий процес. До нього включені допоміжні сценарії, такі як завантажити гру та зберегти гру;

б) керування головним героєм – центральний use-case, що об’єднує всі взаємодії гравця з персонажем. Він розширюється низкою додаткових сценаріїв:

1) рухатися (<<extend>>) – переміщення персонажа по локації;

2) здійснити постріл (<<extend>>) – атака на відстані;

3) провести атаку (<<extend>>) – ближній бій;

4) знищити противника (<<extend>>) – результат успішної атаки;

5) відновити здоров’я (<<extend>>) – взаємодія з лікувальним фонтаном;

6) зміна зовнішності (<<extend>>) – вибір іншого скіна персонажа;

7) обшукати сундук (<<extend>>) – взаємодія з об’єктами для отримання нагороди;

8) покращити спорядження (<<extend>>) – модернізація зброї;

9) перейти на інший рівень (<<extend>>) – використання порталу.

Таким чином, сценарій «Керування головним героєм» є композиційним та описує усю поведінку персонажа в межах ігрового процесу.

Системний актор має два ключові сценарії:

– зберегти гру – використовується при виході або під час переходу між сценами;

– звантажити гру – включений до основного сценарію «Грати» і дозволяє відновити стан персонажа, інвентаря та прогресу.

Ці сценарії демонструють взаємодію гравця зі сховищем стану гри, але без прямої участі NPC.

NPC має власні сценарії поведінки, які не залежать від дій гравця:

– охорона заданої області – противник стоїть на місці й активується, коли гравець наближається;

– патрулювання – пересування між певними точками маршруту.

Обидва сценарії є автономними, проте їхня реалізація впливає на сценарії гравця (наприклад, бій або уникнення ворога).

2.4 Розробка діаграми класів

Діаграма класів (Class Diagram) є одним із ключових структурних елементів мови моделювання UML (Unified Modeling Language). Вона використовується для опису статичної структури програмної системи через відображення класів, їхніх властивостей, методів та зв’язків між ними. На відміну від діаграм поведінки (наприклад, діаграми станів або діаграми послідовності), діаграма класів демонструє архітектуру системи та її складові, що робить її невід’ємним елементом під час проєктування й документування програмних застосунків.

Діаграма класів дозволяє:

– відобразити структуру об’єктно-орієнтованої системи;

– показати зв’язки між класами;

– визначити їхні обов’язки, атрибути та методи;

– описати ієрархію наслідування;

– проаналізувати залежності між компонентами;

– закласти основу для кодування та подальшої підтримки проєкту.

У контексті розробки ігор Unity діаграма класів особливо корисна для документування архітектури ігрової логіки, взаємодії об’єктів сцени, а також використання патернів, зокрема Scriptable Object Architecture.

Основні елементи діагрми класів:

– класи, кожен клас на діаграмі відображається у вигляді прямокутника, що поділений на три секції, а саме назву, атрибути та операції або методи. Рівень доступу вказується як + (public), - (private) або #(protected);

– зв’язки, які використовуються для відображення залежностей між класами;

– множинність, яка скільки об’єктів одного класу може бути пов’язано з об’єктом іншого класу. В UML вона позначається числами біля кінців ліній зв’язку. На рисунках 2.7-2.8 зображено розроблену діаграму класів.

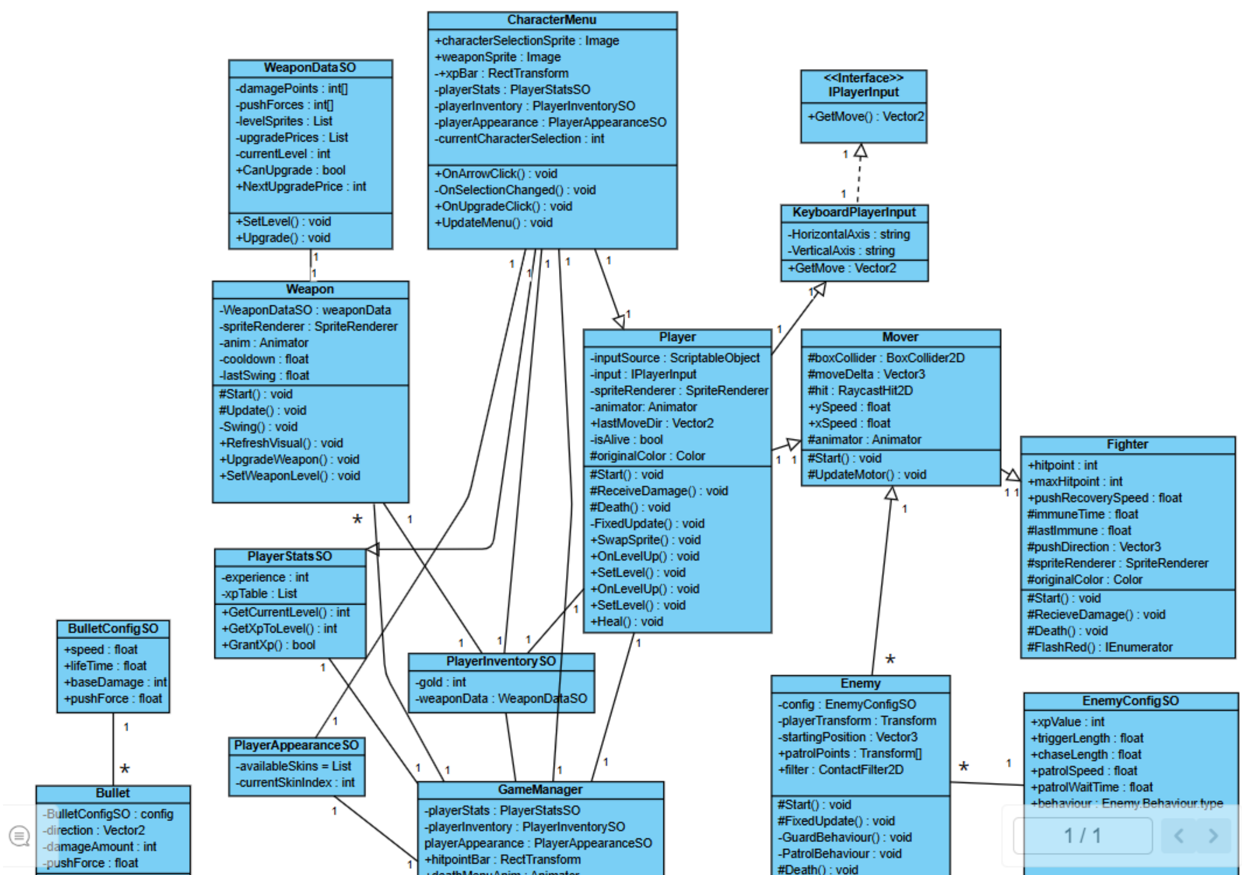


Рисунок 2.8 – діаграма класів

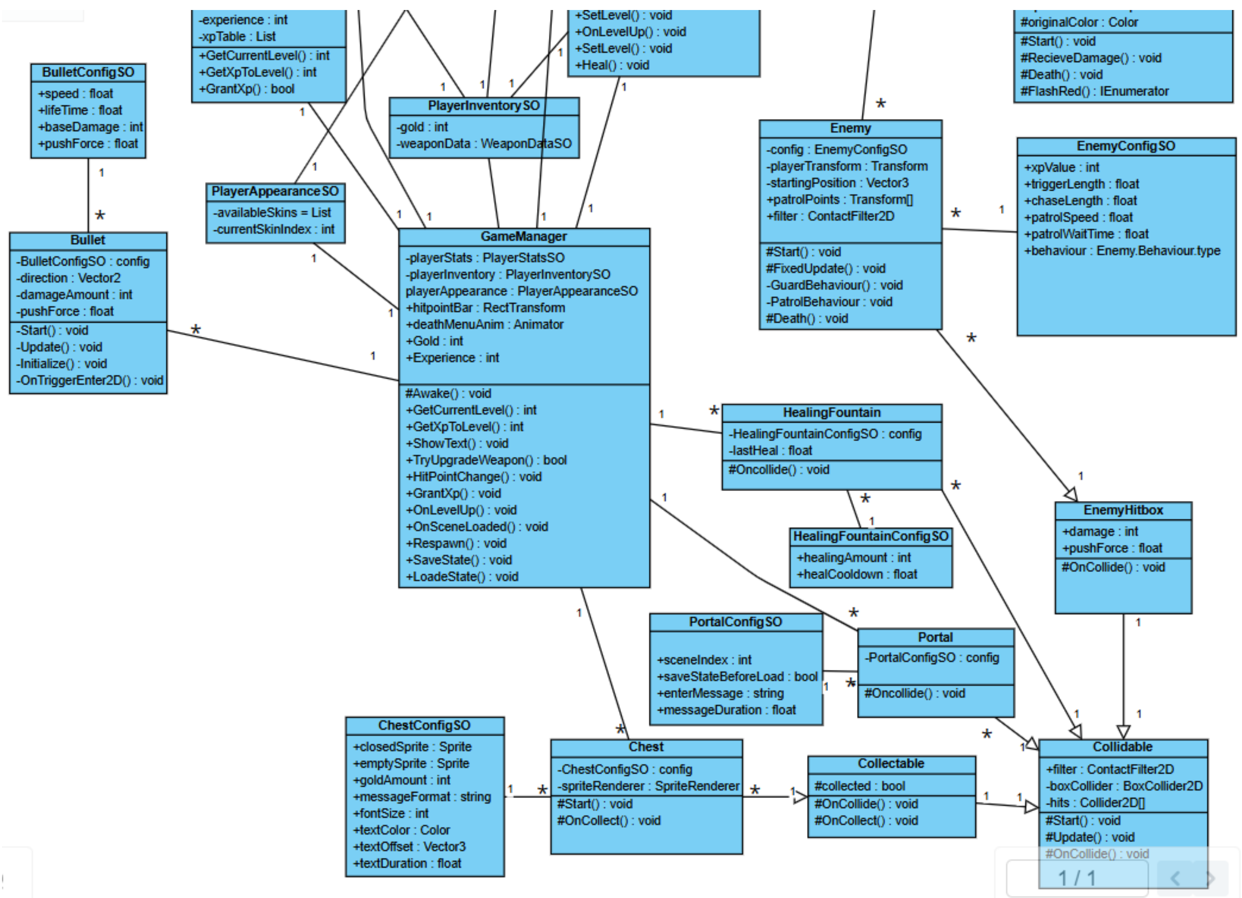


Рисунок 2.9 – продовження діаграми класів

Діаграма класів, що представлена на рисунку, відображає архітектурну модель 2D-ігрового застосунку, створеного з використанням підходу Scriptable Object Architecture (SOA). Модель побудована відповідно до об’єктно-орієнтованого підходу, застосованого у рушії Unity, та показує ключові сутності, їхні атрибути, методи і типи зв’язків.

