ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка кооперативної гри в жанрі економічної стратегії»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олександр ЦУМАН
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Олександр ЦУМАН

Керівник:. Олесь ДІБРІВНИЙ
доктор філософії(PhD)

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Цуману Олександру Сергійовичую

1. Тема кваліфікаційної роботи: «Розробка кооперативної гри в жанрі економічної стратегії»,

Керівник кваліфікаційної роботи доктор філософії (PhD) Олесь ДІБРІВНИЙ, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: опис процесу створення мультиплеєрної покрокової економічної гри на рушії Unity, аналіз жанрових аналогів (Monopoly, Pummel Party) і офіційної документації Unity 2020.3 LTS, Photon PUN 2 SDK та Firebase Unity SDK, формування функціональних і нефункціональних вимог, проектування архітектури з використанням патернів MVP та Dependency Injection, розробка UML-діаграм Use Case і класів, а також план розробки та тестування із застосуванням мови програмування C# і середовища Unity.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Теоретичний огляд і порівняльний аналіз жанрових аналогів покрокових економічних ігор.

2. Визначення функціональних та нефункціональних вимог до мультиплеєрної кооперативної гри.
3. Проектування архітектури системи: UML-діаграми (Use Case, класів), геймплейний цикл, схеми авторизації з PBKDF2.
4. Реалізація клієнтської частини гри на рушії Unity із використанням C#, Photon PUN 2 і Firestore.
5. Інтеграція UI-компонентів (TextMesh Pro, Cinemachine) та налаштувань.
6. Проведення тестування (функціональне, продуктивне, аналіз результатів та доопрацювання механік.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів
2. Вимоги до додатку (функціональні та нефункціональні)
3. Концепт гри
4. Геймплей та механіки
5. Програмні засоби реалізації
6. UML-діаграми (Use Case, класів)
7. Екранні форми
8. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення науково-технічної літератури, аналіз жанрових аналогів	25.02.-10.03.2025	
2	Формулювання вимог гри, складання технічного завдання	11.03-25.03.2025	
3	Проектування архітектури: UML-діаграми, геймплейний цикл, схема авторизації	26.03-10.04.2025	
4	Реалізація клієнтської частини гри (C#, Unity): механіки, інтерфейс	11.04-30.04.2025	
5	Інтеграція мережевих сервісів (Photon PUN 2) та бази даних Firestore	01.05-03.05.2025	
6	Проведення тестування (функціональне, продуктивне), виправлення виявлених помилок	04.05-06.05.2025	

7	Написання розділів 3–5, оформлення розділу тестування	07.05-14.05.2025	
8	Підготовка презентації, остаточне редагування та подання роботи	15.05-18.05.2025	
9	Попередній захист роботи	19.05-23.05.2025	

Здобувач вищої освіти _____ Олександр ЦУМАН

(підпис)

Керівник

кваліфікаційної роботи _____ Олесь ДІБРІВНИЙ

(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра:70 стор., 2 табл., 36 рис., 20 джерел .

Мета роботи – розробка інтерактивної мультиплеєрної покрокової економічної гри на рушії Unity, яка через моделювання реальних економічних механізмів у гейміфікованому середовищі підвищує рівень фінансової грамотності користувачів .

Об'єкт дослідження – процес взаємодії гравців у покрокових економічних стратегіях із використанням реальних механізмів кредитування та інвестицій.

Предмет дослідження – програмне забезпечення для реалізації клієнтської частини гри на рушії Unity із підтримкою мережевої взаємодії (Photon PUN 2) та збереження даних у Firebase Firestore .

Короткий зміст роботи:

1. Аналіз жанрових аналогів (Monopoly, Pummel Party, Cashflow, Tabletop Simulator): порівняння механік, глибині фінмеханік і навчального потенціалу конкурентів .

2. Проектування архітектури: вибір патернів MVP, Dependency Injection та Service Locator; поетапна ініціалізація через глобальний і локальний Bootstrap.

3. Реалізація клієнтської частини на Unity з C# – інтеграція TextMesh Pro, Cinemachine, Photon PUN 2 та Firestore для синхронізації стану гри і збереження статистики .

4. Тестування розробленого продукту: функціональне та продуктивне випробування ключових сценаріїв (реєстрація/вхід, створення/приєднання до сесії, кидок кубика, мережна синхронізація, налаштування фішок, вихід); підтверджено затримки ≤ 100 мс та коректність збереження даних у Firestore.

5. Висновки та рекомендації: проаналізовано досягнення цілей, виявлено напрямки оптимізації мережевих викликів та обробки помилок для підвищення продуктивності та зручності користування.

Сфера використання: освітні та тренінгові ігрові платформи для підвищення фінансової грамотності.

КЛЮЧОВІ СЛОВА: ІНТЕРАКТИВНА МУЛЬТИПЛЕЄРНА ЕКОНОМІЧНА ГРА, UNITY, C#, PHOTON PUN 2, FIREBASE FIRESTORE, MVP, DEPENDENCY INJECTION.

ЗМІСТ

ВСТУП

Актуальність: сучасні цифрові ігри демонструють високий потенціал для формування навичок через гейміфікацію та інтерактивне моделювання складних процесів, зокрема економічних, що сприяє глибшому засвоєнню фінансових понять і розвитку критичного мислення. Розробка інтерактивної мультиплеєрної покрокової економічної гри на рушії Unity з використанням хмарного сервісу Photon PUN 2 та бази Firebase Firestore дозволяє поєднати в одному продукті платформу для моделювання реальних фінансових механізмів, масштабовану мережеву взаємодію та захищену авторизацію користувачів. Обрана модульна архітектура з глобальним і локальним Bootstrap-системами та патернами Dependency Injection і Service Locator забезпечує послідовну ініціалізацію підсистем, гнучкість розширення й стабільність роботи навіть за високоінтенсивного мережевого навантаження.

Мета роботи – підвищити рівень фінансової грамотності користувачів шляхом моделювання реальних економічних механізмів у гейміфікованому кооперативному середовищі на рушії Unity.

Об'єкт дослідження – процес взаємодії гравців і механізми прийняття стратегічних економічних рішень у мультиплеєрних покрокових іграх.

Предмет дослідження – розробка покрокової кооперативної економічної стратегії на Unity, що симулює реальні економічні механізми.

Методи дослідження: здійснено аналіз наукової й технічної літератури та огляд жанрових аналогів для формулювання вимог; змодельовано систему ініціалізації та архітектуру гри за допомогою UML-діаграм; розроблено клієнтську частину на Unity із інтеграцією TextMesh Pro, Cinemachine, Photon PUN 2 і Firestore; проведено функціональне, продуктивне та юзабіліті-тестування для оцінки швидкодії, стабільності мережевих викликів і зручності інтерфейсу.

Наукова новизна роботи полягає в комплексній інтеграції рушія Unity із хмарними мережевими сервісами Photon PUN 2 і Firebase Firestore; запровадженні модульної клієнт-серверної архітектури з подвійною Bootstrap-системою; застосуванні патернів MVP, Dependency Injection та Service Locator для забезпечення гнучкості й тестованості; розробці захищеної системи авторизації з хешуванням паролів і офлайн-кешуванням даних.

Практична значущість результатів полягає в можливості використання створеного прототипу як навчального інструмента для гейміфікованих тренінгів із фінансової грамотності та швидкого масштабування на різні платформи завдяки крос-платформеності Unity і готовим рішенням хмарної інфраструктури.

1. АНАЛІЗ ПРОБЛЕМИ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Опис предметної галузі

Ігри є однією з найпоширеніших форм розваг та пізнавальної діяльності в цифровому середовищі. Вони поєднують інтерактивність, сюжетні елементи, аудіовізуальні ефекти та соціальний компонент, що забезпечує високий рівень залучення користувачів. Сучасні цифрові ігри охоплюють безліч жанрів — від аркад і пригодницьких симуляторів до складних стратегічних проектів із відкритим світом. Завдяки постійному технічному розвитку та розширенню можливостей рушіїв, ігрова індустрія здатна моделювати реалістичні сценарії, створювати інтерактивні навчальні середовища і задовольняти потреби різноманітної аудиторії.

Особливо цікавою з погляду навчання є категорія «серйозних ігор», які поєднують ігрові механіки із цілеспрямованими освітніми чи тренувальними завданнями. У таких проєктах гейміфікація використовується для формування практичних навичок у різних галузях: від медицини до фінансів. Користувачі отримують змогу виконувати певні дії в безпечному віртуальному середовищі, аналізувати результати і коригувати свою стратегію, що сприяє закріпленню знань і розвитку критичного мислення.

Покрокові економічні ігри як жанр поєднують елементи стратегічного планування та моделювання фінансових процесів у форматі по черзі виконуваних ходів. Кожен гравець має обмежений набір ресурсів і часу на розробку власної економічної стратегії: купівлю активів, інвестування в проєкти, залучення позик, укладання угод. Інтеграція реальних механізмів — криптовалюти, іпотечних кредитів, дивідендів — підвищує глибину симуляції і дозволяє користувачам розвивати практичні навички управління капіталом. Завдяки візуалізації наслідків кожного рішення та чіткому зворотному зв’язку такі ігри стають ефективним

інструментом підвищення фінансової грамотності у молоді та широкої аудиторії, яка звикла до інтерактивного навчання через ігровий інтерфейс.

1.2 Огляд програмних продуктів-конкурентів

1.2.1 Monopoly (класична настільна гра)

Monopoly — це культова настільна економічна гра, у якій 2–6 гравців по черзі пересувають фішки по квадратному полю (10×10 клітинок), моделюючи механіку купівлі-продажу нерухомості. На початку сесії кожному учаснику видається стартовий капітал у розмірі 1500 грошових одиниць, набір карток «Шанс» і «Казна» та фішка, що символізує його присутність на дошці. Кожен хід розпочинається з кидка двох шестигранних кубиків: за їхнім результатом гравець пересуває фішку й може придбати нерухомість на новій клітинці або сплатити оренду власникам, якщо ця площа вже належить іншому учаснику.

Механіка інвестицій розгортається навколо будівництва будинків і готелів на придбаних ділянках та залучення іпотечних позик, що дозволяє учасникам маневрувати фінансовими потоками й формувати стратегію нарощування доходів. У разі банкрутства гравець позбавляється всіх активів, які переходять до його кредиторів, а партія продовжується, поки не залишиться єдиний фінансово спроможний учасник або не вичерпається заздалегідь узгоджена кількість раундів.

Серед головних переваг Monopoly — інтуїтивно зрозуміла механіка купівлі-продажу активів та яскравий соціальний досвід «гра за одним столом», що стимулює взаємодію та переговори між гравцями. Водночас довгі партії, обмеження до шести учасників і необхідність вручну підраховувати орендні виплати та іпотечні умови знижують ефективність гри як інструмента для гейміфікованого навчання фінансовим концептам.

1.2.2 Pummel Party (цифровий аналог із аркадними міні-іграми)

Pummel Party поєднує покрокові економічні ходи на віртуальному полі з яскравими аркадними міні-іграми, пропонуючи гравцям динамічний та соціальний

досвід. Сесія розпочинається в лобі, де 2–8 учасників підключаються до онлайн-матчу, вибирають карту й узгоджують кількість ходів. В основній фазі гравці кидають віртуальні кубики, пересувають фішки по полю, купують “бази” та збирають ресурси для нарощування капіталу.

Через кожні 3–5 ходів відбувається міні-гра (стрілялки, головоломки, змагання на спритність), у якій переможець отримує бонусні монети або додаткові ходи, що додає невимушеної конкуренції та розбавляє стратегічні етапи. Партія завершується, коли хтось досягає поставленої цілі (наприклад, накопичення найбільшого капіталу) або проходить узгоджена кількість раундів.

Серед головних переваг Pummel Party — повна автоматизація всіх розрахунків і статистики, що знімає з гравців необхідність вести облік вручну, а також поєднання стратегічного планування з розважальними аркадними вставками, які урізноманітнюють ігровий процес. Водночас у грі використовується спрощена економічна модель без реалізації реальних фінансових інструментів, а вимоги до апаратної частини через насичену 3D-графіку та ефекти можуть стати перешкодою для менш потужних пристроїв.

1.2.3 Інші схожі рішення

Cashflow — навчальна фінансова гра, створена Робертом Кійосакі в 1996 році, яка має на меті моделювання реальних шляхів до фінансової незалежності. Гравці спочатку знаходяться в “Rat Race” — стартовій фазі з фіксованим доходом та витратами, де потрібно нарощувати активи (нерухомість, акції, бізнес), доки не накопичать достатньо пасивного доходу, щоб перейти у фазу “Fast Track” з вищими цілями та механіками для створення великого капіталу

Гра розрахована на 3–6 учасників для максимального ефекту навчання через групову динаміку; існує також безкоштовна онлайн-версія для зареєстрованих користувачів, яка нараховує понад 100 000 активних гравців щомісяця. Завдяки процедурі зміни ролей (різні професії з унікальними доходами й витратами) Cashflow розвиває у гравців навички аналізу бюджету, управління ризиками та прийняття інвестиційних рішень у безпечному ігровому середовищі.

Tabletop Simulator — це універсальна онлайн-платформа для відтворення будь-яких настільних ігор у мультиплеєрному фізичному «пісочниці». Розроблений Berserk Games, він дозволяє до 10 гравців одночасно грати на одній віртуальній дошці, імпортувати власні 3D-моделі та створювати сценарії за допомогою Lua-скриптів.

Одна з ключових інновацій Tabletop Simulator — повна інтеграція зі Steam Workshop: на сьогодні в категорії «Game» налічується понад 37 000 модифікацій, які включають як класичні настільні ігри, так і оригінальні проєкти користувачів. Це дає змогу не лише відтворювати відомі ігри, а й експериментувати з новими механіками без необхідності глибокої технічної підготовки. З точки зору аудиторії, за оцінками PlayTracker, Tabletop Simulator має приблизно 9,8 млн власників на Steam та 816 000 активних гравців на місяць. Середня кількість одночасних гравців протягом останнього місяця становила близько 5 100, із піковим значенням понад 9 900 користувачів одночасно.

Хоча Tabletop Simulator надає нескінченні можливості кастомізації та швидкий запуск будь-якої настільної гри, він вимагає ручного налаштування правил і механік («докрад» карток, встановлення таймерів тощо). Відсутність автоматичної обробки економічних розрахунків та необхідність створення модифікацій для кожного нового сценарію обмежують його придатність як готового інструмента для гейміфікованого навчання фінансовим концептам.

1.2.4 Порівняльна таблиця функціональних можливостей і недоліків

Таблиця 1.1

Аналіз конкурентів проєкта

Критерій	**Monopoly**	**Pummel Party**	**Cashflow**	**Tabletop Simulator**	**Grift Together (проєкт)**
Кількість гравців	2–6	2–8	2–6	довільна (залежить від модифікацій)	2–6
Тип сесії	офлайн, нескінченні раунди	онлайн, фіксовані раунди	офлайн/онлайн, раунди до цілі	офлайн/онлайн, залежить від гри	онлайн, нескінченні раунди
Автоматизація розрахунків	відсутня	повна	часткова	відсутня/часткова	повна
Глибина фінмеханік	базова	спрощена	середня	залежить від гри	розширена (кредити, біржі)
Навчальний потенціал	низький	середній	високий	середній	високий
Платформа	настільна	ПК (Steam)	настільна/онлайн	ПК (Steam, локальний сервер)	крос-платформова (Windows, macOS)
Навантаження на систему	мінімальне	високе	мінімальне	мінімальне/залежить	оптимізоване

1.3. Огляд IT-засобів і технологій для реалізації

1.3.1 Ігровий рушій Unity

Unity — це потужний та універсальний крос-платформений ігровий рушій, що поєднує в собі гнучкість середовища розробки, багатий набір готових рішень і масштабованість для проєктів будь-якого рівня складності. Він дозволяє

створювати ігри для Windows, macOS, Linux, мобільних пристроїв (iOS та Android) і веб-браузерів через WebGL, що робить покрокову економічну стратегію доступною для максимальної кількості користувачів без додаткових витрат на перенесення проєкту на інші платформи.

Інтерфейс редактора Unity поєднує візуальне компонування сцен, інспектор властивостей і потужний Asset Browser, що спрощує імпорт і керування ресурсами: від 2D-спрайтів та 3D-моделей до аудіо кліпів і анімацій. Особливо важливими для проєкту є TextMesh Pro — рішення для чіткого відображення тексту та підтримки різних мов, а також Cinemachine, яка надає готові плагіни для динамічного керування камерою під час ходів гравців.

Графічні можливості рушія реалізовані через Universal Render Pipeline (URP) та High Definition Render Pipeline (HDRP), які дозволяють налаштовувати освітлення, тіні та пост-процесинг відповідно до потреб проєкту. Система частинок та фізичні рушії PhysX (для 3D) і Box2D (для 2D) забезпечують реалістичну взаємодію об’єктів і динамічні ефекти під час фінансових операцій на полі — наприклад, анімацію купівлі нерухомості чи рух камери під час відображення подій.