Клас Fighter є одним з ключових класів. Він відповідає за механіки бою. Його атрибутами є:

– hitpoint;

– maxHitpoint;

– pushRecoverySpeed;

– immuneTime;

– lastImmune;

– pushDirection;

– spriteRenderer;

– originalColor.

До його методів належать:

– Start();

– ReceiveDamage();

– Death();

– FlashRed().

Наступним ключовим класом є Mover, він наслідує клас Fighter та потрібен для пересування ігрових об'єктів (таких як NPC та гравець) по ігровій сцені. Його атрибути:

– boxCollider;

– moveDelta;

– ySpeed;

– xSpeed;

– animator.

Методами цього класу є:

– Start();

– UpdateMotor().

Клас Player відповідає за персонажа, яким керує гравець. Його атрибутами є:

– inputSource;

– input;

– spriteRenderer;

– animator;

– lastMoveDir;

– isALive;

– originalColor.

До методів відносяться:

– Start();

– ReceiveDamage();

– Death();

– FixedUpdate();

– SwapSprite();

– OnLevelUp();

– SetLevel();

– SetLevel();

– Heal().

Клас CharacterMenu відповідає за механіку інвентаря гравця. Там є уся потрібна інформація про гравця та його предмети та характеристики. Його атрибутами є:

– characterSelectionSprite;

– weaponSprite;

– xpBar;

– playerStats;

– playerInventory;

– playerAppearance;

– currentCharacterSelection.

Методами є:

– OnArrowClick();

– OnSelectionChanged();

– OnUpgradeClick();

– UpdateMenu().

Клас Weapon потрібен для зброї гравця, що представлена у ігровому застосунку. Атрибути класу:

– WeaponData;

– spriteRenderer;

– anim;

– cooldown;

– lastSwing.

До методів відносяться:

– Start();

– Update();

– Swing();

– RefreshVisual();

– UpgradeWeapon();

– SetWeaponLevel().

Клас Bullet також є частиною механіки бою та відповідає за кулі, якими стріляє гравець. Атрибути класа:

– BulletConfigSO;

– direction;

– damageAmount;

– pushForce.

Методи класу:

– Start();

– Update();

– Initialize();

– OnTriggerEnter2D().

Клас Enemy відповідає за ворогів у грі. Він керує їхніми параметрами та поведінкою. Атрибути:

– config;

– playerTransform;

– startingPosition;

– patrolPoints;

– filter.

Методи класу:

– Start();

– FixedUpdate();

– GuardBehaviour();

– PatrolBehaviour();

– Death().

GameManager – це один із ключових класів усього застосунку, він зберігає в собі найбільше інформації та взаємодіє з майже кожним елементом гри. Його атрибутами є:

– playerStats;

– playerInventory;

– playerAppearance;

– hitpointBar;

– deathMenuAnim;

– Gold;

– Experience.

Методи класу:

– Awake();

– GetCurrentLevel();

– GetXpToLevel();

– TryUpgradeWeapon();

– HitpointChange();

– GrantXp();

– OnLevelUp();

– OnSceneLoaded();

– Respawn();

– SaveState();

– LoadeState().

Також на діаграмі класів відображені SO конфіги. Вони зберігають у собі дані, які потім використовуються класами. Кожен з перелічених до цього класів наслідує відповідний конфіг. Це потрібно для того, щоб розділити скрипти та дані, що дозволить легко змінювати дані, без необхідності втручання у код. До скриптів відносяться:

– EnemyConfigSO;

– BulletConfigSO;

– ChestConfigSO;

– PortalConfigSO;

– HealingFountainSO;

– PlayerInventorySO;

– PlayerStatsSO;

– PlayerAppearanceSO;

– WeaponDataSO;

– KeyboardPlayerInput.

Також на діаграмі зображено інтерфейс IPlayerInput. Він потрібен для реалізації керування ігровим персонажем.

2.5 Розробка діаграми послідовності дій

Для детального аналізу поведінки системи та її окремих підсистем у процесі виконання конкретних сценаріїв використовується діаграма послідовності (Sequence Diagram). Це один із видів діаграм взаємодії в UML, основним завданням якого є відображення часової динаміки обміну повідомленнями між об’єктами.

Діаграма послідовності показує:

– які саме об’єкти або компоненти беруть участь у сценарії;

– у якій послідовності вони взаємодіють між собою;

– які повідомлення та виклики методів надсилаються;

– які дії виконуються синхронно чи асинхронно;

– умови, що призводять до зміни стану об’єктів.

Цей тип діаграм є особливо корисним при моделюванні ігрових застосунків, оскільки такі системи характеризуються великою кількістю взаємодій між різними сутностями: персонажем, противниками, об’єктами середовища, конфігураційними компонентами (ScriptableObject), а також глобальними менеджерами.

Діаграма послідовності містить низку структурних компонентів:

– актори – зовнішні учасники, що ініціюють взаємодію (у грі це гравець);

– об’єкти/класи – сутності системи, які обмінюються повідомленнями: Player, Enemy, Mover, GameManager, ConfigSO тощо;

– життєві лінії (lifelines) – вертикальні лінії, що показують існування об’єкта в межах сценарію;

– повідомлення – стрілки, що представляють виклик методу або передання даних;

– активні фрагменти (activation bars) – позначають період виконання певної операції об’єктом.

На рисунках 2.12-2.13 зображено розроблені діаграми послідовностей дій для різних сценаріїв.

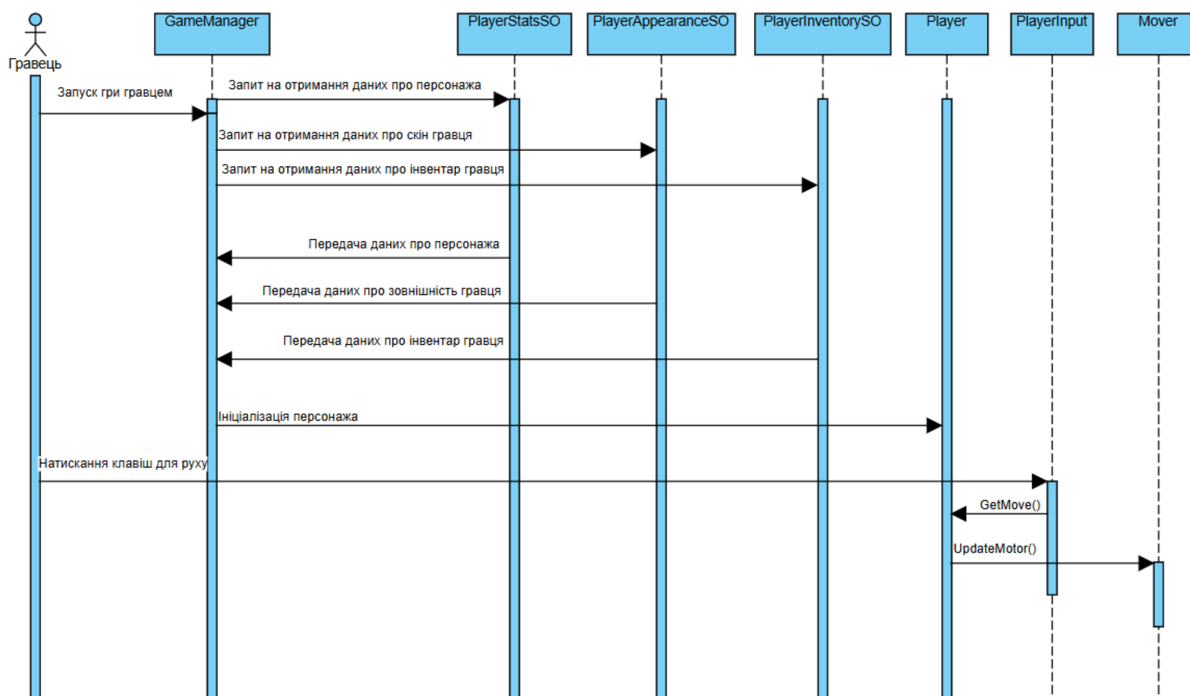


Рисунок 2.10 – Діаграма послідовності дій керування головним героєм

На діаграмі послідовності зображено динаміку взаємодії між ключовими компонентами ігрового застосунку під час процесу запуску гри та подальшого керування головним персонажем. Основними учасниками даного сценарію є актор «Гравець», а також внутрішні об’єкти системи: GameManager, PlayerStatsSO, PlayerAppearanceSO, PlayerInventorySO, Player, PlayerInput та Mover.

Сценарій починається з того, що актор «Гравець» ініціює запуск гри. У відповідь на це компонент GameManager активується та починає процес отримання необхідних даних для створення та ініціалізації головного героя.

GameManager послідовно надсилає запити на отримання конфігураційних даних із різних SO-компонентів, що відповідають за автономне зберігання стану персонажа:

– запит параметрів персонажа надсилається до PlayerStatsSO, який повертає інформацію про кількість досвіду, поточний рівень тощо;

– запит на отримання даних про зовнішність відправляється до PlayerAppearanceSO, що повертає вибраний індекс моделі персонажа та перелік доступних скінів;

– запит інвентарних даних надсилається до PlayerInventorySO, що повертає інформацію про зброю, кількість золота, характеристики предметів.

Кожен із цих SO-об’єктів передає відповідні дані назад до GameManager, забезпечуючи повну конфігурацію стану гравця.