Написання коду у Unity здійснюється на мові C#, яка поєднує потужну статичну типізацію, автоматичне збирання сміття та підтримку сучасних конструкцій .NET (LINQ, корутини, async/await). Це значно спрощує розробку мережевих механік: створення асинхронних запитів до Photon і Firestore, обробку подій і синхронізацію стану гри.

Однією з найбільших переваг Unity є його активна спільнота та екосистема: в Asset Store доступні тисячі готових плагінів, від інтерфейсних штук до складних AI-модулів; офіційна документація і форуми підтримують швидке вирішення будь-яких технічних питань. За допомогою Unity Cloud Build і Collaborate можна організувати безшовний процес командної роботи та безперервної інтеграції, що актуально навіть для невеликих команд, які працюють над бакалаврською роботою.

Завдяки поєднанню зручного редактора, розгалуженої екосистеми плагінів, розвинених графічних і фізичних рушіїв, а також потужної мовної підтримки Unity

створює міцну основу для реалізації мультиплеєрної покрокової економічної гри із фокусом на простоту розробки та глибину геймплею.

1.3.2 Мова програмування C#

C# є основною мовою програмування в середовищі Unity, завдяки чому забезпечує єдину основу для реалізації ігрової логіки, взаємодії з рушієм та зовнішніми сервісами. Ця статично типізована мова, розроблена з урахуванням простоти вивчення, поєднує в собі синтаксичні конструкції, близькі до Java та C++, із потужним інструментарієм .NET-платформи. За рахунок суворого контролю типів на етапі компіляції та вбудованого механізму збірки сміття C# забезпечує високу надійність і ефективність управління пам'яттю, що особливо критично для мультиплеєрних покрокових ігор із динамічними оновленнями стану.

Однією з ключових переваг C# є сувора статична типізація та безпека типів. Усі змінні й вирази мають бути явно оголошені з типовими анотаціями, що дозволяє виявляти помилки ще під час компіляції й істотно знижує ймовірність непередбачуваних збоїв у ході гри. Це особливо важливо для мультиплеєрних сесій, де синхронізація стану між клієнтом і сервером потребує чіткої відповідності структур даних.

C# — це повністю *об'єктно-орієнтована мова*, що надає розробникам зручні засоби для організації коду в класи та інтерфейси. Для нашої гри застосовуються архітектурні патерни Model–View–Presenter, Service Locator і Dependency Injection, які допомагають чітко розмежувати бізнес-логіку, UI та мережеві сервіси. Повторне використання компонентів і модульність прискорюють підтримку та розширення функціональності.

Ще однією важливою особливістю є *автоматичне керування пам'яттю через збір сміття (garbage collection).* Розробники звільнені від ручного виділення та звільнення ресурсів, що значно зменшує ймовірність витоків пам'яті й полегшує створення складних ігрових механік, таких як динамічне завантаження даних гравців із Firestore чи управління об'єктами гри під час багатокористувацьких сесій.

C# також має широку сумісність із платформами: реалізації .NET Framework та .NET Core працюють на Windows, macOS і Linux. У контексті Unity це означає, що один і той самий код можна використовувати для всіх цільових платформ без модифікацій. Багата стандартна бібліотека .NET забезпечує готові класи для роботи з мережею, колекціями, файлами, а також підтримує сучасні можливості — LINQ для обробки даних, async/await для асинхронних операцій і корутини Unity для управління таймерами та анімаціями.

Розробка на C# супроводжується високоякісними IDE — Microsoft Visual Studio та Visual Studio Code. Ці середовища надають автозавершення коду, інструменти налагодження, аналізатор коду та вбудовану підтримку Git, що значно підвищує продуктивність розробників. Активна спільнота C#-розробників Unity пропонує численні онлайн-ресурси, форуми, документацію та бібліотеки з відкритим кодом. Це створює сприятливі умови для швидкого вирішення проблем і обміну досвідом.

Завдяки універсальності C# можна створювати додатки різного типу — від веб- і мобільних додатків до настільного ПЗ та ігор. Вона також має сумісність з іншими .NET-мовами (Visual Basic.NET, F#), що дозволяє інтегрувати існуючі кодові бази або використовувати сторонні бібліотеки. Погоджуючись із фокусом на мультиплеєрній покроковій економічній стратегії, C# забезпечує необхідний баланс між простотою використання, продуктивністю та розширюваністю.

1.3.3 Порівняння Unity з іншими ігровими рушіями

З огляду на завдання розробки мультиплеєрної покрокової економічної стратегії, доцільно порівняти Unity із двома популярними альтернативами — Unreal Engine та Godot — щоб обґрунтування вибору рушія.

Unreal Engine, розроблений компанією Epic Games, є одним із найпотужніших рушіїв для створення відеоігор і інтерактивних симуляцій, для AAA-проектів. Основою рушія є мова C++, яка забезпечує максимальну продуктивність і низькорівневий контроль над ресурсами. Unreal Engine вирізняється такими ключовими можливостями:

1 High Fidelity Rendering: рушій підтримує передові технології візуалізації — від реалістичного глобального освітлення (Lumen) до високоякісних тіней і матеріалів (Nanite), що дозволяє створювати кінематографічні сцени із деталізацією на рівні кіноіндустрії.

2 Blueprint Visual Scripting: система візуального скриптингу, яка надає змогу дизайнерам і програмістам швидко прототипувати ігрову логіку без написання коду, що пришвидшує ітерації та експерименти з механіками.

3 Розвинена фізика та анімації: інтегровані рушії Chaos для симуляції руйнувань та Cloth для тканини, а також потужні інструменти анімації (Control Rig, Sequencer) для створення натуральних рухів персонажів і спецефектів.

4 Мультиплатформеність: підтримка десктопних (Windows, macOS), мобільних (iOS, Android), консолей (PlayStation, Xbox, Switch), що робить Unreal Engine універсальним вибором для великих студій із різноманітними цільовими платформами.

5 Інструменти для командної роботи: Unreal Engine інтегрується з системами версіонування (Perforce, Git), а функція Live Coding дозволяє вносити зміни в код і бачити результат миттєво без перезавантаження рушія.

6 Marketplace та екосистема: Epic Marketplace пропонує широкий вибір готових моделей, плагінів і шаблонів, а також безкоштовні щомісячні пакети, що підтримують розробників у створенні складних проєктів.

Unreal Engine застосовується такими відомими студіями, як Ubisoft, Epic Games, CD Projekt RED, для розробки ігор рівня S.T.A.L.K.E.R., Gears of War і Fortnite, де необхідна максимальна графічна якість, деталізація оточення й реалістична фізика.

Godot — це відкритий рушій із відкритим вихідним кодом, який відзначається легкою вагою та гнучкістю, що робить його привабливим вибором для інді-розробників і невеликих команд. Основні особливості Godot:

- Node-Scene архітектура: система вузлів та сцени забезпечує модульне компонування проєкту, де будь-який об’єкт гри можна представити як дерево

вузлів зі своїми властивостями та поведінкою. Це спрощує повторне використання компонентів і масштабування проєкту.

- GDScript та мульти-мовність: вбудована мова GDScript зі схожим на Python синтаксисом дозволяє швидко писати ітеративний код; за потреби можна використовувати C# або навіть C++ для критичних за продуктивністю модулів.
- 2D-орієнтованість: рушій має першокласну підтримку 2D-графіки, в тому числі вбудовані інструменти для тайлмапів, шейдерів і фізики, що робить його оптимальним для спрайтових ігор.
- Легка вага та крос-платформеність: Godot працює на Windows, macOS, Linux та мобільних платформах без великих системних вимог, що прискорює запуск і тестування на різних пристроях.
- Розширюваність і кастомізація: завдяки відкритому коду розробники можуть доповнювати рушій власними плагінами, змінювати ядро або вбудовані модулі відповідно до специфічних потреб проєкту.
- Спільнота та документація: хоча Godot має меншу спільноту, ніж Unity чи Unreal Engine, вона активно розвивається, а офіційна документація й форуми забезпечують підтримку для розв'язання технічних задач.

Godot ідеально підходить для прототипування та невеликих ігор із акцентом на 2D, де важливі швидкий цикл розробки та контроль над рушієм.

Висновок: У світлі наведеного порівняння можна зробити такі висновки щодо вибору рушія для нашого бакалаврського проекту:

1. *Unreal Engine* ідеально підходить для створення високобюджетних AAA-проектів з кінематографічною графікою, складними шейдерами та просунутими фізичними симуляціями, але його складність і вимоги до ресурсів роблять його надмірним для невеликої студентської команди.
2. *Godot Engine* відзначається легкою вагою, гнучкою node-scene архітектурою та відкритим кодом, що робить його відмінним вибором для прототипування та 2D-ігор; проте обмежена кількість готових мережевих рішень і відсутність хмарної інфраструктури значно ускладнюють розробку мультиплеєра.

3. *Unity* поєднує простий у використанні візуальний редактор із розвиненою екосистемою плагінів, потужним C#-скриптингом та підтримкою хмарних сервісів для мережі (Photon) і аналітики (Firestore). Завдяки цьому Unity забезпечує швидке прототипування та оперативне налаштування як UI-, так і мережевих механік, підтримує крос-платформеність та має інструменти для профілювання продуктивності.

Отже, з огляду на баланс між продуктивністю, гнучкістю розробки й готовими рішеннями для мультиплеєра, Unity є найбільш відповідним середовищем для реалізації мультиплеєрної покрокової економічної стратегії в рамках бакалаврського проєкту.

1.3.5 Photon для мережевої взаємодії

Photon Engine — це хмарне рішення для реалізації мультиплеєра в іграх на рушії Unity, що надає готову інфраструктуру та інструменти для синхронізації гравців у реальному часі. У рамках проєкту з покрокової економічної стратегії Photon було обрано як оптимальну мережеву бібліотеку завдяки простоті інтеграції, широкій спільноті та наявності готових інструментів для лобі.

Використання Photon дозволяє відмовитися від ручного налаштування серверної частини: достатньо інтегрувати офіційну Unity SDK, налаштувати параметри лобі (максимальна кількість гравців, приватність лобі), після чого Photon автоматично розподілить сесії між гравцями і забезпечить низькі затримки завдяки географічно розподіленим серверам.

Основою Photon є модель клієнт–сервер, у якій клієнти обмінюються даними через центральний сервер, що гарантує узгодженість стану гри. Механізми RPC (Remote Procedure Call) надає можливість виклику віддалених методів і передачі подій одним рядком коду — це значно спрощує реалізацію логіки передачі ходів, оновлення стану ігрового поля і синхронізацію економічних операцій у покроковому режимі.

Photon також містить вбудований матчмейкінг, а міні-консоль у редакторі Unity дозволяє тестувати підключення та моніторити трафік у реальному часі.

Завдяки таким можливостям розробникам не потрібно глибоко занурюватися в особливості мережевих протоколів і розгортати власні сервери — це економить час на налаштування інфраструктури та дозволяє зосередитися на геймплейній логіці.

1.3.6 Photon у порівнянні з Mirror

Mirror — це популярна open-source бібліотека для мережевої взаємодії в Unity, яка розвинулась із Unity UNet. Вона дозволяє розробникам створювати клієнт–серверні або peer-to-peer рішення без використання сторонніх хмарних сервісів. Основні характеристики Mirror:

- Open-source: код доступний на GitHub, що дає змогу адаптувати бібліотеку під власні потреби та усувати помилки самостійно.
- Локальний хостинг: Mirror не вимагає сторонніх серверів — можна розгорнути власний сервер у будь-якому середовищі, включно з віддаленим сервером чи локальною машиною.
- Простий API: Mirror оснований на класах з мережевої бібліотеки UNet. Звідки вона отримала ряд класів, таких, як NetworkBehaviour, NetworkManager і NetworkIdentity, що полегшує перехід для тих, хто вже працював із UNet.
- Безкоштовність: Mirror не має ліцензійних платежів — усі можливості доступні безкоштовно за умовами MIT-ліцензії.

Таблиця 1.2

Порівняння Photon та Mirror

Критерій	**Photon**	**Mirror**
Інфраструктура	Хмарна (географічно розподілені сервери)	Локальний або хмарний (на власному хості)
Вартість	Платна модель згідно з тарифами Photon	Безкоштовно (MIT-ліцензія)
Масштабованість	Автоматична, залежить від сервісу Photon	Обмежена потужністю власного сервера
Підтримка	Офіційна підтримка, документація, форуми	Спільнота GitHub, форуми Unity
API і простота	Інтуїтивні RPC та RaiseEvent	Звичний UNet-подібний API, але більше коду для реалізації специфічних сценаріїв
Надійність	Висока — сервера сервісу Photon	Залежить від якості налаштування сервера

Висновок: Mirror добре підходить для проєктів із обмеженим бюджетом або коли потрібен повний контроль над серверною інфраструктурою. Однак для бакалаврського проєкту мультиплеєрної покрокової стратегії Photon надає готові сервіси лобі та синхронізації, глобальне розміщення серверів і мінімальні налаштування. Це дозволяє зосередитися на розробці ігрової логіки, а не на адмініструванні мережевих серверів.

1.3.8 Firestore для зберігання даних

Google Firestore — це документоорієнтована база даних у складі платформи Firebase, розроблена для зберігання та синхронізації даних у реальному часу. У проєкті Firestore використовується для збереження облікових записів гравців та налаштувань ігрової сесії.

Firestore надає гнучку схему колекцій і документів, що дозволяє зберігати кожного гравця як окремий документ із довільним набором полів. Реальне оновлення (Realtime Listeners) забезпечує автоматичну передачу змін на клієнтські пристрої: наприклад, оновлення балансу чи статусу кімнати в лобі відображається миттєво без перезавантаження.

Для захисту даних Firestore використовує Security Rules, які дають змогу детально налаштувати права читання й запису залежно від ролі користувача та стану документа. Інтеграція з Firebase Authentication гарантує, що кожен запит походить від авторизованого гравця, а конфіденційні поля (соль, хеш) залишаються прихованими від інших користувачів.

Firestore включає в себе потужні інструменти аналітики, що дозволяють відстежувати поведінку користувачів і стан додатка без необхідності впровадження зовнішніх сервісів. Завдяки вбудованій статистиці Firebase можна отримувати базові показники (активні користувачі, частота звернень, помилки), а за допомогою власних подій — створювати власну аналітику. Це дозволяє, наприклад, відправляти на сервер користувацькі події з будь-якими параметрами (час, нікнейм, затримка з’єднання) та будувати за ними звіти у вигляді графіків чи таблиць у

консолі Firebase, що полегшує моніторинг продуктивності та прийняття рішень щодо оптимізації мережевих налаштувань.

Інтеграція Firestore зі стандартним Unity SDK зводиться до декількох рядків коду: створення посилання на колекцію, встановлення слухачів змін та виконання операцій. Це значно пришвидшує реалізацію серверної логіки, звільняючи від необхідності власної інфраструктури для збереження статистики. Також Firestore автоматично кешує останні дані локально, забезпечуючи офлайн-доступ і автоматичну синхронізацію після відновлення з'єднання, що важливо для мобільних гравців з нестабільним інтернетом.

1.3.9 Firestore у порівнянні з MySQL

На відміну від традиційних реляційних баз даних, таких як MySQL, Firestore пропонує документно орієнтовану модель, що забезпечує більшу гнучкість у роботі зі структурованими, але різнорідними даними. У MySQL доводиться заздалегідь проектувати табличну схему з чітко прописаними зв'язками між таблицями, що часто призводить до складних запитів і потребує додаткового адміністрування сервера баз даних.

Натомість Firestore використовує колекції та документи, де кожен документ може мати власну структуру полів без необхідності змінювати схему при додаванні нових параметрів. Це особливо корисно для додавання динамічних даних — наприклад, власної аналітики — без ризику порушити цілісність існуючих записів.

Крім того, Firestore пропонує вбудовані механізми офлайн-кешування та автоматичної синхронізації: мобільні клієнти можуть продовжувати роботу при втраті з'єднання, а після відновлення всі зміни згладжуються і відправляються на сервер. У MySQL для подібного функціоналу потрібні сторонні рішення та додатковий код.

З точки зору масштабованості, Firestore автоматично розподіляє навантаження між серверами Google, що дозволяє безперебійно обслуговувати тисячі одночасних з'єднань. У MySQL горизонтальне масштабування потребує налаштування власних розподільних механізмів, що значно ускладнює адміністрування.

Отже, Firestore є кращим варіантом для збереження ігрових даних мультиплеєрної покрокової стратегії, оскільки забезпечує гнучкість у структурі

даних, вбудовані засоби офлайн-доступу та просте автоматичне масштабування без необхідності підтримувати складну інфраструктуру бази даних.

1.4 Визначення об’єкта, предмета, мети та завдань

Об’єкт дослідження. Процес взаємодії гравців у покроковій мультиплеєрній економічній стратегії з використанням реальних фінансових механізмів (кредити, криптовалюта, інвестиції).