Після отримання необхідної інформації GameManager викликає процедуру ініціалізації Player, надсилаючи персонажу відновлені параметри:

– рівень та досвід;

– зовнішній вигляд;

– стан зброї;

– інвентар.

На цьому етапі головний герой повністю готовий до участі у грі.

Після завершення ініціалізації сценарій переходить до активної взаємодії користувача з грою. При натисканні гравцем клавіш керування:

– компонент PlayerInput отримує дані про введення та викликає метод GetMove(), який повертає вектор напряму руху;

– отриманий вектор передається об’єкту Player, який ініціює переміщення;

– Player делегує рух компоненту Mover, який виконує метод UpdateMotor(), безпосередньо оновлюючи позицію персонажа у ігровому просторі.

Таким чином, від моменту натискання клавіші до фактичного переміщення персонажа відбувається чітко структурований ланцюг викликів.

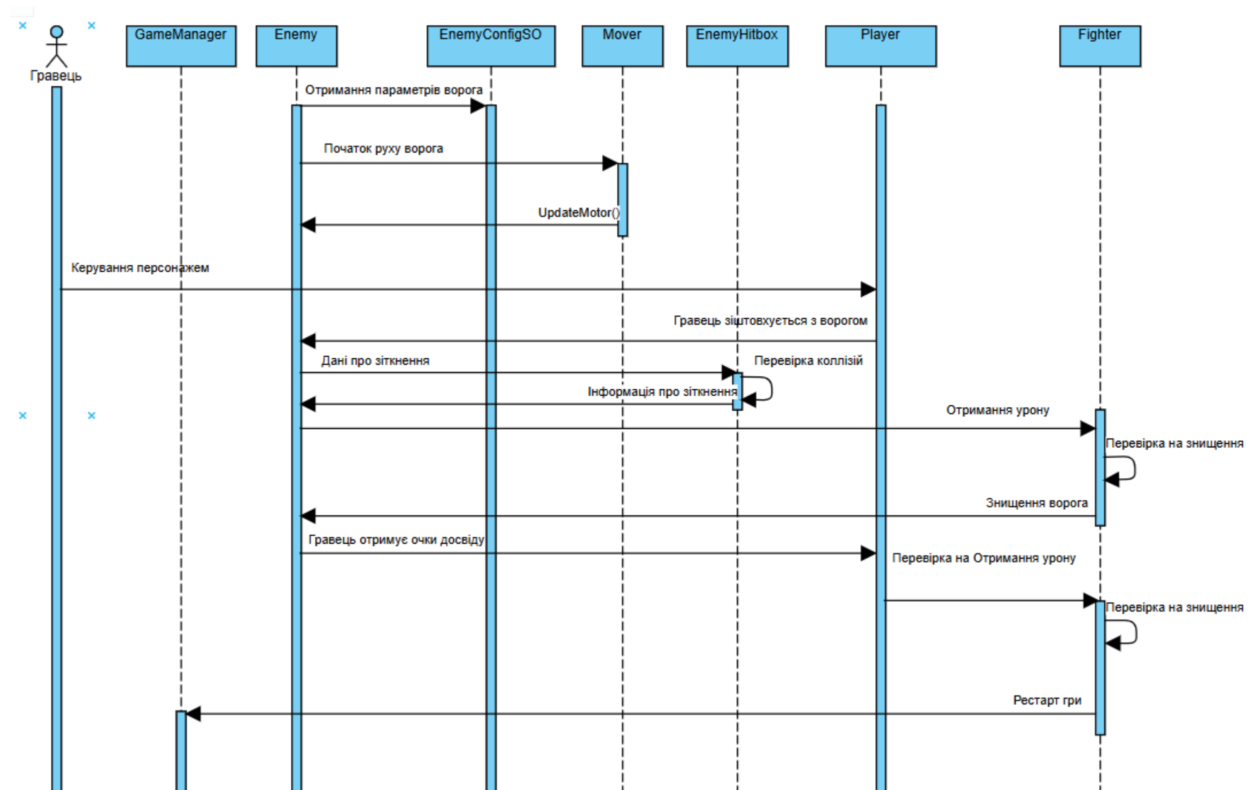


Рисунок 2.11 – Діаграма послідовності дій взаємодії гравця з ворогом

На цій діаграмі послідовності представлено повний цикл взаємодії між головним персонажем і ворогом під час ігрового процесу. Зображено, як об’єкти системи реагують на рух гравця, як відбувається зіткнення з NPC-ворогом, нанесення шкоди та нарахування досвіду, а також що відбувається у випадку знищення одного із учасників.

У сценарії беруть участь такі об’єкти: Гравець (актор), GameManager, Enemy, EnemyConfigSO, Mover, EnemyHitbox, Player, Fighter.

Сценарій починається з того, що актор «Гравець» керує головним персонажем у просторі гри. Персонаж пересувається за допомогою механізму,

описаного на іншій діаграмі, і в якийсь момент наближається до ворога, входячи в його зону видимості.

Enemy звертається до EnemyConfigSO, звідки отримує налаштування:

– дистанцію переслідування;

– швидкість руху;

– радіус тригера атаки;

– тип поведінки (Guard або Patrol);

– значення досвіду, що видається після знищення.

Це відповідає принципам SOA, оскільки поведінкові параметри не «зашиті» у ворога, а передаються через зовнішній конфігураційний модуль.

Після отримання параметрів ворог активує власний алгоритм:

– перевіряє положення гравця;

– визначає, чи входить він до зони переслідування;

– викликає метод UpdateMotor() через компонент Mover, який відповідає за фізичне переміщення.

Таким чином, ворог починає рух до гравця відповідно до встановленої поведінкової моделі.

У момент, коли ворог зближується з героєм:

– Player фіксує факт контакту – «гравець зіштовхується з ворогом»;

– інформація про зіткнення передається до компонента EnemyHitbox, який відповідає за перевірку колізій та нанесення шкоди.

EnemyHitbox обробляє подію та генерує результат зіткнення – виклик методу, який передає об’єкту Player інформацію про отриману шкоду.

При колізії:

– Player отримує шкоду через метод Fighter → ReceiveDamage();

– у гравця зменшується кількість життів;

– після цього Fighter виконує перевірку стану гравця на можливе знищення.

Якщо гравця не знищено – гра продовжується, якщо знищено – рівень перезапускається.

Паралельно з цим, гравець може атакувати ворога у відповідь. У цьому випадку ворог отримує шкоду та проходить перевірку на знищення. Якщо ворога було знищено, то:

– об’єкт Enemy викликає власний метод Death();

– ворог зникає зі сцени.

Далі викликається GameManager, який виконує:

– нарахування досвіду гравцю через SO PlayerStats;

– оновлення інтерфейсу досвіду;

– візуалізацію тексту «+XP».

3 ОПИС ПРИЙНЯТИХ ПРОЄКТНИХ РІШЕНЬ ПІД ЧАС РОЗРОБКИ МОДУЛЬНОЇ АРХІТЕКТУРИ ІГРОВОГО ЗАСТОСУНКУ

3.1 Опис архітектури (структури) ігрового застосунку

Архітектура програмного забезпечення є фундаментальною складовою будь-якої сучасної системи, включно з ігровими застосунками. Під архітектурою розуміють структурну організацію програмного продукту, яка визначає його внутрішню логіку, компоненти, способи їхньої взаємодії, принципи обміну даними, а також механізми розширення та модифікації. Іншими словами, архітектура задає «скелет» ігрової системи, на який надалі нашаровуються ігрові механіки, логіка персонажів, інтерфейс користувача та інші функціональні частини гри.

Наявність чітко сформованої архітектури є критично важливою з кількох причин:

– підтримуваність. Добре структурований проєкт легше аналізувати, масштабувати, модифікувати та виправляти помилки без ризику порушення цілісності системи;

– гнучкість. Архітектурні патерни дозволяють швидко додавати нові механіки або змінювати існуючі, не переписуючи усю програму з нуля;

– модульність. Правильно організовані компоненти мінімізують взаємну залежність та забезпечують повторне використання частин системи;

– продуктивність. Архітектурні рішення визначають, як викликаються функції, що зберігається в пам’яті, як працює оновлення станів об’єктів та як оптимізується взаємодія між ними;

– зрозумілість. Архітектурна схема дає розробникам і дослідникам цілісне уявлення про логіку роботи застосунку та структуру проєкту.

У рамках даної роботи обрано архітектурний підхід, орієнтований на Scriptable Object Architecture (SOA), що дозволяє розділяти дані, поведінку й

логіку конфігурації ігрових об’єктів між різними частинами системи. Це забезпечує більш гнучке керування параметрами гри, спрощує тестування та значно підвищує масштабовність проєкту.

На рисунку 3.1 показано схему розробленої архітектури ігрового застосунку.

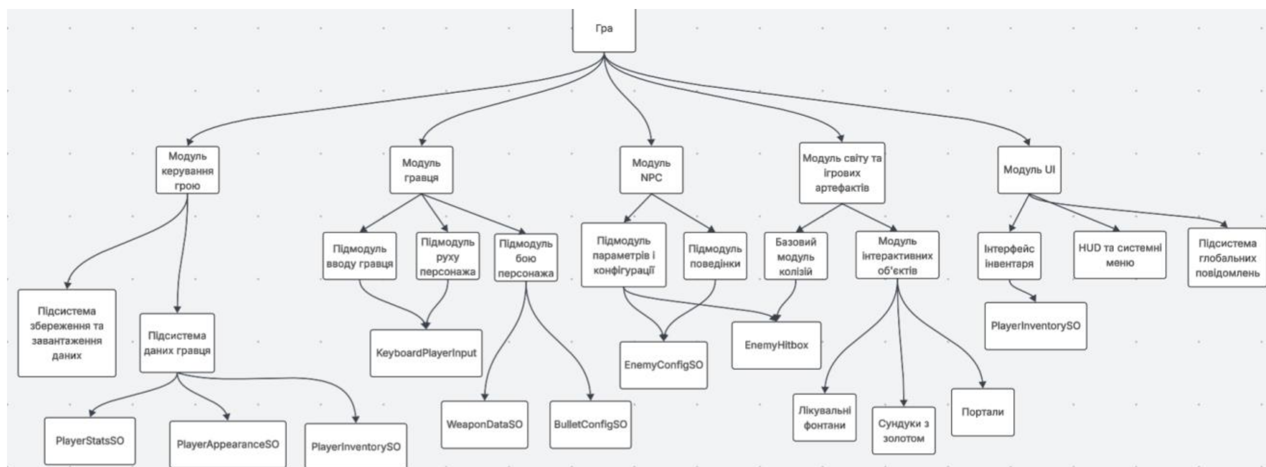


Рисунок 3.1 – Архітектура демонстраційного ігрового застосунку

На рисунку 3.1 показано модульну структуру демонстраційного ігрового застосунку, побудовану з урахуванням Scriptable Object Architecture. Кореневий елемент «Гра» відображає весь застосунок як єдину систему. Від нього відходять п’ять основних модулів верхнього рівня, кожен з яких відповідає за окремий аспект роботи гри: модуль керування грою, модуль гравця, модуль NPC, модуль світу та ігрових артефактів і модуль UI. Далі кожен із модулів декомпозується на підмодулі та підсистеми, серед яких окремо виділено конфігураційні об’єкти типу ScriptableObject.

Від вузла «Гра» першим відгалужується модуль керування грою – це логічний центр застосунку, що відповідає за глобальний стан, збереження прогресу та координацію інших підсистем. Усередині нього виділено дві ключові підсистеми:

– підсистема збереження та завантаження даних – реалізує логіку серіалізації стану гри (рівень, досвід, стан інвентаря, налаштування) та його відновлення при наступному запуску або переході між сценами;

– підсистема даних гравця, що працює поверх Scriptable Object-конфігів.

Сама підсистема даних гравця складається з ще 3 елементів:

– PlayerStatsSO – зберігає параметри розвитку персонажа (поточний досвід, таблиця XP, рівень тощо);

– PlayerAppearanceSO – описує зовнішній вигляд гравця (набір доступних скінів, вибраний індекс);

– PlayerInventorySO – містить інформацію про ресурси та предмети гравця (золото, поточна зброя, інші елементи інвентаря).

Таким чином модуль керування грою поєднує управління життєвим циклом гри з централізованим доступом до даних, що конфігуруються через SO-об’єкти.

Другим великим відгалуженням є модуль гравця, який описує всі аспекти поведінки основного персонажа. У ньому виділено три підмодулі:

– підмодуль вводу гравця – відповідає за абстракцію джерела вводу та працює через ScriptableObject, а саме KeyboardPlayerInput – конфігураційний об’єкт, який реалізує інтерфейс вводу, зчитує стан клавіатури та перетворює його на вектор руху;

– підмодуль руху персонажа – інкапсулює логіку пересування гравця (обробка векторів руху, фізична взаємодія зі світом, повороти спрайту);

– підмодуль бою персонажа, що оперує двома SO-конфігами – WeaponDataSO та BulletConfigSO.

Модуль NPC описує поведінку ворожих персонажів і дружніх агентів. Він містить:

– підмодуль параметрів і конфігурації, базованих на EnemyConfigSO – у цьому ScriptableObject задаються тип поведінки (гарда/патруль), радіуси виявлення, швидкість пересування, значення досвіду, що нараховується за знищення ворога, тощо;

– підмодуль поведінки – реалізує алгоритми руху, переслідування гравця, патрулювання та реакції на події світу з урахуванням параметрів із EnemyConfigSO;

– базовий модуль колізій – представлений компонентом EnemyHitbox, який відповідає за обробку зіткнень ворога з гравцем, снарядами та оточенням, формує структуру пошкодження і передає її в бойову логіку.

У результаті NPC-логіка розділена на конфігураційну частину (через SO) та виконавчу поведінку, що спрощує створення нових типів ворогів шляхом зміни лише даних.

Далі йде модуль світу та ігрових артефактів, який описує статичні та інтерактивні об’єкти рівня. Основний його елемент – модуль інтерактивних об’єктів, що об’єднує:

– лікувальні фонтани, параметри яких задаються відповідним SO-конфігом і визначають величину та частоту відновлення здоров’я при взаємодії гравця з об’єктом;

– сундуки з золотом, що використовують свій конфіг (ChestConfigSO) для опису кількості золота, графічних станів (закритий/порожній) та параметрів відображення повідомлення;

– портали, налаштовані через PortalConfigSO: у ньому задається цільова сцена, спосіб переходу та поведінка під час завантаження.

Таким чином модуль світу концентрує усі об’єкти, з якими гравець може безпосередньо взаємодіяти, а SO-конфіги дозволяють гнучко змінювати їх параметри без модифікації коду.

Останнім елементом верхнього рівня є модуль UI, який відповідає за відображення інформації та взаємодію користувача з системою. Він містить:

– інтерфейс інвентаря – екрани та панелі, які відображають стан інвентаря гравця й використовують PlayerInventorySO як джерело даних (кількість золота, поточна зброя, рівень прокачки тощо);

– HUD та системні меню – елементи головного екрану, шкали здоров’я, досвіду, повідомлення про рівень, пауза-меню, меню виходу з гри;

– підсистема глобальних повідомлень – відображає плаваючі тексти (набір досвіду, отримання золота, влучання по ворогу), синхронізуючи їх із подіями, що надходять від модулів гравця, NPC та світу.

У підсумку ця діаграма демонструє модульну архітектуру ігрового застосунку, де кожен модуль відповідає за свою область відповідальності, а ключові параметри геймплею винесені в окремі ScriptableObject-конфіги. Такий підхід полегшує повторне використання та розширення компонентів, спрощує налаштування балансу гри й наочно показує, як саме SOA інтегрується в структуру 2D-гри жанру Top-Down RPG.

3.2 Розробка ігрового меню

Ігрове меню є одним із ключових компонентів будь-якого сучасного ігрового застосунку, оскільки воно забезпечує користувача засобами взаємодії з системою поза межами основного геймплейного процесу. До його складу входять різні елементи інтерфейсу – інвентар гравця, системні панелі керування, меню вибору параметрів персонажа, відображення статистичних даних та інші інформаційні модулі. Саме через меню гравець отримує доступ до внутрішніх механік гри: переглядає стан персонажа, змінює зовнішність, взаємодіє зі спорядженням, відстежує прогрес та здійснює налаштування.
На рисунках 3.2 – 3.5 показано елементи коду для функціонування інвентарю гравця.

```
// Оновлення спрайтів при зміні скіна
private void OnSelectionChanged()
{
    var appearance = GameManager.instance.PlayerAppearance;
    if (appearance == null || appearance.Skins == null || appearance.Skins.Count == 0)
        return;

    int idx = appearance.CurrentSkinIndex;
    if (idx < 0 || idx >= appearance.Skins.Count)
        return;

    Sprite skinSprite = (Sprite)appearance.Skins[idx];

    // Спрайт у вікні інвентарю
    if (characterSelectionSprite != null)
        characterSelectionSprite.sprite = skinSprite;

    // Спрайт реального гравця в світі
    if (player != null)
        player.SwapSprite(idx);
}

// Кнопка апгрейду зброї
public void OnUpgradeClick()
{
    if (GameManager.instance.TryUpgradeWeapon())
    {
        UpdateMenu();
    }
}
```

Рисунок 3.2 – Елемент коду для реалізації зміни скіна гравця

```csharp
// Поточний вибір скіна в меню
private int currentCharacterSelection = 0;

// === Викликається стрілками в меню (право/ліво) ===
public void OnArrowClick(bool right)
{
    var appearance = GameManager.instance.PlayerAppearance;
    if (appearance == null || appearance.Skins == null || appearance.Skins.Count == 0)
        return;

    // змінюємо індекс
    if (right)
        currentCharacterSelection++;
    else
        currentCharacterSelection--;

    int count = appearance.Skins.Count;
    currentCharacterSelection = (currentCharacterSelection % count + count) % count;

    // оновлюємо SO і відображення
    appearance.CurrentSkinIndex = currentCharacterSelection;
    OnSelectionChanged();
}
```

Рисунок 3.3 – Елемент коду для відображення кнопок зміни скіна персонажа

```csharp
public void UpdateMenu()
{
    var gm = GameManager.instance;
    if (gm == null)
        return;

    // ===== ЗБРОЯ =====
    var weaponData = playerInventory != null ? playerInventory.WeaponData : null;
    if (weaponData != null)
    {
        if (weaponSprite != null)
            weaponSprite.sprite = weaponData.CurrentSprite;

        if (!weaponData.CanUpgrade)
            upgradeCostText.text = "MAX";
        else
            upgradeCostText.text = weaponData.NextUpgradePrice.ToString();
    }

    // ===== РІВЕНЬ, HP, GOLD =====
    if (playerStats != null)
    {
        levelText.text = playerStats.GetCurrentLevel().ToString();
        goldText.text = gm.Gold.ToString();
    }

    if (player != null)
    {
        hitpointText.text = player.hitpoint.ToString();
    }
```

Рисунок 3.4 – Елемент коду для відображення інформації про зброю та параметри гравця

```
// ===== XP BAR =====
if (playerStats != null && playerStats.XpTable != null && playerStats.XpTable.Count > 0)
{
    int currLevel = playerStats.GetCurrentLevel();

    if (currLevel >= playerStats.XpTable.Count)
    {
        // максимум рівня - показуємо загальний XP
        xpText.text = playerStats.Experience.ToString() + " total experience points";
        xpBar.localScale = Vector3.one;
    }
    else
    {
        int prevLevelXp = playerStats.GetXpToLevel(currLevel - 1);
        int currLevelXp = playerStats.GetXpToLevel(currLevel);

        int diff = currLevelXp - prevLevelXp;
        int currXpIntoLevel = playerStats.Experience - prevLevelXp;

        float completionRatio = diff > 0 ? (float)currXpIntoLevel / diff : 1f;
        xpBar.localScale = new Vector3(completionRatio, 1, 1);
        xpText.text = currXpIntoLevel + " / " + diff;
    }
}

// Оновити скін у меню згідно поточного стану SO
var appearance = GameManager.instance.PlayerAppearance;
currentCharacterSelection = appearance != null ? appearance.CurrentSkinIndex : 0;
OnSelectionChanged();
```