Предмет дослідження. Програмне забезпечення для реалізації клієнтської частини гри на рушії Unity з інтеграцією мережевих та аналітичних сервісів (Photon, Firestore) для синхронізації ігрових сесій та збирання статистики.

Мета роботи. Розробити мультиплеєрну покрокову економічну гру на рушії Unity, яка через моделювання реальних економічних механізмів у гейміфікованому кооперативному середовищі підвищує рівень фінансової грамотності користувачів.

Завдання роботи:

1. Проаналізувати жанрові аналоги і визначити їхні переваги та недоліки у контексті навчання фінансовим механізмам.
2. Визначити функціональні та нефункціональні вимоги до проєкту.
3. Обґрунтувати вибір архітектурного підходу та ключових технологій (Unity, C#, Photon, Firestore).
4. Спроєктувати архітектуру гри, UML-діаграми та макети інтерфейсу.
5. Реалізувати клієнтську частину гри з можливістю мережевих сесій і збереження статистики.
6. Налаштувати механізми безпечної авторизації користувачів та обробку мережевих подій.
7. Провести юзабіліті-тестування з реальними гравцями і профілююче тестування продуктивності (FPS, пінг).
8. Проаналізувати отримані результати, надати рекомендації щодо оптимізації та розвитку проєкту.

2. АНАЛІЗ ПРОБЛЕМИ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

2.1 Загальна архітектура та система ініціалізації

У даному проєкті застосовано модульний архітектурний підхід, за якого код розділений на окремі, чітко ізольовані модулі з власними зонами відповідальності, та організовану за допомогою bootstrap system — системи поетапної ініціалізації.

Модульна система означає, що кожна функціональна складова програми (мережевий шар, інтерфейс, аналітика тощо) упакована в окремий модуль із чітко визначеним інтерфейсом взаємодії з іншими модулями.

Ключові ознаки модульної системи:

1. Чітке розділення обов'язків (singleton solid principles). Кожен модуль відповідає лише за свою задачу — наприклад, обробку мережевих запитів або відображення інтерфейсу.

2. Слабке зв'язування (Loosely Coupled). Модулі взаємодіють через абстракції (інтерфейси), а не напряму, що спрощує їх заміну або оновлення.

3. Висока внутрішня згуртованість (High Cohesion). Компоненти всередині модуля тісно пов'язані за своєю функцією, але не "витікають" за межі доменної області.

4. Повторне використання (Reusability). Модулі легко переносити в інші проєкти або використовувати в різних сценах без суттєвих змін.

5. Тестованість. Завдяки чітким кордонам та інтерфейсам, будь-який модуль можна окремо покрити юніт-тестами або замінити мок-реалізаціями.

6. Незалежне розгортання та оновлення. Додавання нових функцій або виправлення багів у межах одного модуля не вимагає змін у всій системі.

Bootstrap System — це механізм поетапної ієрархічної ініціалізації модулів, що забезпечує:

1. Гарантована послідовність. Кожен наступний рівень ініціалізації очікує на готовність попереднього — це виключає ситуації, коли сервіс звертається до ще не створеного компонента.

2. Модульність. Додавання або видалення окремих сервісів не вимагає зміни логіки старту — досить зареєструвати їх у контейнері DI.

3. Контроль залежностей. Через DI-контейнер кожен модуль отримує строго визначений набір сервісів, позбавляючи від жорстких посилань на конкретні класи.

Нижче наведено, як ця система вбудована в проєкт:

1. Глобальна фаза

Перша точка входу відповідає за створення контейнера залежностей (IoC), реєстрацію й запуск глобальних сервісів (мережевих, аналітичних, навігаційних).

2. Локальна фаза

Друга точка входу в кожній сцені перевіряє готовність глобальних сервісів і ініціює локальні менеджери, відповідальні за функціональність поточної сцени.

3. UI-модулі

Після запуску менеджерів кожен UI-модуль (View + Presenter) отримує через контейнер лише ті залежності, що йому потрібні, і починає роботу.

Такий підхід виключає гонки ініціалізації, спрощує додавання нових модулів і гарантує, що кожен компонент стартує в належному порядку.

Нижче показано, як Bootstrap System реалізоване в проєкті:

- GameEntryPoint, показано на рисунку 2.1 Глобальний Bootstrap

1. Ініціалізує DI-контейнер і реєструє глобальні сервіси (Localization, Sound і т.д.).

2. Запускає глобальні менеджери (Global Managers) у визначеному порядку — кожен менеджер при створенні отримує доступ до вже ініціалізованих сервісів.

3. Завантажує стартову сцену (Login або Menu) після того, як усі ключові сервіси готові до роботи.

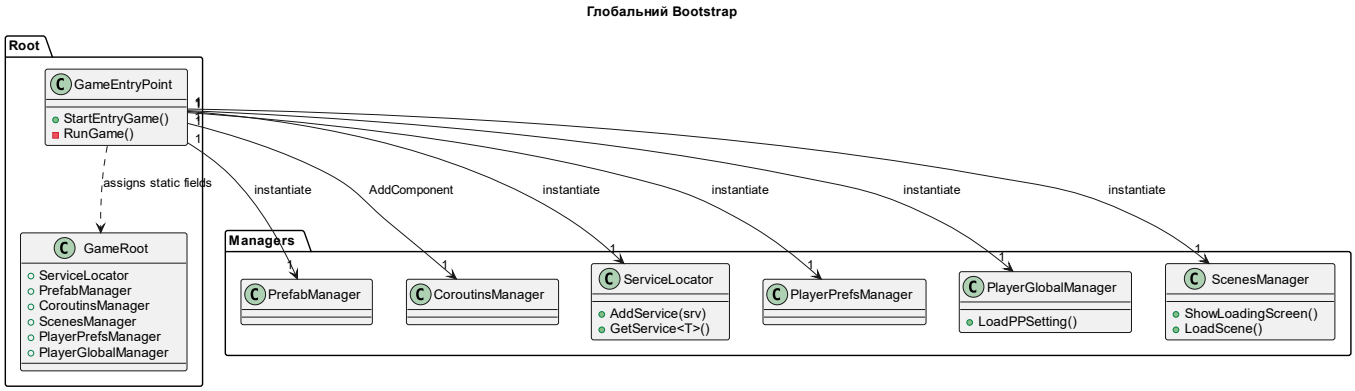


Рис. 2.1 Глобальний Bootstrap

- SceneEntryPoint, представлено на рисинок 2.2 Локальний Bootstrap)

1. Перевіряє, що глобальні сервіси ініціалізовані та доступні.

2. Реєструє та запускає локальні менеджери сцени (Scene Managers), кожен із яких відповідає за свою підсистему (наприклад, LobbyManager, GameManager).

3. Передає кожному локальному менеджеру ті глобальні сервіси, з якими він працює, гарантуючи послідовність ієрархії ініціалізації.

- UI-модулі та компоненти, рисунок 2.2 Локальний Bootstrap

1. Кожен локальний менеджер ініціює певні UI-модулі (View + Presenter) для реалізації екранних функцій.

2. Менеджер викликає відповідний View, а Presenter отримує через DI-контейнер усі необхідні залежності (мережа, аналітика, модель даних).

3. UI-модулі працюють незалежно, але завжди можуть звернутися до глобальних і локальних менеджерів через загальну систему сервісів.

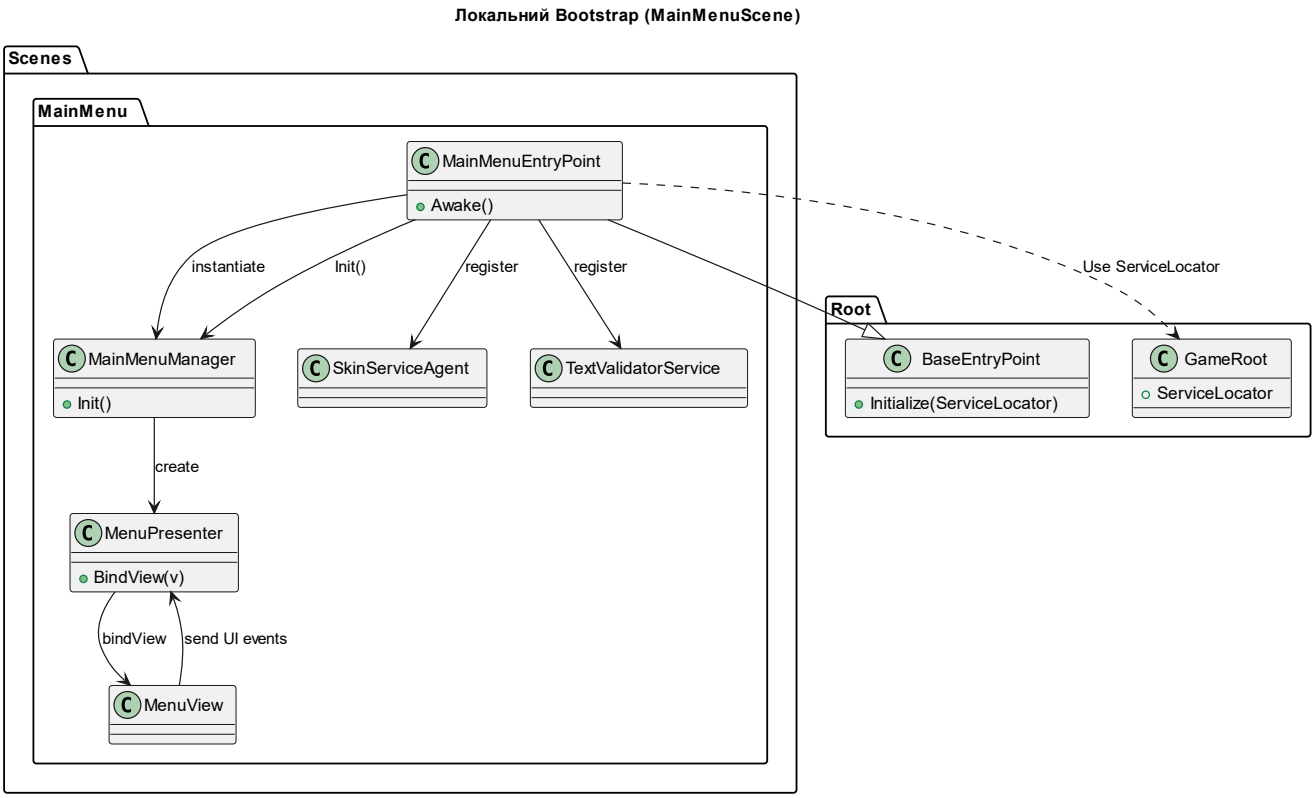


Рис. 2.2 Локальний Bootstrap

Такий підхід виключає гонки ініціалізації, спрощує додавання нових модулів і гарантує, що кожен компонент стартує в належному порядку.

2.2 Патерн єдиної точки входу (EntryPoint)

Патерн єдиної точки входу передбачає наявність у кожному контексті виконання (старт програми чи завантаження нової сцени) одного класу–ініціатора. Цей клас відповідає за послідовну ініціалізацію всіх необхідних підсистем: налаштування системи залежностей, реєстрацію сервісів та запуск базових менеджерів.

Головні переваги застосування EntryPoint:

- Контроль порядку запуску. Забезпечується гарантована послідовність ініціалізації, що виключає звернення до неготових компонентів.

- Централізація логіки старту. Весь код, пов’язаний із підготовкою середовища та першим запуском сервісів, зосереджено в одному місці, спрощуючи підтримку та розширення.
- Модульність та розширюваність. Додавання нових підсистем або налаштування існуючих зводиться до внесення змін лише в клас EntryPoint без необхідності правок у інших модулях.

Нижче наведено приклад реалізації EntryPoint на рівні сцени, що ілюструє його роль у структурі проєкту.

Приклад реалізації на базі BaseEntryPoint та MainMenuEntryPoint.

BaseEntryPoint — абстрактний клас, який задає шаблон ініціалізації для всіх сцен, подано на рисунку 2.3 BaseEntryPoint і продемонстрована його реалізація на рисунку 2.4 MainMenuEntryPoint:

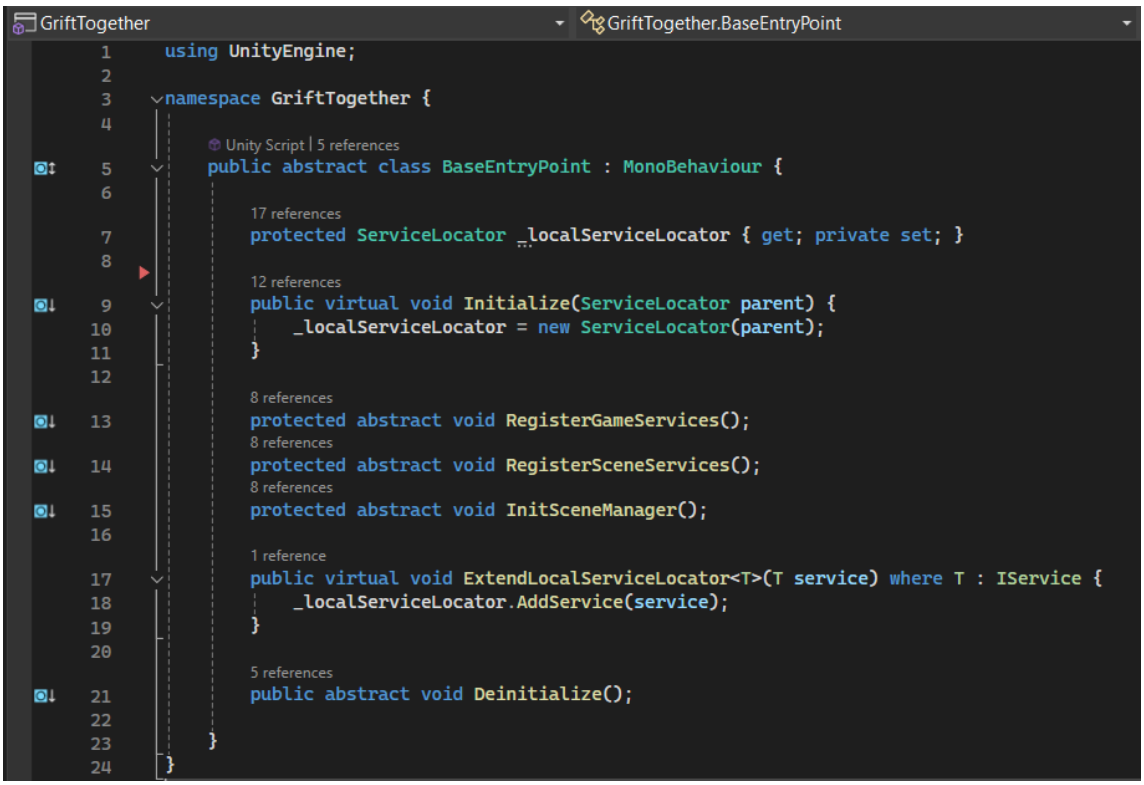


Рис. 2.3 BaseEntryPoint

GriftTogether — GriftTogether.MainMenuEntryPoint

```
using UnityEngine;

namespace GriftTogether {

    Unity Script (1 asset reference) | 0 references
    public class MainMenuEntryPoint : BaseEntryPoint {

        [SerializeField] private GameObject _mainCanvas;
        [SerializeField] private GameObject _overlayCanvas;
        [SerializeField] protected SkinServiceAgent _skinServiceAgent;

        private MainMenuManager _menuManager;

        Unity Message | 0 references
        private void Awake() {
            Initialize(GameRoot.ServiceLocator);
        }

        6 references
        public override void Initialize(ServiceLocator parent) {
            base.Initialize(parent);
            RegisterGameServices();
            RegisterSceneServices();
            InitSceneManager();
        }

        2 references
        protected override void RegisterGameServices() {

            if(GameRoot.ServiceLocator.Resolve(out SkinPhotonService service) == false) {
                service = new SkinPhotonService();
                GameRoot.ServiceLocator.AddService(service);
            }

            if (GameRoot.ServiceLocator.Resolve(out TextValidatorService textValidatorService) == false) {
                TextValidatorService textValidatorService1 = new TextValidatorService();
                GameRoot.ServiceLocator.AddService(textValidatorService1);
            }
        }

        2 references
        protected override void RegisterSceneServices() {
            if (_localServiceLocator.Resolve(out SkinService skinService) == false) {
                skinService = new SkinService();
                _localServiceLocator.AddService(skinService);
            }

            skinService.SetSkinAgent(_skinServiceAgent);
            skinService.ResolveCurrentSkin();
        }

        2 references
        protected override void InitSceneManager() {
            _menuManager = new MainMenuManager(_mainCanvas, _overlayCanvas, _localServiceLocator);
            _menuManager.Initialize();
        }

        2 references
        public override void Deinitialize() {
            _menuManager.Deinitialize();
        }
    }
}
```

Рис. 2.4 MainMenuEntryPoint

1. Awake() у BaseEntryPoint автоматично спрацьовує при завантаженні сцени й отримує ServiceLocator.

2. RegisterSceneServices() дозволяє підкласом додати лише ті сервіси, які потрібні на цій сцені.

3. InitSceneManagers() створює локальні менеджери, передаючи їм через DI вже зареєстровані сервіси.

Таким чином, патерн єдиної точки входу забезпечує чіткий, послідовний і контрольований запуск усіх підсистем у будь-якій сцені, спрощує розширення та підтримку коду. Створена ієрархія BaseEntryPoint та MainMenuEntryPoint є наочним прикладом застосування цього підходу.