Рисунок 3.5 – Елемент коду для відображення полоски досвіду

На представлених рисунках зображено реалізацію інвентарю гравця. Там відображається кількість очок досвіду, кількість золота та очок здоров’я гравця. Також інвентар дозволяє змінювати зовнішність персонажа та покращувати зброю. Для реалізації механік інвентарю було використано архітектурний шаблон SOA. Він дозволяє відокремити логіку від даних. SOA-конфіги наведено на рисунках 3.6-3.9

```
[CreateAssetMenu(menuName = "SO/Player Inventory")]
public class PlayerInventorySO : ScriptableObject
{
    [SerializeField]
    private int gold;
    [SerializeField]
    private WeaponDataSO weaponData;

    public int Gold
    {
        get => gold;
        set => gold = Mathf.Max(0, value);
    }

    public WeaponDataSO WeaponData => weaponData;
}
```

Рисунок 3.6 – SOA-конфіг інвентарю гравця

```csharp
[CreateAssetMenu(menuName = "SO/Player Appearance")]
public class PlayerAppearanceSO : ScriptableObject
{
    [SerializeField]
    private List<Sprite> availableSkins = new List<Sprite>();
    [SerializeField]
    private int currentSkinIndex;

    public IList<Sprite> Skins => availableSkins;

    public int CurrentSkinIndex
    {
        get => currentSkinIndex;
        set
        {
            if (availableSkins.Count == 0)
            {
                currentSkinIndex = 0;
                return;
            }

            currentSkinIndex = (value % availableSkins.Count + availableSkins.Count)
                               % availableSkins.Count;
        }
    }
}
```

Рисунок 3.7 – SOA-конфіг для зміни зовнішності гравця

```csharp
[CreateAssetMenu(menuName = "SO/Player Stats")]
public class PlayerStatsSO : ScriptableObject
{
    [SerializeField]
    private int experience;
    [SerializeField]
    private List<int> xpTable = new List<int>();

    public int Experience
    {
        get => experience;
        set => experience = Mathf.Max(0, value);
    }

    public IReadOnlyList<int> XpTable => xpTable;

    public int GetCurrentLevel()
    {
        int r = 0;
        int add = 0;

        // логіка перенесена з GameManager.GetCurrentLevel() :contentReference[oaicite:3]{index=3}
        while (r < xpTable.Count && experience >= add)
        {
            add += xpTable[r];
            r++;
        }

        return Mathf.Clamp(r, 1, xpTable.Count);
    }
```

Рисунок 3.8 – SOA-конфіг для відображення параметрів гравця

```csharp
public int GetXpToLevel(int level)
{
    int r = 0;
    int xp = 0;

    // логіка з GameManager.GetXpToLevel() :contentReference[oaicite:4]{index=4}
    while (r < level && r < xpTable.Count)
    {
        xp += xpTable[r];
        r++;
    }

    return xp;
}

/// <summary>
/// Повертає true, якщо стався апгрейд рівня.
/// </summary>
public bool GrantXp(int amount)
{
    int prevLevel = GetCurrentLevel();
    Experience += amount;
    int newLevel = GetCurrentLevel();
    return newLevel > prevLevel;
}
```

Рисунок 3.9 – Продовження SOA-конфігу для відображення параметрів гравця

3.3 Розробка поведінкової моделі керованого персонажа

Поведінкова модель керованого персонажа є центральним елементом ігрової логіки, оскільки саме вона визначає реакцію героя на дії гравця, взаємодію з елементами ігрового світу та участь у бойових чи дослідницьких сценаріях. Від коректності її побудови залежить загальний ігровий досвід, плавність керування, відповідність дій персонажа очікуванням користувача та інтеграція з іншими підсистемами застосунку.

У межах представленої архітектури поведінкова модель реалізована модульно та включає декілька ключових компонентів:

– модуль обробки вводу гравця, що відповідає за отримання команд і їх трансформацію у внутрішні дії;

– модуль руху, який реалізує переміщення персонажа з урахуванням фізичних та анімаційних параметрів;

– модуль бою, що вирішує питання нанесення та отримання шкоди, використання зброї та реакції на ворожі взаємодії;

– підмодулі станів і параметрів, реалізовані на основі ScriptableObject та відповідальні за збереження характеристик героя між сесіями.

Важливою особливістю розробленої поведінкової моделі є її відокремлення даних від логіки та високий рівень повторного використання компонентів, що досягається за рахунок Scriptable Object Architecture. Це дозволяє легко змінювати параметри персонажа (швидкість, здоров'я, спорядження, характеристики зброї тощо) без модифікації основного коду. Такий підхід суттєво спрощує модифікацію та розширення поведінкових сценаріїв, а також покращує тестованість системи.

На рисунку 3.10 зображено SOA-конфіг для керування персонажем за допомогою клавіатури. На рисунку 3.11 зображено код інтерфейсу для керування головним героєм. На рисунках 3.12-3.14 показано елементи коду для керування персонажем.

```
[CreateAssetMenu(menuName = "Input/Keyboard Player Input")]
public class KeyboardPlayerInput : ScriptableObject, IPlayerInput
{
    [SerializeField]
    private string horizontalAxis = "Horizontal";
    [SerializeField]
    private string verticalAxis = "Vertical";

    public Vector2 GetMove()
    {
        float x = Input.GetAxisRaw(horizontalAxis);
        float y = Input.GetAxisRaw(verticalAxis);
        var v = new Vector2(x, y);
        return v.sqrMagnitude > 1f ? v.normalized : v;
    }

}
```

Рисунок 3.10 – SOA-конфіг для вводу через клавіатуру

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

public interface IPlayerInput
{
    // Нормалізований напрям руху [-1..1] по кожній осі
    Vector2 GetMove();
}
```

Рисунок 3.11 – Код інтерфейсу для керування персонажем

```
public class Player : Mover
{
    // ДОДАЛИ: посилання на ScriptableObject з ввідом
    [Header("Input (SO)")]
    [SerializeField]
    private ScriptableObject inputSource; // признач у інспекторі KeyboardPlayerInput або AIPlayerInput
    private IPlayerInput input; // інтерфейс

    private SpriteRenderer spriteRenderer;
    private Animator animator;
    public Vector2 lastMoveDir = Vector2.right;

    private bool isAlive = true;
    protected Color originalColor;

    protected override void Start()
    {
        base.Start();

        spriteRenderer = GetComponent<SpriteRenderer>();
        animator = GetComponent<Animator>();

        if (spriteRenderer != null)
            originalColor = spriteRenderer.color;

        // ініціалізуємо інтерфейс вводу з SO
        input = inputSource as IPlayerInput;
        if (input == null)
        {
            Debug.LogError("Player: inputSource не реалізує IPlayerInput. Признач KeyboardPlayerInput або AIPlayerInput asset.");
        }
    }
```

Рисунок 3.12 – Елемент коду для реалізації керування перонажем

У наведеному фрагменті коду реалізовано ініціалізацію керованого персонажа та підключення до нього модулів введення відповідно до концепції Scriptable Object Architecture. Клас Player успадковує базовий руховий компонент і доповнює його власною логікою, пов’язаною з отриманням даних про ввід. Через серіалізоване поле до об’єкта підключається ScriptableObject, який повинен реалізовувати інтерфейс IPlayerInput. Це дозволяє в інспекторі легко замінювати різні джерела вводу – наприклад, вибирати між керуванням від клавіатури, ІІ-модулем або будь-яким іншим модулем, не змінюючи структуру основного класу.

```
private void FixedUpdate()
{
    if (!isAlive || input == null)
        return;

    // беремо напрям з SO, а не з Input.GetAxisRaw
    Vector2 move = input.GetMove();
    Vector3 moveDelta = new Vector3(move.x, move.y, 0);

    UpdateMotor(moveDelta);

    if (moveDelta != Vector3.zero)
    {
        lastMoveDir = move.normalized;
        if (animator != null)
            animator.SetBool("IsRun", true);
    }
    else
    {
        if (animator != null)
            animator.SetBool("IsRun", false);
    }
}
```

Рисунок 3.13 – Елемент коду для реалізації анімацій персонажа

Метод FixedUpdate() відповідає за обробку руху керованого персонажа та оновлення його анімаційного стану з фіксованою частотою кадрів. На початку виконання здійснюється перевірка, чи є персонаж живим та чи доступний модуль введення; якщо одна з умов не виконується, подальша логіка не запускається. Напрямок руху отримується не з класичного Input, а через спеціалізований модуль введення, реалізований за допомогою Scriptable Object

Architecture, що забезпечує відокремлення вводу від логіки руху та підвищує модульність архітектури.

Отриманий напрямок конвертується у вектор зміщення, після чого передається у моторну систему персонажа, яка відповідає за фактичне оновлення його позиції. Паралельно виконується обробка анімації: якщо персонаж рухається, зберігається останній напрямок руху та активується анімація бігу; якщо ж введення не надходить, персонаж переходить у стан спокою. Таким чином, метод забезпечує синхронізовану роботу підсистем руху, вводу та анімації, підтримуючи принципи розподілення відповідальностей та модульності, закладені у SOA-архітектуру.

3.4 Розробка поведінкової моделі некерованих персонажів

Некеровані персонажі являють собою одну з ключових підсистем ігрового застосунку, оскільки саме вони формують активність і динаміку ігрового світу, забезпечують інтерактивність оточення та створюють різноманітні ігрові сценарії для взаємодії з головним персонажем. Вони можуть виконувати різні ролі: супротивники, нейтральні істоти, союзники або об’єкти, що реагують на дії гравця. Тому поведінкова модель NPC має бути гнучкою, розширюваною та здатною адаптуватися до різних типів сцен і ігрових механік.

Поведінка NPC у створеному застосунку базується на поєднанні модульної архітектури Unity та Scriptable Object Architecture (SOA), що дозволяє відділити логіку від даних і забезпечити можливість зміни характеристик персонажів без внесення змін до основного коду. Кожний NPC функціонує на основі набору конфігурацій, які визначають його параметри: швидкість руху, радіус виявлення, тип поведінки (патрулювання, переслідування, охорона території), тривалість очікування, бойові характеристики тощо. Завдяки цьому створення нових NPC або налаштування існуючих зводиться до редагування їхніх ScriptableObject-конфігів.

На рисунку 3.14 зображено розроблений SOA-конфіг для некерованих персонажів. На рисунках 3.15-3.17 зображено розроблені скрипти для некерованих персонажів.