2.3 Патерн Dependency Injection (DI)

2.3.1 Базовий Dependency Injection

Патерн Dependency Injection передбачає, що об’єкт отримує всі свої залежності (служби, репозиторії, менеджери) не через власні виклики конструктора чи статичні фабрики, а «зверху» — через механізм інверсії керування. У проєкті це реалізовано через DI-контейнер, який у момент старту гри реєструє всі сервіси та потім автоматично передає їх у конструктори або властивості тих класів, які їх потребують.

Основні переваги застосування DI:

- Слабке зв’язування**.** Класи працюють з інтерфейсами, а не з конкретними реалізаціями, що дозволяє змінювати сервіс без модифікації споживача.
- Централізована конфігурація**.** Усі залежності налаштовуються в одному місці — у стартовому EntryPoint — замість того, щоб розкидувати створення об’єктів по всьому коду.
- Покращена тестованість. У юніт-тестах достатньо зареєструвати мок-об’єкти в контейнері, щоб ізолювати логіку тестованого класу.
- Модульність і повторне використання. Модулі, які не створюють власні залежності, можна легко переносити між проєктами та сцена ми без ризику циклічних викликів або дублікатів коду.

Завдяки DI-контейнеру архітектура лишається гнучкою, а налаштування сервісів — зрозумілим і централізованим.

2.3.2 Патерн Service Locator

Service Locator — архітектурний патерн, який централізує доступ до набору сервісів через єдину «точку входу». Замість того, щоб кожен клас самостійно створював або знаходив потрібну реалізацію, він звертається до об’єкта-локатора, викликаючи метод типу GetService<T>(). Локатор утримує словник «інтерфейс - екземпляр» і повертає зареєстровану реалізацію на запит.

Історія та еволюція:

1. 2003 рік: Мартін Фаулер у книзі *Patterns of Enterprise Application Architecture* формалізує Service Locator як спосіб реалізації інверсії керування в корпоративних застосунках.

2. 2000-ті: Патерн набуває популярності в середовищах .NET та Java, де ранні DI-контейнери були ще недостатньо зрілими або потребували складної конфігурації.

3. Проміжний крок: Service Locator часто використовували як перший етап впровадження IoC, а згодом поступово перейшли до повноцінних DI-фреймворків (наприклад, Spring для Java, Microsoft.Extensions.DependencyInjection для .NET).

4. Сучасне застосування: хоча багато команд віддає перевагу DI-контейнерам з автоматичною ін’єкцією, Service Locator все ще зустрічається у проєктах зі статичними обмеженнями (MonoBehaviour у Unity, статичні класи) або як адаптація існуючих кодових баз.

Ключові характеристики Service Locator:

1. Єдина точка реєстрації. Усі сервіси додаються в локатор під час старту застосунку або сцени.

2. Єдиний API доступу. У клієнтам достатньо викликати один метод (GetService<T>()), щоб отримати будь-яку реалізацію.

3. Гнучка заміна реалізацій. Заміна справжніх сервісів на мок-версії у тестах або на нові реалізації в продакшні відбувається шляхом перереєстрації в локаторі.

4. Статичний чи Singleton-доступ. Зазвичай Service Locator реалізують як статичний клас або Singleton, щоб його можна було дістати з будь-якої точки коду.

2.3.3 Service Locator як різновид DI

Традиційне введення залежностей через конструктори (Constructor Injection) вважається правильнішою формою DI, Service Locator також відповідає базовим принципам інверсії керування:

1. Винос створення залежностей за межі клієнтів

Класи-споживачі не створюють і не шукають самостійно об’єкти-сервіси — вони звертаються до локатора і отримують готові екземпляри.

2. Централізована конфігурація

Усі відповідності між інтерфейсами та їхніми реалізаціями налаштовуються в одному місці (рядках реєстрації в Service Locator), як і в DI-контейнері.

3. Гнучкість заміни реалізацій

Для тестування чи зміни поведінки системи достатньо змінити реєстрацію в Service Locator, без правок у коді споживачів.

4. Чітка інверсія контролю (IoC)

Замість того щоб керувати своїми залежностями, клас делегує це Service Locator, який здійснює інверсію контролю за створенням та управлінням життєвим циклом сервісів.

Таким чином, Service Locator виконує роль легкого DI-механізму, зберігаючи слабке зв’язування та централізовані налаштування, але з явним запитом залежності (ServiceLocator.Get<T>), а не через автоматичну ін’єкцію конструкторами.

2.3.4 Роль Service Locator у проєкті

У проєкті реалізовано два рівні Service Locator: глобальний та локальний.

- Глобальний Service Locator існує в класі GameRoot і живе протягом усього циклу роботи гри. Саме в ньому реєструються базові сервіси — мережевий зв’язок, аналітика, менеджер сцен і т.і. У будь-якій точці коду можна отримати доступ до цих сервісів викликом GameRoot.ServiceLocator.Resolvet<Інтерфейс>().
- Локальний Service Locator ініціюється в кожній сцені через її SceneEntryPoint. Він наслідує глобальні сервіси і дозволяє додавати лише ті

залежності, які потрібні для роботи саме цієї сцени (наприклад, валідація форми логіну або скіни головного меню).

*Висновок***:** Такий двоступеневий підхід до організації Service Locator забезпечує суворе дотримання принципів модульності та інверсії керування в архітектурі проєкту. Глобальний локатор у GameRoot гарантує наявність єдиного, незмінного набору основних сервісів протягом усього життєвого циклу гри, водночас локальні локатори, ініційовані в межах кожної сцени, обмежують область дії специфічних залежностей лише необхідними компонентами. Це дозволяє уникнути зайвого завантаження загального простору і забезпечує точне управління життєвим циклом сервісів: кожна сцена отримує лише ті сервіси, які їй дійсно потрібні, а після виходу зі сцени вони коректно звільняються. Така структура суттєво спрощує підтримку та розширення коду, оскільки дозволяє ізольовано змінювати або тестувати окремі модулі без ризику побічних ефектів у інших частинах програми.

2.4 Огляд патернів MV-сімейства

2.4.1 Вступ до патернів MV-сімейства

Патерни MV-сімейства (Model-View) виникли на межі 1970–1980-х років у середовищі графічних інтерфейсів, зокрема в Smalltalk-системах, як відповідь на проблему надмірного змішування логіки відображення (UI) та бізнес-логіки. Головна ідея полягала в тому, щоб чітко розділити обов’язки:

- *Model* (Модель) — зберігає дані та реалізує бізнес-логіку, не містить жодного коду, пов’язаного з відображенням.
- *View* (Подання) — відповідає лише за відображення даних і взаємодію з користувачем (натискання кнопок, введення тексту), не містить бізнес-логіки.
- *Controller/Presenter/ViewModel* — проміжний шар, що координує взаємодію Model і View, обробляє події користувача та оновлює відображення.

Основні цілі MV-патернів:

1. Слабке зв’язування UI та логіки**.** Забезпечити можливість змінювати інтерфейс без правок у бізнес-логіці й навпаки.

2. Тестованість. Виділення бізнес-логіки в Model і проміжний шар дозволяє зверстувати юніт-тести без залежності від фреймворку UI.

3. Модульність і повторне використання. Компоненти можна легко переносити між проєктами або сценами, оскільки вони реалізують чіткі контракти через інтерфейси.

4. Гнучкість зв’язування даних. У залежності від конкретної реалізації (MVC, MVP, MVVM) застосовуються різні механізми оновлення UI — від ручного виклику методів оновлення до автоматичного Data Binding.

Усі MV-патерни мають спільну архітектурну основу, але розрізняються тим, де саме відбувається зв’язування Model і View, і хто несе відповідальність за передачу даних. У наступних підпунктах розглядається три основні варіанти — MVC, MVVM і MVP.

2.4.2 MVC та його обмеження в Unity

У контексті Unity впровадження класичного MVC часто ускладнюється особливостями рушія та його компонентної системи.

Архітектура MVC

- *Model* — відповідає за дані та бізнес-логіку (стан гри, правила економічного моделювання).
- *View* — MonoBehaviour-скрипти та UI-елементи, що відображають дані та передають події користувача.
- *Controller* — окремі скрипти, які обробляють події з View, викликають методи Model і керують оновленням View.

Основні обмеження MVC в Unity:

1. Тісна інтеграція з MonoBehaviour. Unity-компоненти UI одночасно виступають і View, і Controller, що призводить до змішування відображення та логіки в одному класі.

2. Відсутність Data Binding. MVC передбачає, що Model може сповіщати View про зміни, але Unity не має вбудованого механізму цього сповіщення, тому

розробники створюють кастомні події або використовують UnityEvent, що ускладнює архітектуру.

3. Ускладнене тестування. Controller у вигляді MonoBehaviour залежить від життєвих циклів (Awake, Start, Update) та сцен, що ускладнює його юніт-тестування без значних моків чи емуляцій рушія.

4. Дублювання коду. Через відсутність чіткого розділення часто реалізації обробки подій та роботи з UI дублюються в різних скриптах.

Висновок: MVC у Unity призводить до змішування ролей, дублювання відповідальностей і ускладнень із тестуванням та підтримкою. Через це для клієнтського UI гри доцільніше застосувати патерн, який краще відповідає компонентній природі Unity та забезпечує чітке розділення View і логіки — зокрема, MVP.

2.4.3 MVVM та Data Binding–підхід

Архітектура MVVM

Model–View–ViewModel (MVVM) розвинувся з MVC і набув популярності завдяки фреймворкам із потужним Data Binding (WPF, Xamarin). Його складові:

- *Model* — модель даних і бізнес-логіка, як і в інших MV-патернах.
- *View* — декларативний опис інтерфейсу (XAML, UXML тощо), який не містить бізнес-логіки.
- *ViewModel* — проміжний шар, який виконує двостороннє зв’язування даних між Model і View, реалізує властивості та команди, до яких «прив’язується» View.

Особливості Data Binding

- Декларативне оновлення інтерфейсу користувача. При зміні будь-якої властивості у ViewModel відповідний елемент UI оновлюється автоматично, без необхідності писати окремий код для кожної операції відображення. Це дозволяє зосередитися на логіці даних, а не на механіці синхронізації з інтерфейсом.
- Команди для обробки подій. Логіка реакції на події (натискання кнопок тощо) інкапсулюється в об’єктах-командах ViewModel. Завдяки цьому UI-

елементи лише ініціюють виконання команди, а вся бізнес-логіка залишається відокремленою від шару представлення.

Переваги MVVM

- Повна відсутність коду у View. Вся логіка подій і оновлень описується у ViewModel, що спрощує підтримку й тестування UI.
- Чіткий поділ обов'язків. Model обробляє дані, ViewModel — оркеструє бізнес-логіку, View — лише відображає.
- Потужні фреймворки. Технології на кшталт WPF, Avalonia, Xamarin.Forms пропонують готові механізми Binding, Commands, Converters.

Обмеження MVVM у Unity

- Відсутність вбудованого Data Binding. Unity UI не підтримує двостороннє прив'язування «з коробки». Для реалізації Binding необхідні сторонні бібліотеки (UniRx, uFrame), що збільшує складність і поріг входу.
- Продуктивність. Механізми Binding та спостереження за властивостями можуть створювати додаткові накладні витрати в контексті рендеринга на мобільних пристроях і слабких ПК.
- Крива навчання. Командний та реактивний підходи можуть бути неприродними для команд, що звикли до імперативної стилістики Unity.

Висновок: MVVM забезпечує елегантне розділення View і логіки в середовищах з потужним Data Binding, але в Unity потребує додаткових фреймворків, має підвищені накладні витрати на продуктивність і складнішу конфігурацію.

2.4.4 MVP: ролі View, Presenter і Model

У патерні Model–View–Presenter (MVP) чітко розділено три компоненти, що забезпечує зрозумілу структуру UI-логіки та високу тестованість:

Model

- Призначення: зберігає стан налаштувань (розширення екрану, мова, звук) та реалізує методи їхнього збереження (MainMenuSettingModel).
- Характеристика: не знає про View чи Presenter, надає лише властивості та методи для роботи з даними.

View

- Призначення: відображає інтерфейс налаштувань (випадачі, повзунки, кнопки) і приймає події користувача.
- Характеристика: реалізується як MonoBehaviour (MainMenuSettingView), містить посилання на UI-елементи, передає події Presenter-у через делегати (OnClose тощо).

Presenter

- Призначення: координує взаємодію між View, Model та глобальними сервісами.
- Обов’язки:

1. Ініціалізує Model (_model.Init()) та View (через PrefabManager).
2. Обробляє події View (зміна екрану, мови, звуку).
3. Застосовує налаштування через сервіси (ResolutionScreenService, SoundManager, LocalizationManager).
4. Оновлює View та зберігає дані моделі.
5. Відповідає за коректне очищення ресурсів у Deinitialize().\

2.4.5 Чому MVP найкраще підходить для проєкту

Порівняльний аналіз трьох MV-патернів у контексті Unity показав:

- *MVC* призводить до змішування ролей View і Controller у MonoBehaviour-скриптах, ускладнюючи тестування і порушуючи принципи розділення обов’язків.
- *MVVM* забезпечує чистий поділ UI та логіки через потужний механізм Data Binding, але вимагає сторонніх фреймворків, має вищі накладні витрати і складну конфігурацію в Unity.
- *MVP*, навпаки, організовує UI-логіку таким чином:

1. *View* залишається простим MonoBehaviour-скриптом, відповідаючи лише за відображення та прийом подій.
2. *Presenter* повністю відокремлений від Unity-специфіки, координує взаємодію з Model та глобальними сервісами (PrefabManager, ResolutionScreenService, SoundManager, LocalizationManager).

3. *Model* інкапсулює дані й бізнес-логіку, не залежить від UI або рушія.

Цей розподіл забезпечує:

- Високу тестованість: Presenter і Model можна юніт-тестувати без залучення UnityEngine.
- Гнучкість і розширюваність: додавання нового екрану або зміна існуючого потребує лише створення нового Presenter і View без правок у загальних компонентах.
- Чистоту коду: MonoBehaviour-класи залишаються «тонкими», без бізнес-логіки, що спрощує підтримку та відлагодження.

Отже, з огляду на вимоги проєкту та особливості Unity, патерн MVP є оптимальним вибором для організації UI-логіки.

2.5 Моделювання вимог та сценаріїв

Перед тим як перейти до графічного представлення сценаріїв взаємодії, визначимо їх у вигляді текстових описів. Це дозволить чітко зафіксувати функціональні вимоги та кордони кожного варіанту використання (Use Case) перед побудовою діаграми.

2.5.1 Опис ключових сценаріїв (Use Case)

Use Case використовується для формалізації поведінки системи з погляду її користувачів. У межах гри слід виділили ключові сценарії, які ілюструють взаємодію гравця з інтерфейсом та серверами: від першої авторизації й до виходу зі сесії гри. Опис кожного сценарію допомагає впорядкувати функціонал, знайти потенційні точки розширення та забезпечити відповідність вимогам.

UC1. Авторизація / Реєстрація

- Актор: Гравець
- *Короткий опис:* забезпечує вхід до системи або створення нового облікового запису.
- *Послідовність кроків:*

1. Гравець вводить логін і пароль.
2. Система перевіряє наявність облікового запису:
   - якщо обліковий запис існує, виконується перевірка хешу пароля (PBKDF2);
   - якщо запис відсутній, генерується новий (salt + ітерації → хеш → збереження).
3. У разі успіху — перехід до головного меню; у разі помилки — відображення повідомлення про невдалу авторизацію.

- Рисунок. 2.5 Use Case–діаграма сценарію «Авторизація / Реєстрація».

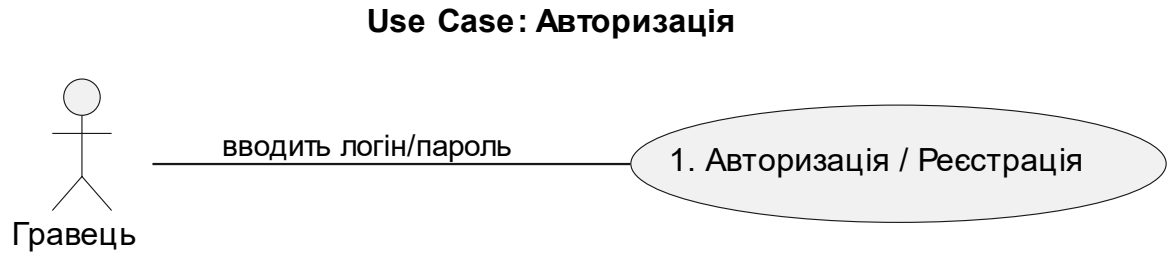


Рис. 2.5 Use Case–діаграма сценарію «Авторизація / Реєстрація»

UC2–UC4. Дія авторизованого гравця

Після успішної авторизації виникає роль «Авторизований гравець», яка може виконувати такі дії:

1. *Робота в головному меню (UC2)*

– Створення лобі (UC2.1): ініціалізація нової ігрової сесії та очікування підключення учасників.