```
[CreateAssetMenu(menuName = "SO/Enemy Config", fileName = "New Enemy Config")]
public class EnemyConfigSO : ScriptableObject
{
    public Enemy.BehaviorType behavior = Enemy.BehaviorType.Guard;

    [Header("Combat / XP")]
    public int xpValue = 1;

    [Header("Guard / Chase")]
    public float triggerLength = 1f;
    public float chaseLength = 3f;

    [Header("Patrol")]
    public float patrolSpeed = 1f;
    public float patrolWaitTime = 1.5f;
}
```

Рисунок 3.14 – SOA-конфіг для некерованих персонажів

```
public enum BehaviorType { Guard, Patrol }

[Header("Config (SO)")]
[SerializeField]
private EnemyConfigSO config;

// runtime state
private bool chasing;
private bool collidingWithPlayer;
private Transform playerTransform;
private Vector3 startingPosition;

[Header("Patrol Settings (scene)")]
public Transform[] patrolPoints;      // точки патрулювання задаємо в сцені
private int patrolIndex = 0;
private float patrolWaitCounter = 0f;

// Hitbox
public ContactFilter2D filter;
private Collider2D[] hits = new Collider2D[10];

// зручні властивості для доступу до параметрів з SO
private BehaviorType Behavior => config != null ? config.behavior : BehaviorType.Guard;
private float TriggerLength => config != null ? config.triggerLength : 1f;
private float ChaseLength => config != null ? config.chaseLength : 3f;
private float PatrolSpeed => config != null ? config.patrolSpeed : 1f;
private float PatrolWaitTime => config != null ? config.patrolWaitTime : 1.5f;
private int XpValue => config != null ? config.xpValue : 1;
```

Рисунок 3.15 – Елемент коду розробленого скрипта для NPC

```
private void FixedUpdate()
{
    // перевірка зіткнень із гравцем
    collidingWithPlayer = false;
    boxCollider.OverlapCollider(filter, hits);
    for (int i = 0; i < hits.Length; i++)
    {
        if (hits[i] == null)
            continue;

        if (hits[i].CompareTag("Fighter") && hits[i].name == "Player")
            collidingWithPlayer = true;

        hits[i] = null;
    }

    // вибір поведінки з SO
    switch (Behavior)
    {
        case BehaviorType.Guard:
            GuardBehavior();
            break;
        case BehaviorType.Patrol:
            PatrolBehavior();
            break;
    }
}
```

Рисунок 3.16 – Елемент коду розробленого скрипту для NPC

```
protected override void Death()
{
    Destroy(gameObject);
    GameManager.instance.GrantXp(XpValue);
    GameManager.instance.ShowText(
        "+" + XpValue + " xp",
        30,
        Color.magenta,
        transform.position,
        Vector3.up * 40,
        1.0f
    );
}
```

Рисунок 3.17 – Елемент коду розробленого скрипту для NPC

На наведених рисунках показано елементи коду розроблених скриптів для NPC. У даних елементах коду показано які дані підтягуються із скрипту та що скрипт робить потім із цими даними.

На рисунку 3.17 зображено метод Death() який відповідає за механіку знищення ворога, після чого гравець отримує очки досвіду, а на екрані з’являється відповідне повідомлення.

На рисунках 3.18-3.20 показано елементи коду, які реалізують поведінку NPC.

```
private void GuardBehavior()
{
    if (Vector3.Distance(playerTransform.position, startingPosition) < ChaseLength)
    {
        if (Vector3.Distance(playerTransform.position, startingPosition) < TriggerLength)
        {
            chasing = true;
        }

        if (chasing && !collidingWithPlayer)
        {
            UpdateMotor((playerTransform.position - transform.position).normalized);
        }
        else if (!chasing)
        {
            UpdateMotor((startingPosition - transform.position).normalized);
        }
    }
    else
    {
        UpdateMotor((startingPosition - transform.position).normalized);
        chasing = false;
    }
}
```

Рисунок 3.18 – Елемент коду для ворогів-охоронців

```
private void PatrolBehavior()
{
    float distanceToPlayer = Vector3.Distance(playerTransform.position, transform.position);

    // --- ПЕРЕСЛІДУВАННЯ ГРАВЦЯ ---
    bool playerInChase = distanceToPlayer < ChaseLength;
    bool playerInTrigger = distanceToPlayer < TriggerLength;

    if (playerInChase)
    {
        if (playerInTrigger)
            chasing = true;

        if (chasing && !collidingWithPlayer)
        {
            Vector3 chaseDir = (playerTransform.position - transform.position).normalized;
            UpdateMotor(chaseDir);   // рух за гравцем
            return;                  // важливо: НЕ патрулюємо в цьому кадрі
        }
    }
    else
    {
        // гравець далеко – повертаємось у режим патруля
        chasing = false;
    }

    // --- ПАТРУЛЮВАННЯ ---
    if (patrolPoints == null || patrolPoints.Length == 0)
    {
        Debug.LogWarning($"Enemy '{name}' has Patrol behavior but no patrol points assigned.");
        return;
    }
```

Рисунок 3.19 – Елемент коду для патрулюючих ворогів

```
Transform targetPoint = patrolPoints[patrolIndex];
float distanceToPoint = Vector3.Distance(transform.position, targetPoint.position);

// якщо дійшли до точки - чекаємо трохи та переходимо до наступної
if (distanceToPoint <= 0.1f)
{
    patrolWaitCounter += Time.deltaTime;
    if (patrolWaitCounter >= PatrolWaitTime)
    {
        patrolIndex = (patrolIndex + 1) % patrolPoints.Length;
        patrolWaitCounter = 0f;
    }

    UpdateMotor(Vector3.zero); // стоїмо під час очікування
}
else
{
    // рухаємося до поточної точки
    Vector3 dir = (targetPoint.position - transform.position).normalized;
    UpdateMotor(dir * PatrolSpeed);
}
```

Рисунок 3.20 – Продовження коду патрулюючих ворогів

На наведених рисунках показано реалізацію поведінкової моделі NPC. У противників є 2 види поведінки. Перший – це NPC-охоронці. Вони охороняються вказану позицію та як тільки гравець потрапляє у радіус їхнього зору починають переслідувати його. Якщо гравець покидає радіус дії, то переслідування припиняється, а NPC повертається на стартову позицію.

Другий тип ворогів – це патрульні. Вони патрулюють задану область та починають переслідування гравця, якщо той потрапляє у радіус їхнього зору. Маршрут патрулювання задається невидимими точками, по яким ворог циклічно рухається. Завдяки SOA, кількість таких точок може бути унікальною для кожного противника. У разі, якщо гравець вийшов з радіусу зору противника, NPC повертається до патрулювання з першої точки.

3.5 Розробка бойових механік ігрового застосунку

Бойова система є одним із ключових елементів геймплею, що визначає динаміку, складність і загальний характер ігрової взаємодії. Саме механіка бою забезпечує емоційний драйв, стимулює дослідження ігрового світу та формує

виклик для гравця. У межах створеного ігрового застосунку бойова система розроблена на основі модульної архітектури, що дозволяє гнучко змінювати параметри зброї, ворожих одиниць і поведінки бою без необхідності модифікувати основну частину коду.

За рахунок використання Scriptable Object Architecture (SOA) бойові механіки стали незалежними від конкретних класів, легко масштабуються та допускають створення нових типів зброї або снарядів без дублювання логіки. Наприклад, зміна швидкості пострілу, типу атаки чи значення шкоди реалізується через редагування відповідного конфігураційного об’єкта, що робить бойову систему відкритою для розширення та подальшої модифікації.

На рисунках 3.21 – 3.23 зображено SOA-конфіги зброї.

```
[CreateAssetMenu(menuName = "SO/Bullet Config", fileName = "New Bullet Config")]
public class BulletConfigSO : ScriptableObject
{
    [Header("Movement")]
    public float speed = 10f;
    public float lifeTime = 2f;

    [Header("Damage")]
    public int baseDamage = 1;
    public float pushForce = 2f;
}
```

Рисунок 3.21 – SOA-конфіг для куль

```
[CreateAssetMenu(menuName = "SO/Weapon Data")]
public class WeaponDataSO : ScriptableObject
{
    [Header("Бойові параметри по рівнях")]
    [SerializeField]
    private int[] damagePoints;
    [SerializeField]
    private float[] pushForces;

    [Header("Візуалізація по рівнях")]
    [SerializeField]
    private List<Sprite> levelSprites;

    [Header("Ціни покращення")]
    [SerializeField]
    private List<int> upgradePrices;

    [SerializeField]
    private int currentLevel = 0;

    public int CurrentLevel => currentLevel;
    public int MaxLevel => Mathf.Min(damagePoints.Length, pushForces.Length);

    public int CurrentDamage =>
        damagePoints[Mathf.Clamp(currentLevel, 0, damagePoints.Length - 1)];

    public float CurrentPushForce =>
        pushForces[Mathf.Clamp(currentLevel, 0, pushForces.Length - 1)];
```

Рисунок 3.22 – Елемент коду SOA-конфігу для зброї

```
public Sprite CurrentSprite =>
    (levelSprites != null && levelSprites.Count > 0)
        ? levelSprites[Mathf.Clamp(currentLevel, 0, levelSprites.Count - 1)]
        : null;

public bool CanUpgrade => currentLevel < upgradePrices.Count;
public int NextUpgradePrice => CanUpgrade ? upgradePrices[currentLevel] : 0;

public void SetLevel(int level)
{
    currentLevel = Mathf.Clamp(level, 0, MaxLevel - 1);
}

public void Upgrade()
{
    if (CanUpgrade)
        currentLevel++;
}
```

Рисунок 3.23 – Продовження SOA-конфігу для зброї

Показані на рисунках SOA-скрипти дозволяють відділити дані від основної логіки задля спрощення процесу зміни даних, а також вони дозволяють легко створювати нові екземпляри куль та зброї без необхідності змінювати код та не будуть впливати на вже існуючи екземпляри у грі.

На рисунках 3.24-3.26 показано елементи коду скрипта зброї.