– Приєднання до лобі (UC2.2): входження в існуючу сесію за кодом.

– Кастомізація персонажа (UC2.3): вибір візуальних ігрових атрибутів із збереженням локально.

– Налаштування (UC2.4): коригування мови, роздільності, гучності та Master Audio через відповідні сервіси.

– Вихід із гри (UC2.5): завершення роботи клієнта або повернення до екрана логіну.

2. *Взаємодія в лобі (UC3)*

– Вхід до лобі після UC2.1 чи UC2.2.

– Запуск гри (UC3.1) доступний лише Мастеру лобі.

– Вихід із лобі (UC3.2) доступний будь–кому, повертає до головного меню.

3. *Ігровий процес (UC4)*

– Хід (UC4.1): кидок кубиків із надсиланням події через мережу.

– Придбання будівлі (UC4.2) та оплата штрафу (UC4.3) як розширення ходу.

– Вихід із гри в ході сесії (UC4.4): дострокове завершення з поверненням до меню.

Детальне представлення всіх перерахованих сценаріїв подано на рисунку 2.7 Use Case–діаграма основних сценаріїв авторизованого гравця, що ілюструє послідовність дій авторизованого гравця у клієнтській частині гри.

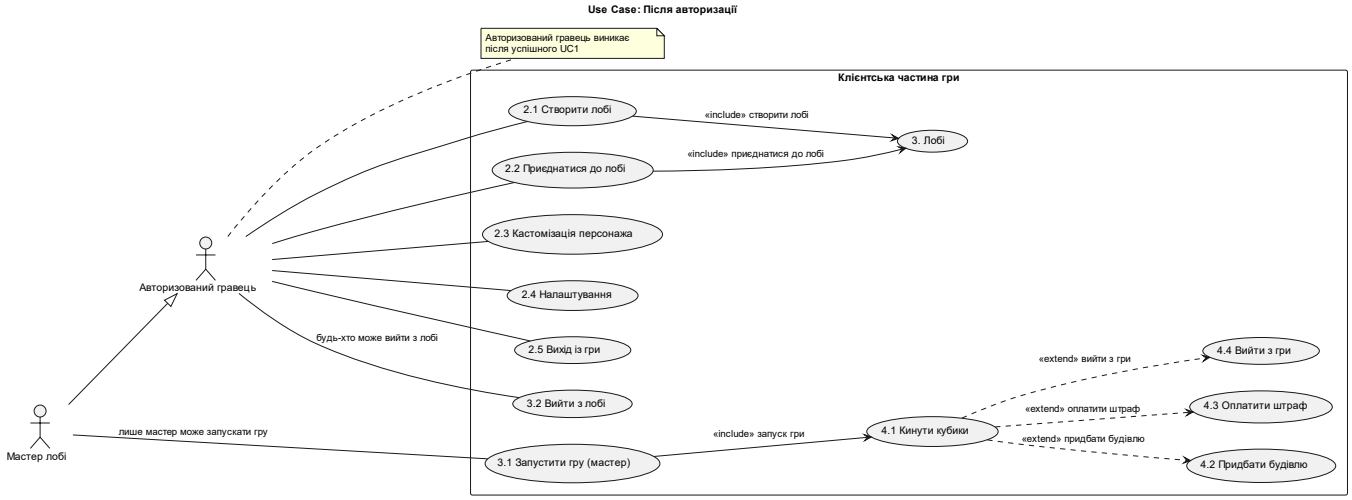


Рис. 2.6 Use Case–діаграма основних сценаріїв авторизованого гравця

Таким чином, два рівні діаграм демонструють повний життєвий цикл користувача: від початкової авторизації до завершення ігрової сесії, що дозволяє впевнено розробляти та тестувати відповідну функціональність.

2.5.2 Діаграми класів

Класові діаграми застосовуються для візуалізації статичної структури системи та відображення взаємозв'язків між її елементами. У контексті гри UML-діаграми класів показують розподіл відповідальностей між компонентами, ієрархію модулів та механізми взаємодії. Наглядне представлення основних класів, їхніх атрибутів і методів дозволяє перевірити коректність обраної модульної архітектури, bootstrap-системи, реалізації Service Locator і патерну MVP, а також спрощує подальшу підтримку й розширення коду.

На рисунку 2.7 подано UML-діаграму рівня Bootstrap and Infrastructure, яка ілюструє початковий етап ініціалізації клієнтської частини гри та створення ключових інфраструктурних сервісів.

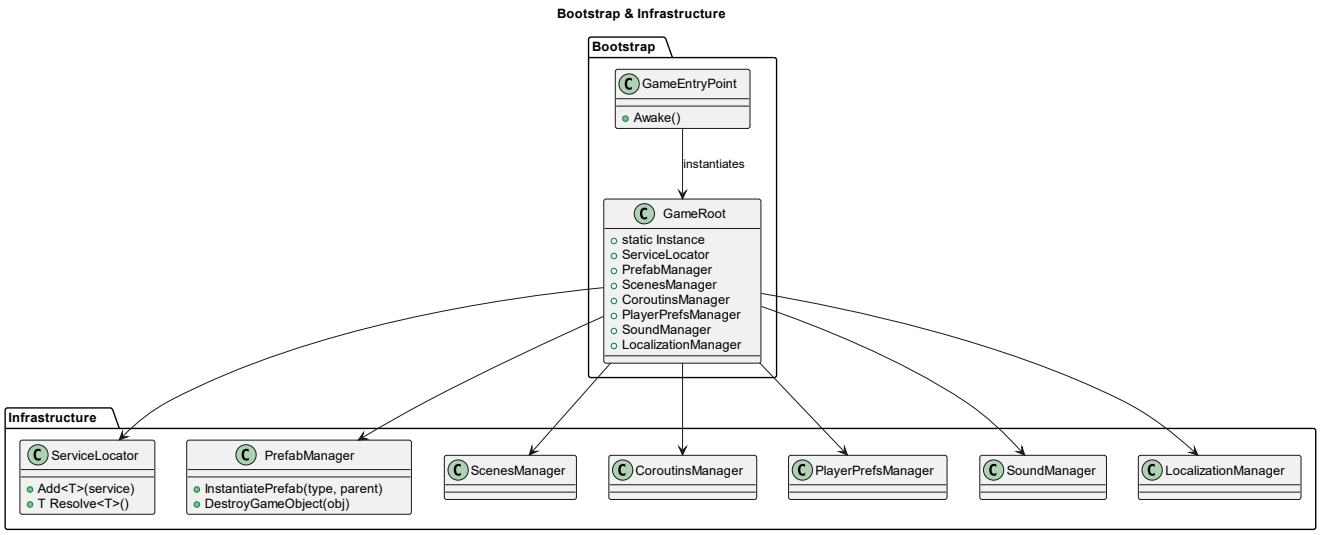


Рис. 2.7 UML-діаграму рівня Bootstrap and Infrastructure

У модулі Bootstrap клас GameEntryPoint відповідає за запуск гри. GameRoot виступає центральним контейнером, що зберігає посилання на всі глобальні підсистеми, які включають:

- ServiceLocator — реалізує механізм інверсії керування (IoC), дозволяючи реєструвати й отримувати сервіси за інтерфейсами.
- PrefabManager — відповідає за інстанціювання та знищення префабів у сценах.

- ScenesManager — забезпечує завантаження та навігацію між сценами з відображенням екранів завантаження.
- CoroutinsManager — координує виконання корутин у межах гри, полегшуючи асинхронні операції.
- PlayerPrefsManager — управляє збереженням користувацьких налаштувань та сесійних даних.
- SoundManager — контролює відтворення аудіо та застосування налаштувань гучності.
- LocalizationManager — обробляє вибір і завантаження мовних ресурсів.

Така структура гарантує єдину точку старту для конфігурації всіх глобальних сервісів і забезпечує легке керування залежностями та їхню доступність протягом усього життєвого циклу проєкту.

На рисунку 2.8 зображено поетапну схему ініціалізації для конкретної ігрової сцени:

1. Локальна точка входу (Entry Point). Після завантаження сцени спрацьовує вхідний модуль, який отримує доступ до глобального контейнера залежностей. На цьому етапі він реєструє локальні сервіси — наприклад, аналітичний, мережевий і сервіс управління камерами — та забезпечує їхню ініціалізацію відповідно до потреб сцени.
2. Ініціалізація основного менеджера сцени. Далі створюється головний менеджер сцени, який відповідає за підготовку ігрового простору: розміщення ігрових об’єктів, налаштування логіки проведення ходів і обробку мережевих подій. Менеджер отримує через DI усі зареєстровані сервіси та встановлює між ними необхідні зв’язки.
3. Запуск додаткових менеджерів. За потреби головний менеджер створює й налаштовує допоміжні підсистеми — наприклад, обробник Remote Procedure Calls для мережевої синхронізації або модуль обробки подій гравця на карті.
4. Ініціалізація UI-модулів. Після налаштування ігрової логіки менеджер сцени ініціює відображення інтерфейсу користувача: створює відповідні UI-модулі,

передаючи їм через контейнер тільки ті залежності, які необхідні для роботи даної сцени (наприклад, сервіси авторизації, налаштувань або аналітики).

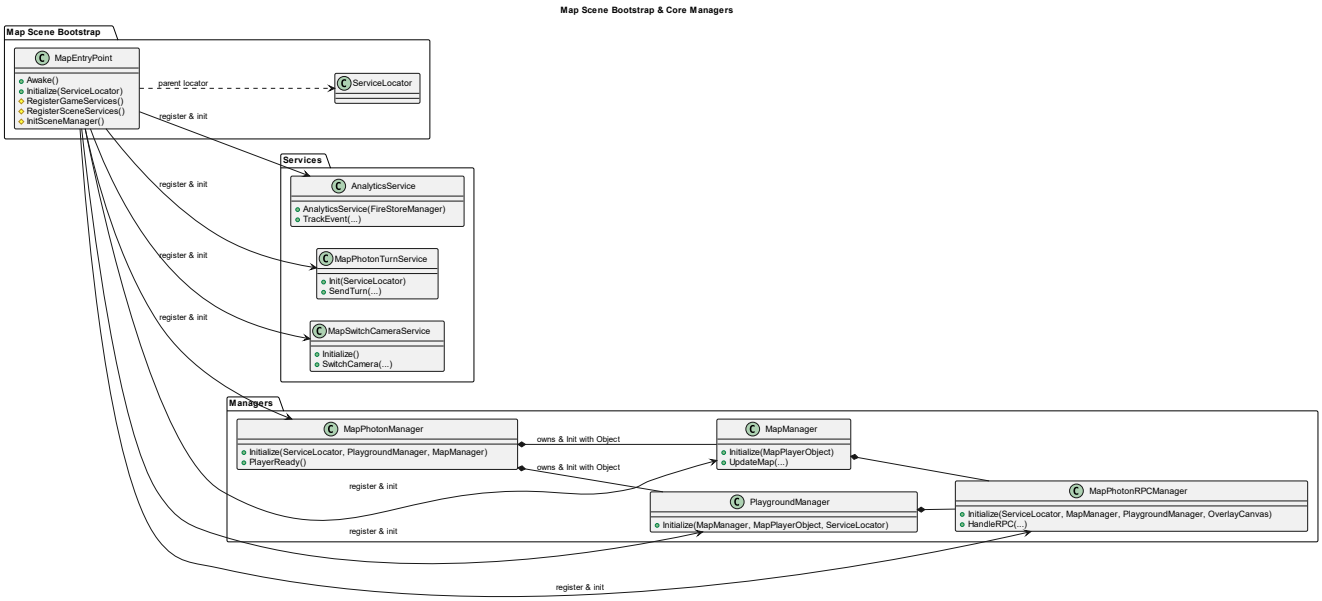


Рис. 2.8 UML-діаграма рівня сцени

Такий підхід гарантує:

- Послідовність — кожен рівень ініціалізації чекає на готовність попереднього.
- Модульність — локальні сервіси та менеджери виділені в окремі компоненти, які можна замінювати або розширювати без зміни глобального стартового механізму.
- Чітке розділення обов’язків — Entry Point відповідає лише за реєстрацію й запуск, менеджери — за бізнес-логіку, а UI-модулі — за відображення та взаємодію з користувачем.

На рисунку 2.9 представлено UML-діаграму підсистеми UI, реалізованої за патерном MVP. Схема ілюструє основні компоненти, їхні ролі та спосіб взаємодії в межах екрану налаштувань головного меню:

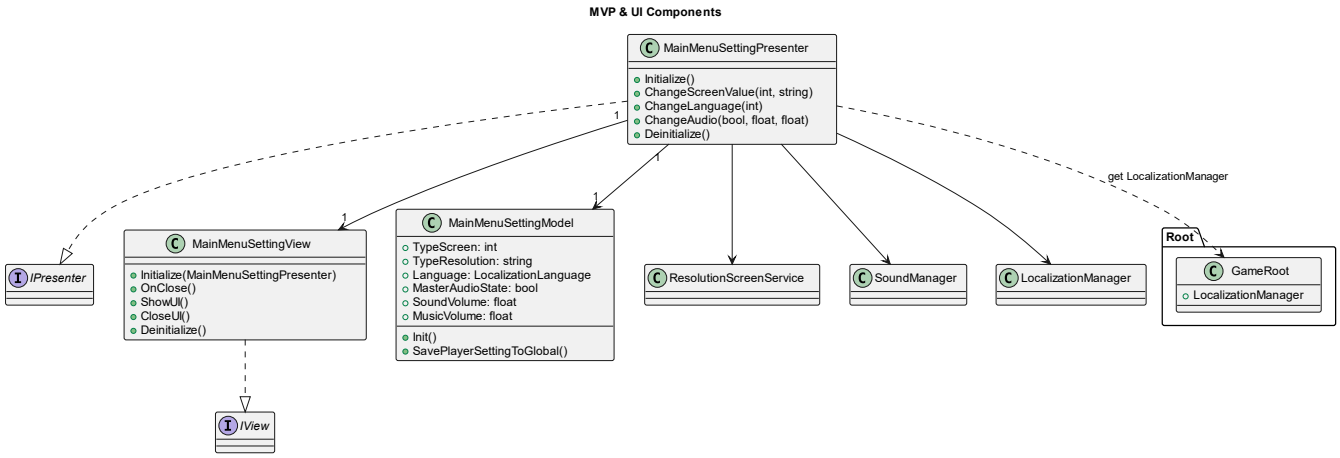


Рис. 2.9 UML-діаграма підсистеми UI

1. Інтерфейси. IPresenter і IView задають контракти для Presenter-ів і View-ів, забезпечуючи слабке зв’язування та уніфікований спосіб ініціалізації й очищення UI-компонентів.
2. Компоненти MVP

Presenter відповідає за життєвий цикл UI:

- Метод Initialize() створює View та Model, підписується на їхні події;

- ChangeScreenValue, ChangeLanguage, ChangeAudio реалізують бізнес-логіку застосування налаштувань;

- Deinitialize() коректно звільняє ресурси.

- View реалізує інтерфейс IView і містить методи відображення (ShowUI/CloseUI), підключення зворотніх викликів (OnClose) і власну ініціалізацію/деініціалізацію.

- Model інкапсулює внутрішній стан налаштувань (екран, мова, аудіо) і надає API для збереження значень.

3. Сервіси

- ResolutionScreenService і SoundManager інкапсулюють застосування технічних налаштувань екрану та звуку. Presenter звертається до них через API

- LocalizationManager отримується з глобального контейнера GameRoot, демонструючи приклад доступу до сервісів, які живуть поза межами конкретної сцени.

4. Зв’язки. Між Presenter-ом, View-ом і Model-лю відображають постійні двосторонні посилання протягом життєвого циклу екрану. А, діаграма підкреслює, що Presenter централізовано координує всі зміни: від ініціалізації UI до виклику зовнішніх сервісів, не порушуючи принципів модульності та інверсії керування.

Таким чином, UML-діаграма підтверджує коректність реалізації MVP-архітектури: Presenter залишається C# класом з чіткими залежностями, View — тонким MonoBehaviour-скриптом, а Model — автономним сховищем даних.

2.6 Проектування структури бази даних

У сучасних мультиплеєрних застосунках для ігор дедалі частіше використовуються NoSQL-сховища через їхню гнучкість у роботі з різнорідними даними, підтримку масштабованості та вбудовані можливості синхронізації в реальному часі. У якості основного бекенду для зберігання даних обрано Google Firestore — документно-орієнтовану базу даних у складі Firebase, яка забезпечує автоматичну синхронізацію змін у реальному часі (Realtime Listeners).