```
public class Weapon : Collidable
{
    [Header("Дані зброї (SO)")]
    [SerializeField]
    private WeaponDataSO weaponData;
    [SerializeField]
    private SpriteRenderer spriteRenderer;

    // Swing
    private Animator anim;
    private float cooldown = 0.5f;
    private float lastSwing;

    protected override void Start()
    {
        base.Start();
        anim = GetComponent<Animator>();

        // оновити спрайт згідно поточного рівня
        RefreshVisual();
    }
```

Рисунок 3.25 – Елемент коду скрипта зброї

Цей фрагмент коду реалізує ініціалізацію зброї, що використовує Scriptable Object Architecture для конфігурації своїх параметрів. Об’єкт Weapon містить посилання на ScriptableObject типу WeaponDataSO, у якому зберігаються всі змінні, що описують характеристики зброї, включно з її візуальним оформленням, рівнем, базовими значеннями атаки та іншими параметрами. Такий підхід дозволяє легко змінювати або розширювати типи зброї без модифікації коду, оскільки всі дані зберігаються у зовнішньому SO-контейнері. Додатково компонент зберігає посилання на SpriteRenderer, що відповідає за відображення відповідного графічного елементу.

```
protected override void Update()
{
    base.Update();

    if (Input.GetKeyDown(KeyCode.Space))
    {
        if (Time.time - lastSwing > cooldown)
        {
            lastSwing = Time.time;
            Swing();
        }
    }
}

protected override void OnCollide(Collider2D coll)
{
    if (coll.tag == "Fighter")
    {
        if (coll.name == "Player")
            return;

        Damage dmg = new Damage
        {
            damageAmount = weaponData.CurrentDamage,
            origin = transform.position,
            pushForce = weaponData.CurrentPushForce
        };

        coll.SendMessage("ReceiveDamage", dmg);
    }
}
```

Рисунок 3.26 – Елемент коду для скрипту зброї

Цей фрагмент коду реалізує логіку атакування та обробки зіткнень зброї з іншими об’єктами у грі. У методі Update перевіряється натискання клавіші пробілу, що виконує роль тригера атаки. Якщо з моменту останнього удару минув інтервал, більший за встановлений час перезарядки, відбувається оновлення часу останнього удару та виклик методу Swing(), який запускає анімацію атаки та пов’язану з нею логіку. Така реалізація забезпечує контрольовану частоту атак і запобігає надмірному спамінгу ударів.

У методі OnCollide обробляється зіткнення зброї з іншими об’єктами, що мають тег "Fighter". Якщо об’єктом зіткнення є гравець, обробка переривається, щоб уникнути нанесення шкоди самому собі. В інших випадках створюється структура Damage, яка формується на основі даних зі ScriptableObject WeaponDataSO: поточного значення шкоди, сили відштовхування та позиції джерела удару. Після формування цієї структури іншому об’єкту передається повідомлення ReceiveDamage, яке передає йому інформацію про нанесену шкоду.

```
private void Swing()
{
    anim.SetTrigger("Swing");
}

public void RefreshVisual()
{
    if (spriteRenderer != null && weaponData.CurrentSprite != null)
        spriteRenderer.sprite = weaponData.CurrentSprite;
}

public void UpgradeWeapon()
{
    weaponData.Upgrade();
    RefreshVisual();
}

public void SetWeaponLevel(int level)
{
    weaponData.SetLevel(level);
    RefreshVisual();
}

public int CurrentLevel => weaponData.CurrentLevel;
```

Рисунок 3.27 – Елемент коду для скрипту зброї

Цей фрагмент коду відповідає за виклик анімації атаки та оновлення візуальних і функціональних параметрів зброї відповідно до її поточного рівня, зберігаючи при цьому дані в ScriptableObject. Метод Swing активує тригер аніматора, який запускає анімацію удару, забезпечуючи коректну візуальну реакцію на дію гравця. Завдяки цьому логіка атаки залишається простою, а всі візуальні аспекти повністю делегуються системі анімації Unity.

Метод RefreshVisual оновлює спрайт зброї згідно з актуальними даними, отриманими зі ScriptableObject WeaponDataSO. Це дає можливість автоматично змінювати зовнішній вигляд об'єкта при прокачуванні зброї або переході на новий рівень. При цьому логіка зміни спрайтів не прив'язана до самого класу зброї – усі параметри зберігаються в зовнішньому SO-ресурсі, що забезпечує високу розширюваність системи.

Методи UpgradeWeapon та SetWeaponLevel інкапсулюють логіку збільшення рівня зброї та встановлення рівня вручну. Обидва методи викликають оновлення візуального стану після зміни параметрів ScriptableObject, що гарантує узгодженість між внутрішніми характеристиками та їх відображенням у грі. Властивість CurrentLevel забезпечує зручний доступ до поточного рівня зброї, також отримуючи значення з SO.

3.6 Розробка артефактів ігрового світу

Артефакти ігрового світу є невід'ємною складовою структури будь-якого ігрового застосунку, оскільки вони формують взаємодію гравця з оточенням та створюють умови для розвитку геймплею. Під артефактами у даній роботі розуміються всі об'єкти середовища, здатні реагувати на дії гравця або впливати на його стан: сундуки з ресурсами, лікувальні фонтани, портали переходу між рівнями та інші інтерактивні елементи.

Основною особливістю артефактів є те, що вони виконують допоміжні, але важливі функції: забезпечують прогрес персонажа, відновлюють його

характеристики, відкривають доступ до нових зон або рівнів. Взаємодія з такими об’єктами має бути інтуїтивною та стабільною, оскільки вона безпосередньо впливає на комфорт користувача та логіку проходження гри.

У розробленому ігровому застосунку реалізація артефактів побудована відповідно до принципів модульної архітектури та підходу Scriptable Object Architecture (SOA). Це дозволило відокремити дані від логіки взаємодії, забезпечити гнучке налаштування властивостей артефактів та можливість швидкого розширення їх функціональності без змін у базових скриптах. Наприклад, кількість отриманого золота, параметри лікування або цільовий рівень порталу визначаються не кодом, а конфігураційними об’єктами, що значно спрощує оновлення ігрового світу.

На рисунках 3.28 – 3.30 зображено розроблені SOA конфіги, котрі зберігають дані. На рисунках 3.31 – 3.33 зображено елементи коду логіки артефактів ігрового світу.

```
[CreateAssetMenu(menuName = "SO/Healing Fountain Config", fileName = "New Healing Fountain Config")]
public class HealingFountainConfigSO : ScriptableObject
{
    [Header("Healing")]
    public int healingAmount = 1;

    [Header("Cooldown")]
    public float healCooldown = 1.0f;
}
```

Рисунок 3.28 – SOA-конфіг лікувального фонтану

```
[CreateAssetMenu(menuName = "SO/Portal Config", fileName = "New Portal Config")]
public class PortalConfigSO : ScriptableObject
{
    [Header("Scene")]
    [Tooltip("Index of the scene in Build Settings (SceneManager.LoadScene)")]
    public int sceneIndex = -1;

    [Header("Optional")]
    [Tooltip("Saving after using portal")]
    public bool saveStateBeforeLoad = true;

    [Tooltip("Text for entering the portal")]
    public string enterMessage = "";

    [Tooltip("Time for showing message")]
    public float messageDuration = 2f;
}
```

Рисунок 3.29 – SOA-конфіг порталів

```
[CreateAssetMenu(menuName = "SO/Chest Config", fileName = "New Chest Config")]
public class ChestConfigSO : ScriptableObject
{
    [Header("Visual")]
    public Sprite closedSprite;
    public Sprite emptySprite;

    [Header("Reward")]
    public int goldAmount = 5;

    [Header("Floating Text")]
    public string messageFormat = "+{0} gold!";
    public int fontSize = 25;
    public Color textColor = Color.yellow;
    public Vector3 textOffset = new Vector3(0, 50, 0);
    public float textDuration = 3.0f;
}
```

Рисунок 3.30 – SOA-конфіг сундуків із золотом

```
protected override void OnCollect()
{
    if (collected)
        return;

    collected = true;

    if (config == null)
    {
        Debug.LogWarning($"Chest '{name}' has no ChestConfigSO assigned!");
        return;
    }

    // Міняємо спрайт на пустий
    if (spriteRenderer == null)
        spriteRenderer = GetComponent<SpriteRenderer>();

    if (spriteRenderer != null && config.emptySprite != null)
    {
        spriteRenderer.sprite = config.emptySprite;
    }

    // Нараховуємо золото
    GameManager.instance.Gold += config.goldAmount;
```

Рисунок 3.31 – Елемент коду логіки сундуків із золотом

Зображений на рисунку 3.31 метод реалізує логіку взаємодії гравця зі скринями, які можна зібрати. На початку виконується перевірка, чи не була скриня вже зібрана раніше, що запобігає повторному нарахуванню нагороди. Якщо об’єкт ще не взаємодіяв із гравцем, встановлюється відповідний прапорець та перевіряється наявність конфігураційного ScriptableObject. За

його відсутності система формує попередження, що дозволяє розробнику швидко виявити помилки конфігурації під час розробки.

Після цього оновлюється візуальний стан скрині: за потреби отримується компонент SpriteRenderer, і якщо він існує та містить визначений у конфігурації пустий спрайт, зображення скрині змінюється на порожнє. Це створює ефект відкритої або вже зібраної скрині. Завдяки винесенню графічних параметрів у ScriptableObject, вигляд різних типів скринь може змінюватися без необхідності редагувати код.

На завершення виконання методу відбувається нарахування золота гравцеві: до глобального лічильника додається значення, визначене в конфігурації ChestConfigSO.