Дані у проєкті організовано в кілька основних колекцій, кожна з яких відображає певний аспект роботи клієнтської частини:

- Players містить документи з інформацією про облікові записи користувачів (login, nickname), параметри PBKDF2-хешування паролів (hash, salt, iterations), а також ігрові метрики — баланс (coin) і кількість виграних сесій (winCount).
- BugReports призначена для збору зворотного зв’язку про помилки; кожен запис містить авторські метадані (userName, dateTime, countPlayer) і деталі проблеми (titleBug, descriptionBug).
- BuildStats відстежує активність гравців щодо ігрових об’єктів, записуючи унікальний ідентифікатор типу будівлі, час останньої активації і загальну кількість використань.
- PlayerStats (або PingPlayerStats) використовується як аналітика продуктивності: зберігаються час фіксації, ping і число учасників у сесії, а за потреби — показник FPS.

Кожна колекція може розширюватися додатковими полями без зміни серверної схеми. Для прискорення запитів налаштовано індекси, що забезпечує високу продуктивність навіть при значному обсязі документів. Реалізація через Realtime Listeners гарантує клієнтам отримання оновлень у режимі live, а локальне кешування — безперебійну роботу при нестабільному з'єднанні.

2.7 Огляд використаних бібліотек та плагінів

У проєкті застосовано низку сторонніх бібліотек і плагінів для Unity, які забезпечують реалізацію ключових підсистем: UI, мережевої взаємодії, збереження даних, рендерингу та управління камерою. Кожен із перелічених компонентів імпортується як Unity Package або через Unity Asset Store, а його налаштування виконується у вікні Package Manager або через відповідні меню плагіна.

2.7.1 TextMesh Pro

Призначення: розширена система рендерингу тексту в Unity, що замінює стандартний компонент Text і забезпечує високу чіткість шрифтів, підтримку стилів (наприклад, жирний, курсив), багатомовність та динамічну зміну контенту.

Ключові можливості:

- Підтримка SDF--шрифтів для чіткого відображення при будь--якому масштабі.
- Вбудовані ефекти — обведення, тіні, градієнти.
- Швидке оновлення тексту без перезавантаження сцени.

Роль у проєкті: використовується для всіх UI-елементів із текстом — кнопок, діалогових вікон, повідомлень про помилки та табличних звітів телеметрії.

2.7.2 Photon PUN 2

Призначення: SDK від Exit Games для побудови мультиплеєрних ігор із моделлю peer-to-peer та клієнт–серверною синхронізацією без необхідності власного серверного рішення.

Ключові можливості:

- Автоматична реплікація об’єктів через Networked Prefabs.
- Виклик RPC та RaiseEvent для синхронізації подій між клієнтами.
- Управління лобі, кімнатами, списками гравців і чергами очікування.

Роль у проєкті: забезпечує створення та приєднання до лобі, передачу ходів (RPC), синхронізацію стану ігрового поля та ігрових подій у реальному часі.

2.7.3 Firebase Firestore SDK

Призначення: клієнтська бібліотека для роботи з Google Firestore — документно-орієнтованим NoSQL-сховищем у складі платформи Firebase.

Ключові можливості:

- Realtime Listeners для автоматичної доставки змін на клієнт без опитування.
- Локальне офлайн--кешування із подальшою синхронізацією.
- Інтеграція з Firebase Authentication і Security Rules для захищеного доступу.

Роль у проєкті: використовується для зберігання інформації про гравців, ігрові сесії, звіти про баги та аналітику продуктивності.

2.7.4 2D Sprite Renderer

Призначення: стандартний Unity-інструмент для відображення спрайтів — растрових зображень, що використовуються як елементи UI та в ігровому полі.

Ключові можливості:

- Анімація через Sprite Atlas та Animator Controller.
- Масштабування та обрізка текстур у Canvas.
- Оптимізація через Sprite Packing.

Роль у проєкті: забезпечує відображення ігрових токенів, фішок, іконок ресурсів і елементів інтерфейсу в 2D--сценах.

2.7.5 Cinemachine

Призначення: модуль камер від Unity для створення динамічних і плавних переходів, слідування за об’єктами та складних кінематографічних ефектів без вручну прописаної логіки.

Ключові можливості:

- Вбудовані типи камер: слідкування (Follow), фокусування (LookAt), транскінг (Transposer).
- Підтримка Blend-режимів для плавних переходів між камерами.
- Інтеграція з Timeline для анімації кінематографічних сцен.

Роль у проєкті: використовується для керування переглядом ігрової карти та фокусування на активному гравцеві під час ходу, що підвищує залученість користувача.

3. РЕАЛІЗАЦІЯ ГРИ

3.1 Структура проєкту

У якості середовища розробки використано Unity 6000.0.46f1. Файли та директорії проєкту організовано за функціональним принципом, що спрощує орієнтацію у кодовій базі та подальшій підтримкі. Структуру проєкту можна переглянути на рисунку 3.1.

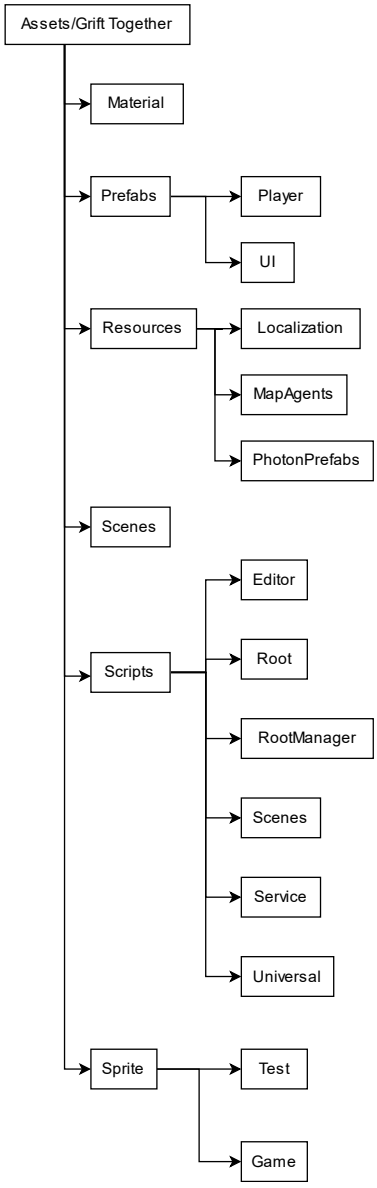


Рис. 3.1 Структура проєкту

3.2 Огляд основних компонентів проєкту

У цьому розділі представлено призначення та вміст ключових папок із кореня Assets/Grift Together, які відображають основні функціональні блоки клієнтської частини гри.

Папка Material

Папка Material містить усі матеріали Unity, які поєднують текстури та налаштування відображення. Кожен матеріал заданий через .mat-файл і використовується для рендерингу 3D- або 2D-об’єктів на сценах. Централізоване зберігання матеріалів спрощує їхнє повторне використання та дає змогу швидко змінювати візуальні властивості (наприклад, колір, прозорість чи текстуру) без необхідності модифікувати кожен об’єкт окремо.

Папка Prefabs

Папка Prefabs містить шаблони готових об’єктів (префаби), які можна повторно використовувати на різних сценах або створювати динамічно в коді.

- Player: префаби ігрових токенів та фішок, що репрезентують гравців.
- UI: готові екрани та компоненти інтерфейсу (кнопки, панелі).

Завдяки префабам кожна зміна в шаблоні автоматично застосовується до всіх його екземплярів на сценах, що суттєво пришвидшує розробку й забезпечує узгодженість UI та ігрових об’єктів.

Папка Resources

У папці Resources зберігаються дані, які завантажуються за допомогою *Resources.Load()* під час виконання:

- Localization: CSV–та інші файли з перекладами інтерфейсу.
- MapAgents: скриптовані об’єкти й дані для генерації агентів на карті.
- PhotonPrefabs: префаби мережевих об’єктів для Photon PUN 2 (напр., *NetworkedPlayer*).

Ця папка дозволяє гнучко підвантажувати ресурси без включення їх у конкретну сцену, а також відокремлювати конфігураційні файли від решти Assets.

Папка Scenes

Директорія Scenes містить усі Unity-сцени клієнтської частини:

- Login — екран авторизації та реєстрації;
- MainMenu — головне меню з вибором режимів;
- Lobby — лобі та підготовка до гри;
- SmallMap — безпосередній ігровий процес.

Кожна сцена починається з виклику відповідного *SceneEntryPoint*, що реєструє локальні сервіси та менеджери перед завантаженням UI.

Папка Scripts

Ця директорія об’єднує весь C#-код за підпапками відповідно до ролі:

- Editor — інструменти для розробки (*Custom inspectors*).
- Root — *GameEntryPoint, GameRoot, ServiceLocator*.
- RootManager — глобальні сервіси (*ScenesManager, SoundManager, LocalizationManager*).
- Scenes — локальні точки входу (*MapEntryPoint, MainMenuEntryPoint*).
- Service — допоміжні служби (*ResolutionScreenService, AnalyticsService*).
- Universal — загальні утиліти та розширення (корисні класи).

Структурований поділ дозволяє швидко знаходити потрібний функціонал та ізолювати компоненти для тестування і рефакторингу.

Папка Sprite

Папка Sprite призначена для зберігання растрових зображень у форматі PNG і не тільки, які використовуються як елементи інтерфейсу та 2D-об’єктів на сценах. Окремий каталог дозволяє централізовано налаштовувати спрайти, оптимізувати розмір текстур і забезпечувати однорідний вигляд UI-ікон.

Такий поділ папок і файлів за функціональними зонами забезпечує зручну навігацію по проекту, прискорює розробку та полегшує подальший супровід коду.

3.3 Опис реалізації програмного коду

3.3.1 Глобальний Bootstrap

У проєкті задіяно єдину точку входу — клас *GameEntryPoint*, який ініціюється ще до завантаження першої сцени завдяки атрибуту *[RuntimeInitializeOnLoadMethod]*. Нижче наведено рисунок 3.2 з спрощеною версію основного коду з позначенням ключових кроків:

```
GriftTogether                                    GameEntryPoint
1   using GriftTogether;
2   using UnityEngine;
3
    1 reference
4   public class GameEntryPoint {
5
6
7       [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
        0 references
8       static void StartEntryGame() => new GameEntryPoint().RunGame();
9
10      // Головний метод ініціалізації
        1 reference
11      void RunGame() {
12          InitUI();           // 1. Налаштування UI та менеджера сцен
13          InitSave();         // 2. Завантаження збережених налаштувань
14          InitDI();           // 3. Створення контейнера залежностей
15          InitSettings();     // 4. Застосування екранних і звукових параметрів
16          InitBackends();     // 5. Підключення Firestore і Photon
17          GoToLogin();        // 6. Перехід до екрану логіну
18      }
19
20      //...
21
```

Рис. 3.2 Єдину точку входу для проєкту

Пояснення кроків:

1. *InitUI()* — створення *PrefabManager* для префабів, запуск *CoroutinsManager* (щоб підписатися на події оновлення) і налаштування *ScenesManager* із показом екрану завантаження.

2. *InitSave()* — ініціалізація *PlayerPrefsManager і PlayerGlobalManager*, а також завантаження збережених налаштувань (екран, звук, мова).

3. *InitDI()* — створення нового *ServiceLocator*, який виступає контейнером залежностей для всіх подальших сервісів.

4. *InitSettings()* — застосування екранних параметрів через *ResolutionScreenService*, запуск *SoundManager* із поточними налаштуваннями та ініціалізація *LocalizationManager*. Крім того, підписка на *OnStartUnity* для динамічного оновлення звуку.

5. *InitBackends()* — підключення бекендів: створення FireStoreManager для роботи з *Firestore* і запуск *PhotonManager* для мережевої синхронізації (*RPC*, *RaiseEvent*).

6. *GoToLogin()* — остаточний виклик *ScenesManager.SwitchScene* для завантаження сцени логіну.

Підсумовуючи, *GameEntryPoint* та супутні класи *GameRoot* ***і*** *ServiceLocator* формують каркас клієнтської архітектури, забезпечуючи централізовану конфігурацію, послідовну реєстрацію та запуск всіх підсистем гри перед переходом у інші екрани.

3.3.2 Поетапний старт через BaseEntryPoint

У всіх дочірніх сценах застосовано абстрактний клас BaseEntryPoint, що реалізує загальний алгоритм реєстрації сервісів і запуску менеджерів у чітко визначеному порядку. Нижче наведено код цього класу з коментарями на рисунку 3.3:

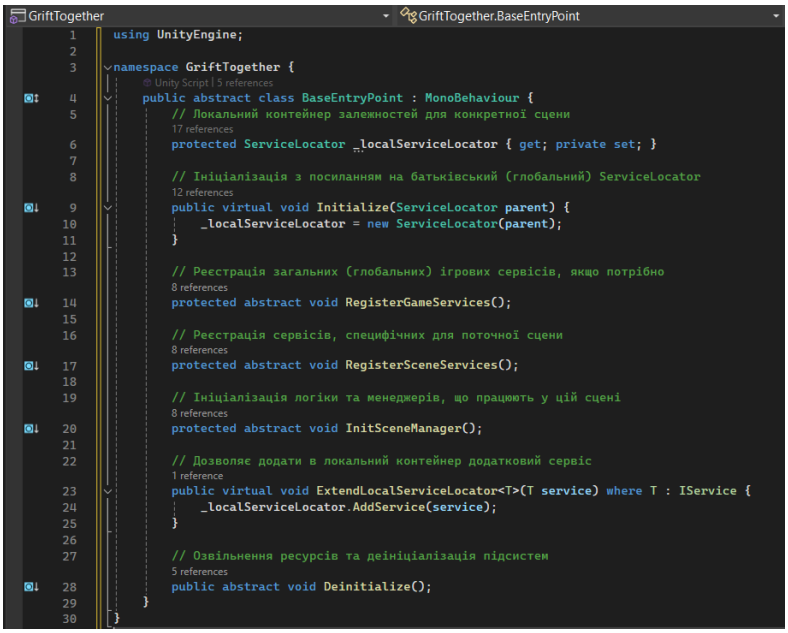


Рис. 3.3. BaseEntryPoint

Пояснення ключових етапів:

1. Ініціалізація контейнера. При виклику *Initialize(parent)* створюється новий *ServiceLocator*, у який передається посилання на глобальний контейнер. Це гарантує доступ до вже зареєстрованих глобальних сервісів поряд з новими, локальними.

2. RegisterGameServices(). Викликається у випадку, якщо сцені потрібні загальні ігрові сервіси, яких немає в глобальному контексті. У більшості сцен цей метод може бути порожнім або не перевизначатися.

3. RegisterSceneServices(). Кожна сцена додає свої специфічні служби (наприклад, *MapPhotonTurnService, AnalyticsService*) у локальний контейнер залежностей.

4. InitSceneManager(). Тут створюються й запускаються менеджери логіки сцени (наприклад, *LobbyManager, MapManager, GameManager*). Кожен менеджер отримує через *_localServiceLocator* необхідні сервіси.

5. ExtendLocalServiceLocator(). Дозволяє динамічно додавати сервіси після початкової ініціалізації, не порушуючи загальної послідовності.

6. Deinitialize(). Забезпечує коректне знищення локальних менеджерів і відписку від подій при виході зі сцени.

Таким чином, *BaseEntryPoint* задає єдиний шаблон ініціалізації для всіх сцен, забезпечуючи послідовну реєстрацію та запуск підсистем.

3.3.3 Локальні менеджери сцен

Для кожної сцени в проєкті визначено власний менеджер, що наслідує абстрактний клас *BaseSceneManager,* який зображений на рисунку 3.4. Цей підхід гарантує єдиний інтерфейс для запуску й очищення логіки сцени та уніфікує їхню ініціалізацію.

```
GriftTogether                                                    BaseSceneManager
// Базовий контракт для всіх менеджерів сцен
0 references
public abstract class BaseSceneManager {
    // Викликається одразу після старту сцени, тут налаштовується UI і логіка
    0 references
    public abstract void Initialize();

    // Викликається при виході зі сцени — звільнення ресурсів і підписок
    0 references
    public abstract void Deinitialize();
}
```

Рис. 3.4 BaseSceneManager

Як приклад реалізації розглянемо *MainMenuManager*, який керує головним меню, рисунок 3.5:

```
GriftTogether                                                    MainMenuManager
3 references
public class MainMenuManager : BaseSceneManager {
    private readonly GameObject _mainCanvas;
    private readonly GameObject _overlayCanvas;
    private readonly ServiceLocator _serviceLocator;

    private MainMenuRootUIView _rootView;
    private MainMenuHeaderPresenter _headerPresenter;
    private MainMenuPresenter _menuPresenter;

    // У конструкторі отримуємо кореневі канваси та локальний контейнер сервісів
    1 reference
    public MainMenuManager(GameObject mainCanvas,
                           GameObject overlayCanvas,
                           ServiceLocator serviceLocator) {
        _mainCanvas = mainCanvas;
        _overlayCanvas = overlayCanvas;
        _serviceLocator = serviceLocator;
    }

    // Ініціалізуємо View та Presenter-и для головного меню
    2 references
    public override void Initialize() {
        // Створюємо кореневий UI із префабу
        _rootView = GameRoot.PrefabManager
            .InstantiatePrefab(MainMenuPrefabType.RootView, _mainCanvas)
            .GetComponent<MainMenuRootUIView>();

        // Заголовок меню
        _headerPresenter = new MainMenuHeaderPresenter(
            _rootView.GetHeaderParent
        );
        _headerPresenter.Initialize();

        // Основний Presenter з доступом до кнопок і сервісів
        _menuPresenter = new MainMenuPresenter(
            _rootView.GetLeftCornerParent,
            _overlayCanvas,
            _serviceLocator
        );
        _menuPresenter.Initialize();
    }

    // Видаляємо UI та звільняємо ресурси
    2 references
    public override void Deinitialize() {
        _menuPresenter.Deinitialize();
        _headerPresenter.Deinitialize();
        GameRoot.PrefabManager.DestroyGameObject(
            _rootView.gameObject
        );
    }
```

Рис. 3.5. MainMenuManager

Ключові моменти реалізації:

1. Уніфікований інтерфейс: всі менеджери мають методи *Initialize()* і *Deinitialize(),* що спрощує керування життєвим циклом сцени.

2. Ін'єкція залежностей: в конструтор передаються відповідні *GameObject*-канваси та *ServiceLocator,* завдяки чому менеджер може створювати UI та звертатися до сервісів без жорстких зв'язків із глобальними об'єктами.

3. Відділення View і Presenter: спочатку створюється кореневий UI (*RootUIView*), потім ініціалізуються Presenter-и, які відповідають за окремі частини екрану (заголовок, меню).

4. Звільнення ресурсів: у *Deinitialize()* кожен Presenter деініціалізується, а префаб кореневого UI руйнується через *PrefabManager*.

Таким чином, B*aseSceneManager* і його конкретні реалізації (на кшталт *MainMenuManager, LobbyManager* тощо) забезпечують чіткий шаблон для запуску та завершення роботи кожної Unity-сцени, із суворим розподілом відповідальностей і мінімальним зв'язуванням.

3.3.4 UI-модулі за патерном MVP

Для реалізації інтерфейсу застосовано патерн *Model–View–Presenter (MVP)***.** У його основі лежать три контракти, які зображені на рисунку 3.6.:

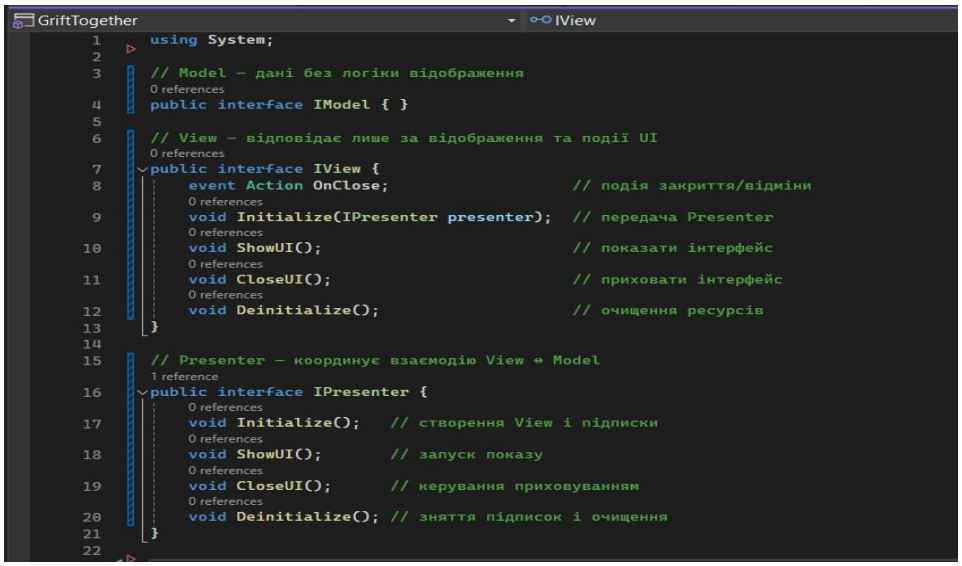


Рис. 3.6 Контракти для патерну MVP

Кожен *Presenter* отримує через конструктор або методи потрібні залежності (*GameObject*-контейнери, сервіси тощо), створює *View* за допомогою *PrefabManager*, підписується на його події та керує логікою відображення.

На рисунку 3.7 зображен приклад реалізації *presenter-a* для виходу з гри:

GriftTogether — MainMenuExitPresenter

```
using GriftTogether;
using System;
using UnityEngine;

6 references
public class MainMenuExitPresenter : IPresenter {
    private readonly GameObject _root;
    private ConfirmUIView _view;

    public event Action OnBack;  // подія повернення без виходу

    1 reference
    public MainMenuExitPresenter(GameObject root) {
        _root = root;
    }

    2 references
    public void Initialize() {
        // Інстанціюємо ConfirmView у заданому контейнері
        _view = GameRoot.PrefabManager
            .InstantiatePrefab(
                UniversalPrefabType.ConfirmView,
                _root
            )
            .GetComponent<ConfirmUIView>();

        _view.Initialize(this);
        _view.OnConfirm += ExitGame;
        _view.OnDecline += Decline;
    }

    2 references
    private void Decline() {
        // Повідомляємо клієнта про відміну
        OnBack?.Invoke();
    }

    2 references
    private void ExitGame() {
        // Показуємо екран завантаження й зберігаємо дані
        GameRoot.ScenesManager.ShowLoadingScreen();
        GameRoot.PlayerGlobalManager.SavePlayerServerData();
        GameRoot.PlayerGlobalManager.SavePlayerSetting();

        Deinitialize();
        Application.Quit();  // завершення гри
    }

    2 references
    public void ShowUI() => _view.ShowUI();
    3 references
    public void CloseUI() => _view.CloseUI();

    3 references
    public void Deinitialize() {
        _view.OnConfirm -= ExitGame;
        _view.OnDecline -= Decline;
        OnBack = null;

        _view.Deinitialize();
        GameRoot.PrefabManager.DestroyGameObject(_view.gameObject);
    }
}
```

Рис. 3.7 MainMenuExitPresenter

Пояснення реалізації:

1. Ін’єкція контейнера (*GameObject root*), куди вставляється підтверджувальне вікно.
2. Initialize():
   - Створення *ConfirmUIView* із префабу.
   - Підписка на події *OnConfirm* (підтвердження виходу) і *OnDecline* (відміна).
3. ExitGame():
   - Показ екрану завантаження.
   - Збереження налаштувань і даних користувача через *PlayerGlobalManager*.
   - Вихід із застосунку (*Application.Quit()*).
4. Deinitialize(): знімання підписок, виклик *Deinitialize()* у *View* та видаленням *GameObject*.

Таким чином, завдяки чітким контрактам і строгому розділенню ролей MVP-модуль повністю ізольований від решти логіки, легко тестується та розширюється.

3.3.5 Головні менеджери та сервіси

У проєкті всі ключові підсистеми реалізовано у вигляді сервісів та менеджерів, що відповідають контракту ініціалізації й очищення, представлений на рисунок. 3.8. Їх реєстрація відбувається в контейнері *ServiceLocator*, завдяки чому компоненти можуть централізовано створюватися та отримуватися без жорстких залежностей.

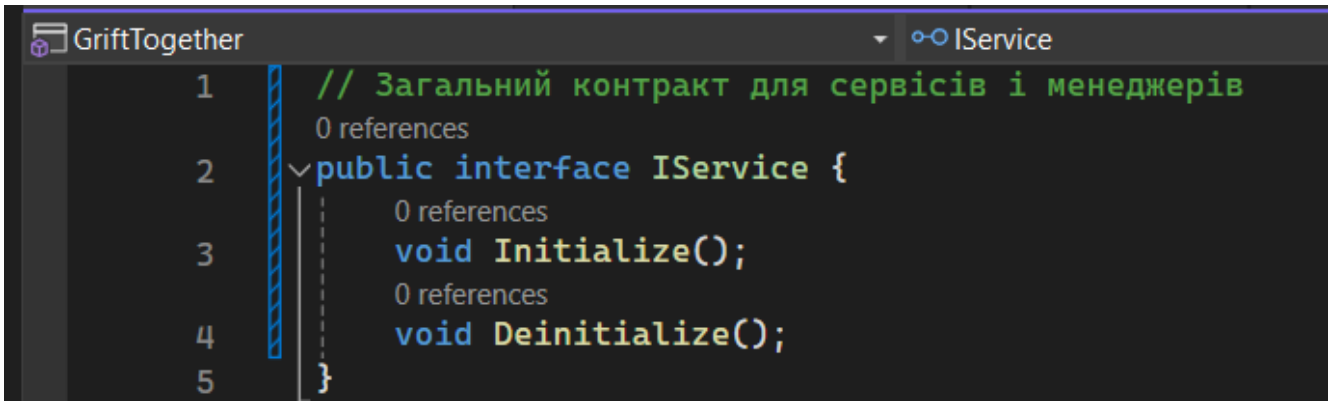


Рис. 3.8 Контракт для ServiceLocator

Контейнер залежностей

- ServiceLocator ініціалізується в *GameEntryPoint* і слугує головним реєстром сервісів.
- Методи *AddService<T>(T service) і Resolve<T>(out T service)* забезпечують слабке зв’язування компонентів.

За навігацію між Unity-сценами та керування екраном завантаження відповідає ScenesManager (рисунок 3.9).

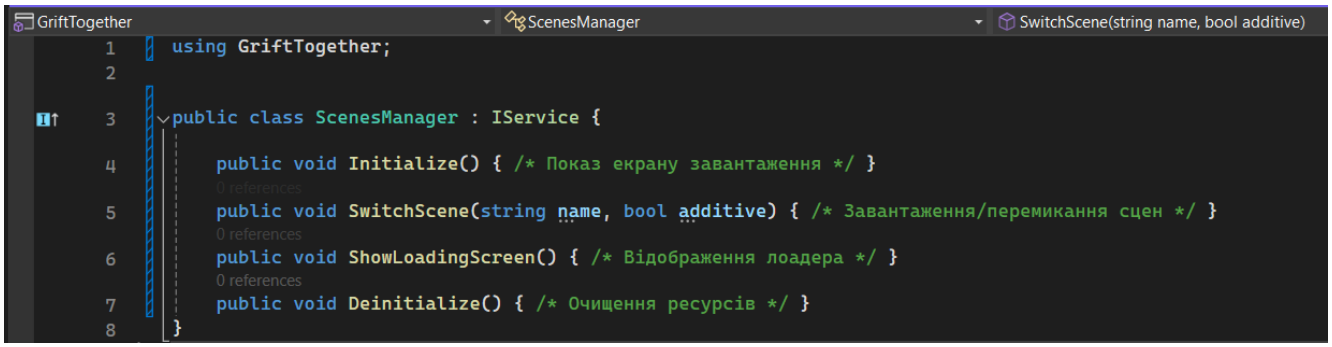


Рис. 3.9 ScenesManager

За динамічне створення та знащення UI-та ігрових об’єктів з префабів відповідає PrefabManager (рисунок 3.10).

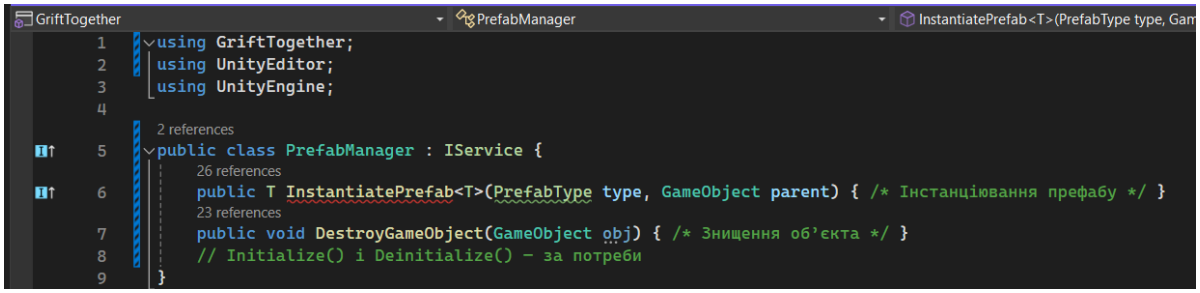


Рис. 3.10 PrefabManager

Надає подію *OnStartUnity*, яка використовується для виконання динамічного коду (наприклад, оновлення налаштувань звуку) після завантаження всіх сцен. Та використовується для запуску *Coroutins* поза межами *MonoBehaviour-класів* (рисунок 3.11).

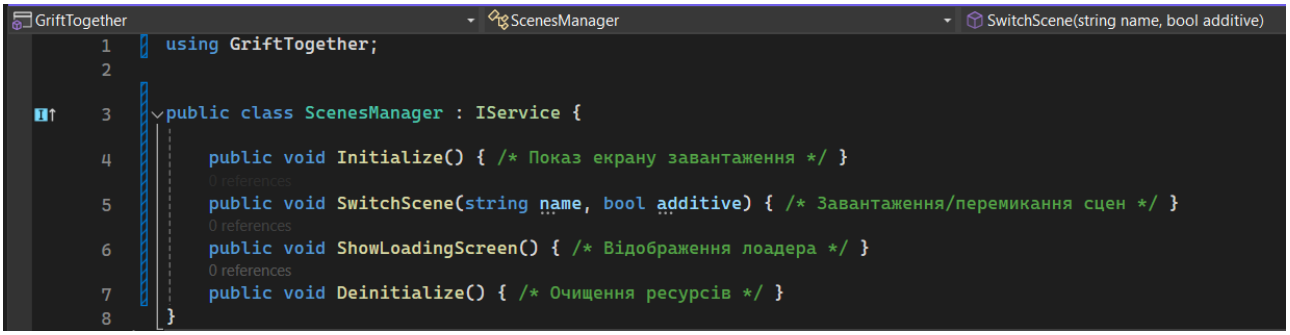


Рис. 3.11 CoroutinsManager

Уніфікований доступ до локального кешу PlayerPrefs для збереження й завантаження налаштувань гравця *PlayerPrefsManager* (рисунок. 3.12.).

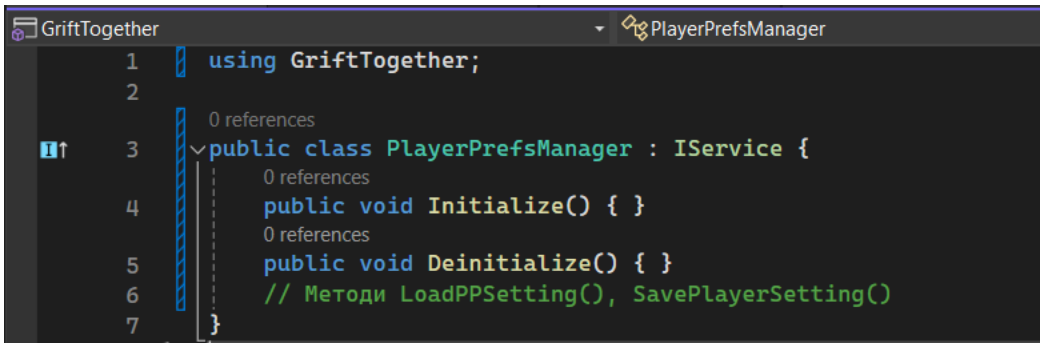


Рис. 3.12 PlayerPrefsManager

За завантаження мовних пакетів та підміну текстів в UI згідно з вибором гравця відповідає *LocalizationManager* (рисунок 3.13)

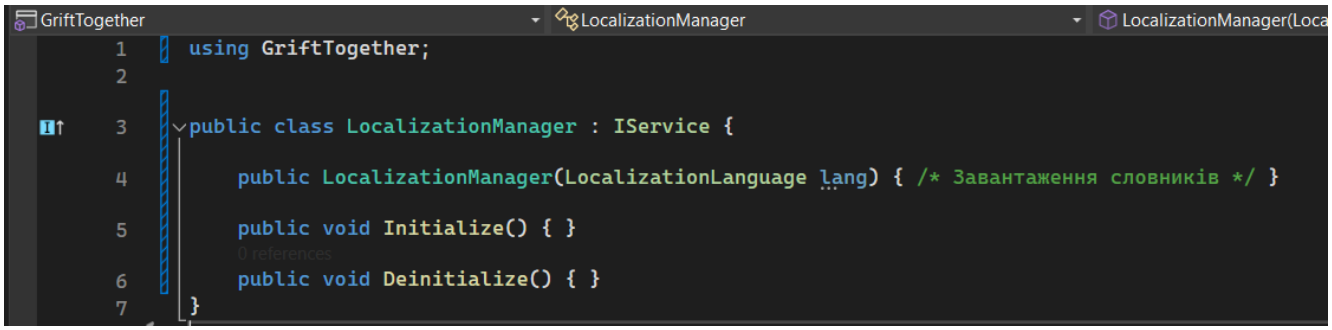


Рис. 3.13. LocalizationManager

Обмін даними з Google Firestore відбувається завдяки наступним класам:

- FireStoreManager — взаємодія з документа-орієнтованою базою, синхронізація в реальному часі;
- AnalyticsService — надсилання кастомних подій у рамках аналітики.