```
public class Portal : Collidable
{
    [Header("Config (SO)")]
    [SerializeField]
    private PortalConfigSO config;

    protected override void OnCollide(Collider2D coll)
    {
        if (coll.name != "Player")
            return;

        if (config == null)
        {
            Debug.LogError($"Portal '{name}' has no PortalConfigSO assigned.");
            return;
        }

        if (config.sceneIndex < 0)
        {
            Debug.LogError($"Portal '{name}' has invalid sceneIndex in config: {config.sceneIndex}");
            return;
        }
```

Рисунок 3.32 – Елемент коду для порталів

У цьому фрагменті коду реалізовано логіку роботи порталу – об’єкта, який дозволяє гравцю переходити між ігровими сценами. При зіткненні портал перевіряє, чи взаємодія відбувається саме з гравцем; у протилежному випадку подальша логіка не виконується. Далі здійснюється перевірка наявності конфігураційного ScriptableObject типу PortalConfigSO, що містить дані про індекс сцени, у яку має бути перенесений гравець. У разі відсутності конфігурації система формує критичне попередження, яке дозволяє швидко виявити помилки при налаштуванні об’єктів у Unity.

```
public class HealingFountain : Collidable
{
    [Header("Config (SO)")]
    [SerializeField]
    private HealingFountainConfigSO config;

    private float lastHeal;

    protected override void OnCollide(Collider2D coll)
    {
        if (coll.name != "Player")
            return;

        if (config == null)
        {
            Debug.LogWarning("HealingFountain: config is null, no healing applied.");
            return;
        }

        // перевіряємо кулдаун із SO
        if (Time.time - lastHeal > config.healCooldown)
        {
            lastHeal = Time.time;

            // лікуємо гравця на значення з SO
            GameManager.instance.player.Heal(config.healingAmount);
        }
    }
}
```

Рисунок 3.34 – Елемент коду для лікувальних фонтанів

У цьому класі реалізовано поведінку лікувального джерела, яке відновлює здоров'я гравця при зіткненні з ним. Об'єкт використовує ScriptableObject типу HealingFountainConfigSO, що містить усі змінні параметри – величину відновлення здоров'я та час перезарядки між повторними активаціями. Це дозволяє легко створювати різні типи лікувальних фонтанів, змінюючи лише конфігураційні дані без модифікації логіки.

Під час зіткнення об'єкт перевіряє, чи є колайдер гравцем. Якщо взаємодія відбувається не з гравцем або конфігурація не призначена, подальша обробка не виконується, що запобігає помилкам і забезпечує коректне налаштування об'єкта в Unity. Далі перевіряється час, що минув із моменту останнього лікування: лікування дозволяється лише тоді, коли поточний час перевищує значення cooldown, визначене у ScriptableObject. Якщо умова виконана, оновлюється час останнього лікування й гравцеві відновлюється здоров'я в обсязі, що також зберігається в конфігурації.

ВИСНОВКИ

У магістерській кваліфікаційній роботі було розв’язано актуальну науково-практичну задачу побудови модульної архітектури ігрового застосунку з використанням підходу Scriptable Object Architecture (SOA). У ході дослідження проведено комплексний аналіз проблеми, розроблено архітектурне рішення та створено демонстраційний ігровий застосунок для перевірки працездатності запропонованої моделі. Основні результати виконаної роботи можна сформулювати так:

– проведено аналіз предметної області, визначено специфіку архітектурних вимог до сучасних ігрових застосунків та окреслено проблеми традиційних монолітних структур (висока зв’язаність, складність масштабування, дублювання логіки);

– виконано огляд сучасних архітектурних шаблонів, що застосовуються у геймдев-індустрії, зокрема MVC, MVVM, ECS, Event-Driven Architecture, Clean Architecture та Scriptable Object Architecture. Показано їхні переваги, недоліки та сфери застосування;

– обґрунтовано вибір Unity як середовища розробки, враховуючи його підтримку модульності, ScriptableObjects, інструментів для створення подієво орієнтованих систем та можливостей для швидкого прототипування;

– поставлено задачу розробки архітектури ігрового застосунку, сформульовано вимоги до її гнучкості, масштабованості, стабільності, повторного використання компонентів та мінімізації зв’язності;

– розроблено функціональні моделі системи відповідно до методології IDEF0, включно з декомпозицією основних модулів: інтерфейсу користувача, ігрового світу та модуля дій персонажів. Виявлено ключові потоки управління, механізмів, входів і виходів;

– побудовано DFD-діаграму (Data Flow Diagram), що відображає логіку руху даних у системі: завантаження ресурсів, ініціалізацію ігрового світу,

взаємодію між підсистемами, обробку дій гравця та оновлення ігрового процесу;

– спроєктовано архітектуру з використанням Scriptable Object Architecture, що включає модулі управління станом, подієві канали, незалежні контейнери даних, конфігураційні SO-файли та механізми реактивної взаємодії;

– реалізовано демонстраційний ігровий застосунок жанру Top-Down RPG, у якому впроваджено розроблену модульну архітектуру та перевірено її практичну ефективність;

Результатом виконаної роботи є розроблена та впроваджена модульна архітектура ігрового застосунку на основі Scriptable Object Architecture, що демонструє ефективність цього підходу при створенні гнучких, масштабованих та керованих ігрових систем. Створений демонстраційний застосунок підтверджує практичну коректність та доцільність використання SOA у сучасній розробці 2D-ігор.

Отримані результати можуть бути використані для подальших досліджень у галузі архітектур програмних систем, створення складніших ігрових проєктів, побудови редакторських інструментів для Unity та розширення можливостей модульного підходу в інтерактивних інформаційних системах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Мартін Р. Чиста архітектура: Посібник майстра зі структури та дизайну програмного забезпечення. Pearson Education, Limited, 2017.

2. Мартін Р. Чистий код: Довідник з майстерності Agile-програмування. Pearson Education Canada, 2008.

3. Фаулер М. Шаблони архітектури корпоративних застосунків. Addison-Wesley Professional, 2002. 560 с.

4. Шаблони ігрового програмування. Дженевер Беннінг, 2014. 354 с.

5. Грегорі Дж. Архітектура ігрового движка, третє видання. A K Peters/CRC Press, 2018. URL: https://doi.org/10.1201/9781315267845 (дата звернення: 02.12.2025).

6. Unity in Action, Third Edition: Multiplatform Game Development in C#. Manning Publications Co. LLC, 2022.

7. Вивчення C# шляхом розробки ігор за допомогою Unity: Ознайомтеся з кодуванням на C# та створюйте прості 3D-ігри в Unity 2022 з нуля, 7-е видання. Packt Publishing, Limited, 2022.

8. Мохмед М. Розробка ігор за допомогою C# та Unity: Unity та C# – для початківців. Незалежне видання, 2019.

9. Troelsen A., Japikse P. Pro C# 10 with .NET 6. Berkeley, CA : Apress, 2022. URL: https://doi.org/10.1007/978-1-4842-7869-7 (дата звернення: 02.12.2025).

10. Побудова керованої подіями мережі даних: шаблони для проектування та побудови керованих подіями архітектур. O'Reilly Media, Incorporated, 2023.

11. Шаблони – програми WPF із шаблоном проектування Model-View-ViewModel. Microsoft Learn: Розвивайте навички, які відкривають двері у вашій кар’єрі. URL-адреса: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern (дата звернення: 02.12.2025).

12. Деспейн В. Професійні методи написання текстів для відеоігор / ред. В. Деспейн. CRC Press, 2020.

13. Unity – API сценаріїв: ScriptableObject. Unity - Посібник: Посібник користувача Unity 6.2. URL: https://docs.unity3d.com/6000.2/Documentation/ScriptReference/ScriptableObject.html (дата звернення: 02.12.2025).

14. PatientZero. Unity: знайомство зі Scriptable Objects. *Хабр*. URL: https://habr.com/ru/articles/421523/ (дата звернення: 02.12.2025).

15. Атанасов Ю. Ігрова архітектура з використанням сценарійних об'єктів Частина 1. HypnoBeaverMoose. URL: https://hypnobeavermoose.github.io/2024/12/02/game-architecture-using-scriptable-objects-part-1.html (дата звернення: 02.12.2025).

16. Максим З. Діаграма послідовності (Sequence Diagrams). *Maxym Zosym*. URL: https://www.maxzosim.com/sequence-diagrams/ (дата звернення: 19.11.2025).

17. Лекція 6. Нотація IDEF0. Головна | ELib LNTU. URL: https://elib.lntu.edu.ua/sites/default/files/elib_upload/%D0%9A%D0%BE%D0%BD%D0%B4%D1%96%D1%83%D1%81%202%20%D0%B3%D0%BE%D1%82%D0%BE%D0%B2%D0%B2%D0%B0/page9.html (дата звернення: 19.11.2025).

18. Презентація "Діаграми UML. Діаграми прецедентів". Освітній проект «На Урок» для вчителів. URL: https://naurok.com.ua/prezentaciya-diagrami-uml-diagrami-precedentiv-238715.html (дата звернення: 19.11.2025).

19. Мілінгтон Я. Штучний інтелект для ігор. Morgan Kaufmann, 2006. 856 с.

20. Моралес Дж. Get to know everything about the DFD. MindOnMap | Free Mind Mapping Tool to Draw Ideas Easily Online. URL: https://www.mindonmap.com/ru/blog/data-flow-diagram/ (дата звернення: 19.11.2025).

21. Торн А. Основи анімації в Unity. Print2print, 2017. 176 с.

22. Дрескін Дж. Практичний гайд по маркетингу інді ігор. Routledge, 2015. URL: https://doi.org/10.4324/9781315754901 (дата звернення: 19.11.2025).

23. Бонд Дж. Unity и C#. Геймдев від ідеї до реалізації. 2022. 928 с.

24. Халперн Дж. Розробка 2D ігор з Юніті. Apress, 2019. 254 с.

25. Солодкий Д. В. Розробка комп'ютерної гри жанру "Top-Down RPG" для ОС Windows/ 28-й Міжнародний молодіжний форум "Радіоелектроніка та молодь у XXI столітті". Зб. матеріалів форуму. Т.6., - Харків: ХНУРЕ. 2024. -с 880-881