Їхня реалізація представлена на рисунку 3.14.

Рис. 3.14. FireStoreManager і AnalyticsService

За мережеву взаємодію, створення та приєднання до кімнат, синхронізацію ходів і подій між клієнтами відповідає *PhotonManager* (рисунок 3.15)

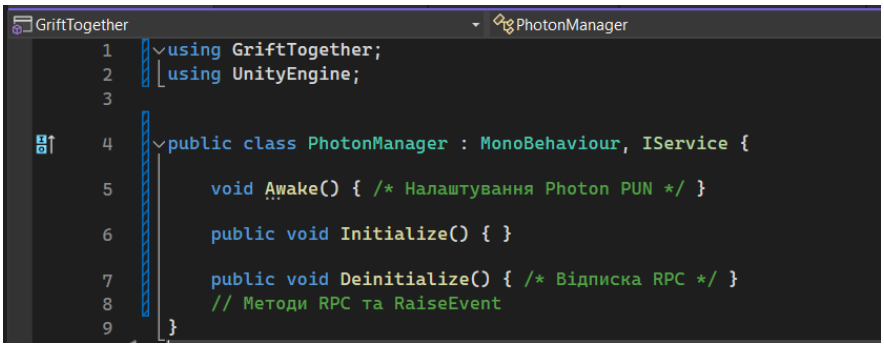


Рис. 3.15 PhotonManager

Додаткові сервіси

- ResolutionScreenService — зміна режиму та роздільності екрану (UI).
- MapPhotonTurnService — передача ходів гравців у покрокових іграх.
- PlaygroundTradeService — логіка обміну ресурсами між гравцями.

Виконуючи контракт *IService*, кожен з цих компонентів реєструється в *ServiceLocator*, ініціалізується одразу після запуску гри та деініціалізується при виході з відповідної сцени, що гарантує стабільну роботу та відсутність витоків пам’яті.

3.3.6 Інтеграція мережі та Firestore

У рамках мультиплеєрної функціональності та збереження даних проєкт використовує два ключові бекенди: Photon PUN 2 для синхронізації ігрових подій між клієнтами та Google Firestore для зберігання облікових записів та аналітики.

Photon PUN 2 (RPC)

Клас *MapPhotonRPCService*, який представлений на рисунку 3.16 інкапсулює всі виклики RPC у сцені карти. Він отримує через *PhotonView* глобальний контекст *MapPhotonRPCContext*, у якому зберігається відповідний *MapPhotonRPCManager*.

```
GriftTogether                                   MapPhotonRPCService
1   using GriftTogether;
2   using Photon.Pun;
3   using UnityEngine;
4
    Unity Script | 0 references
5   public class MapPhotonRPCService : MonoBehaviour, IService {
6       private PhotonView _view;
7       private MapPhotonRPCManager _manager;
8
        Unity Message | 0 references
9       void Start() => Initialize();
10
        1 reference
11      public void Initialize() {
12          _view = GetComponent<PhotonView>();
13          var ctx = GameRoot.PhotonManager.CurrentPhotonContext as MapPhotonRPCContext;
14
15          if (ctx != null) _manager = ctx.GetRPCManage;
16          else Debug.LogError("Не знайдено MapPhotonRPCContext!");
17      }
18
        0 references
19      public void RPC_SendNextTurn()
20          => _view.RPC(nameof(RPC_GetNextTurn), RpcTarget.All);
21
22      [PunRPC]
        1 reference
23      public void RPC_GetNextTurn()
24          => _manager.GetNextTurn();
25
26      // Інші виклики: логування, купівля, оренда, зняття власності
        0 references
27      public void RPC_BuyBuild(string msg, string id, int idx)
28          => _view.RPC(nameof(RPC_GetBuyBuild), RpcTarget.All, msg, id, idx);
29
30      [PunRPC]
        1 reference
31      public void RPC_GetBuyBuild(string msg, string id, int idx) {
32          _manager.BuyBuild(id, idx);
33          _manager.SpawnFadeLog(msg);
34      }
35
36      // … аналогічно для RPC_Rent, RPC_RemoveBuild
37  }
```

Рис. 3.16 Клас MapPhotonRPCService зі специфікацією RPC-методів

1. RPC_SendNextTurn() надсилає запит про настання наступного ходу всім гравцям.

2. Позначені атрибутом [PunRPC] методи (наприклад, RPC_GetNextTurn, RPC_GetBuyBuild) виконуються на кожному клієнті після виклику _view.RPC.

3. Така архітектура гарантує, що всі клієнти завжди знаходяться в одному стані: ходи, купівля будівель, сплата оренди тощо.

Google Firestore

Клас *FireStoreManager*, продемонстрований на рисунку 3.17 відповідає за базові операції читання/запису документів у Firestore.

GriftTogether — FireStoreManager

```
using Firebase.Firestore;
using GriftTogether;
using System.Threading.Tasks;

7 references
public class FireStoreManager {
    private readonly FirebaseFirestore _firestore;

    1 reference
    public FireStoreManager()
        => _firestore = FirebaseFirestore.DefaultInstance;

    // Перевіряє, чи існує документ у вказаній колекції
    2 references
    public async Task<bool> ExistFile(
        string collection, string name, bool toLower = false) {
        var id = toLower ? name.ToLower() : name;
        var snap = await _firestore
            .Document($"{collection}/{id}")
            .GetSnapshotAsync();
        return snap.Exists;
    }

    // Зберігає DTO-об'єкт у Firestore
    6 references
    public async Task SaveToCloud<T>(
        T dto, string collection, string name, bool toLower = false)
        where T : IFireStoreDTO {
        var id = toLower ? name.ToLower() : name;
        await _firestore
            .Document($"{collection}/{id}")
            .SetAsync(dto);
    }

    // Отримує DTO з Firestore, якщо він існує
    2 references
    public async Task<T> TryGetFile<T>(
        string collection, string name, bool toLower = false)
        where T : IFireStoreDTO {
        var id = toLower ? name.ToLower() : name;
        var snap = await _firestore
            .Document($"{collection}/{id}")
            .GetSnapshotAsync();

        return snap.Exists ? snap.ConvertTo<T>() : default;
    }
}
```

Рис. 3.17. FireStoreManager

- ExistFile(...) дозволяє перевірити наявність документа без завантаження даних.
- SaveToCloud(...) записує будь-який тип, що реалізує *IFireStoreDTO*, у вказану колекцію.
- TryGetFile<T>(...) читає документ і конвертує його у відповідний DTO-клас.

Такий поділ обов'язків між *MapPhotonRPCService* і *FireStoreManager* дозволяє чітко ізолювати мережеву логіку від збереження даних, спрощує тестування і подальший розвиток бекенду без зміни клієнтської частини.

Висновок

У цьому розділі детально розглянуто архітектуру та реалізацію клієнтської частини гри: від єдиної точки входу та поетапної ініціалізації через Bootstrap-систему до модульних менеджерів сцен і UI-модулів за патерном MVP. Описані контракти *ServiceLocator* та основні сервіси (*ScenesManager, PrefabManager, CoroutinsManager* тощо) демонструють централізовану конфігурацію та слабке зв'язування компонентів. Для кожної сцени запроваджено єдиний шаблон через *BaseEntryPoint і BaseSceneManager*, що забезпечує послідовну реєстрацію локальних сервісів і життєвий цикл *Presenter-ів і View-ів*. Інтеграція мережевої взаємодії через Photon PUN 2 і збереження даних в Google Firestore гарантують надійну синхронізацію станів гри та гнучку аналітику. У сукупності це створює стійку, масштабовану і легко підтримувану клієнтську архітектуру для розробки мультиплеєрної покрокової економічної стратегії на Unity.

4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОДУКТУ

4.1 Загальні відомості про застосунок

Гра реалізує всі ключові сценарії взаємодії користувача із системою, які включають в себе:

- Реєстрація та вхід – екран введення логіна й пароля, хешування паролів методом PBKDF2, обмін запитом із Firestore для створення або перевірки облікового запису.
- Створення/приєднання до ігрової сесії – вибір режиму «Нова гра» або «Приєднатися», очікування інших гравців у лобі, відображення стану сесії.
- Виконання ходу – кидок кубика, рух фішки по ігровому полю, покупка активів і виплата оренди; відправка подій Photon PUN 2 для синхронізації стану гри на всіх клієнтах.
- Мережна синхронізація – передача подій «Початок ходу», «Кінець ходу», оновлення позицій ігрових фішок, передача аналітики гравця у Firestore.

4.1.1 Технічне середовище

Розробка і тестування проводились із такими компонентами:

- Ігровий рушій: Unity 6
- Мережевий рівень: Photon PUN 2
- Бекенд: Firebase Unity SDK (Firestore)
- Мова програмування: C# - 8
- Платформи тестування: Windows 10 (64-bit)
- Інструменти моніторингу й профілювання: Unity Profiler, логування через Debug.Log, засоби Firestore Console, вбудований Crash Handler Unity.

4.1 Тестування за ключовими сценаріями користувача (Use Case)

4.2.1 UC1: Реєстрація та вхід

Кроки тестування:

1. Відкрити екран реєстрації/входу.
2. Ввести коректні логін та пароль.
3. Натиснути кнопку “Увійти” / “Зареєструватися”.
4. Очікувати відповіді від Firestore.

Очікуваний результат:

- Успішне створення нового облікового запису (якщо реєстрація) або вхід у систему (якщо існуючий користувач).
- У Firestore з’являється документ із коректним полем.

Фактичний результат: Успіх, гравець створив новий обліковий запис подано на рисунку 4.1 та перейшов на нову сцену зображено на рисунку 4.2.

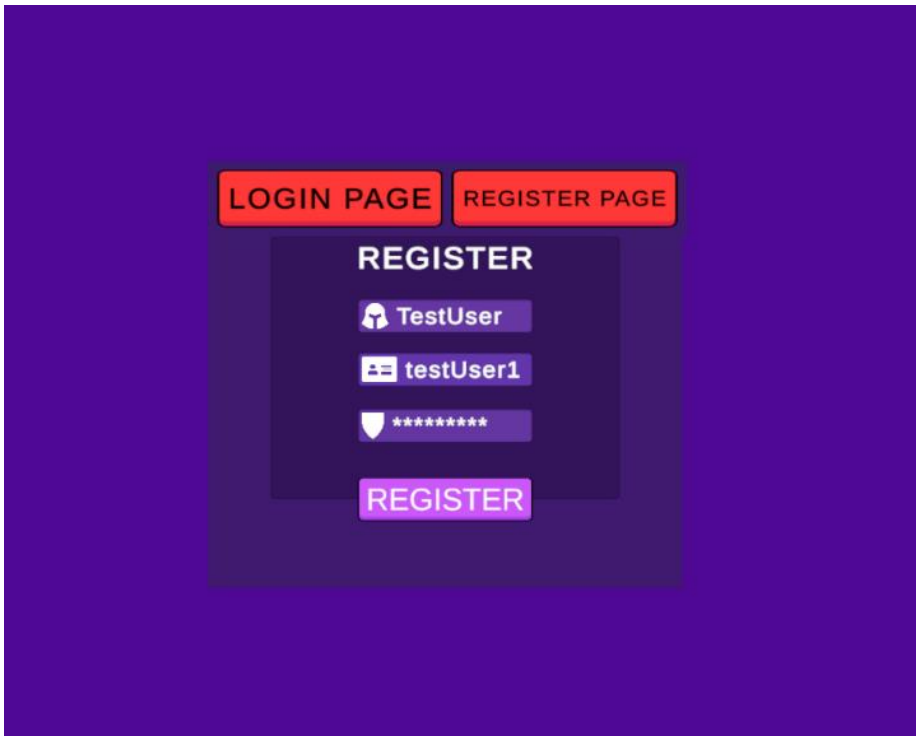


Рис. 4.1 Реєстрація нового гравця

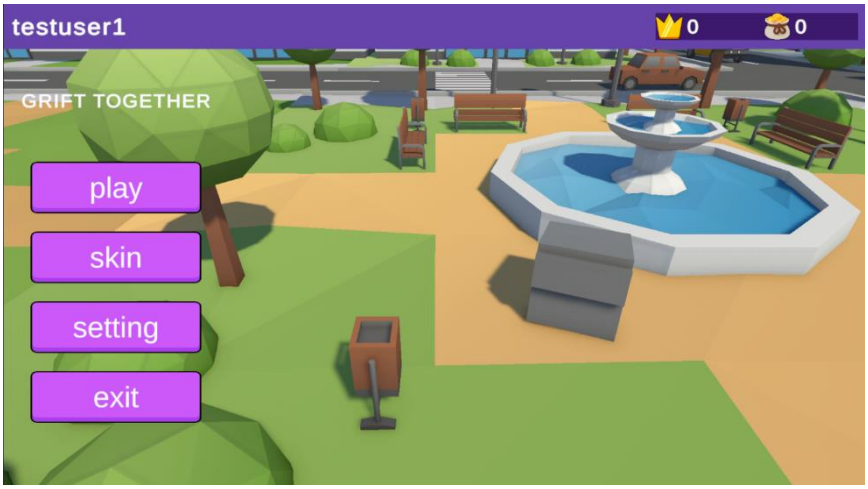


Рис. 4.2 Нова сцена після успішної реєстрації

4.2.2 UC2: Створення/приєднання до ігрової сесії

Кроки тестування:

1. Вибрати “Нова гра” або “Приєднатися до гри”.
2. Дочекатися появи лобі з індикатором кількості гравців.
3. Перевірити, що в обох клієнтів відображається однакова інформація про учасників.

Очікуваний результат:

- Створюється лобі з коректною кількістю слотів.
- Інформація про гравців синхронізується між клієнтами без затримок понад 100 мс.

Фактичний результат: Успіх гравеці коректно приєднався до лобі, рисунок 4.3.

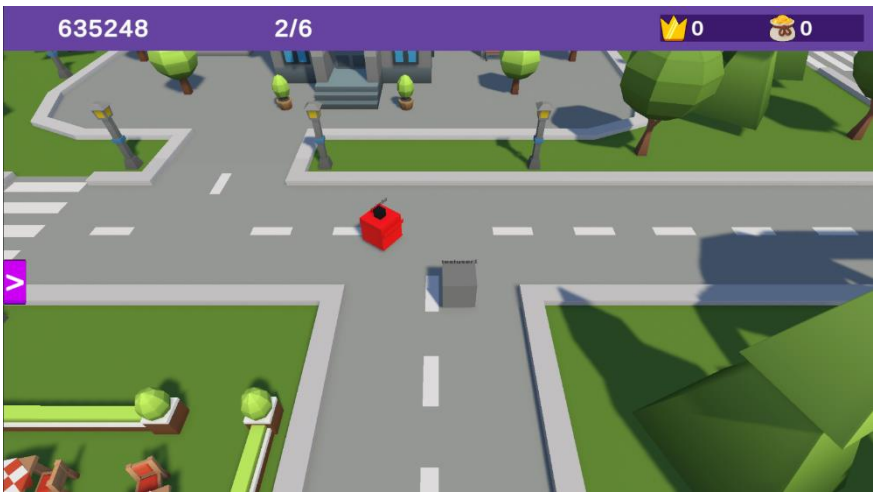


Рис. 4.3 Гравці у лобі

4.2.3 UC3: Виконання ходу

Кроки тестування:

1. Гравець натискає “Кинути кубик”.
2. Виконується рух фішки відповідно до випадкового числа.
3. Якщо фішка зупиняється на активі, відкривається діалог покупки/оренди.
4. Після підтвердження — надсилається подія Photon PUN 2 усім клієнтам.

Очікуваний результат:

- Фішка рухається коректно, відображаються відповідні повідомлення.
- Перелік клієнтів оновлюється одразу ж після завершення ходу.

Фактичний результат: Успіх, гра генерує потрібні події і рухає гравця, результат представлено на рисунку 4.4.

Рис. 4.4 Ігровий процес

4.2.4 UC4: Повернення до головного меню

Кроки тестування:

1. Під час гри натиснути кнопку “Меню”.
2. Перевірити, що відображається меню з пунктами “Вихід”(Див. Рисунок 4.5).

Очікуваний результат:

- Меню відкривається миттєво, без помітних затримок або візуальних артефактів.
- Всі пункти меню доступні для вибору.

Фактичний результат: усіп, гравець успішно повертається в меню і всі пункти меню працють коректно.

Рис. 4.5 Меню під час ігрового процесу

4.2.5 UC5: Налаштування зовнішнього вигляду фішки

Кроки тестування:

1. У головному меню обрати пункт “Скіни ”.
2. Вибрати новий образ фішки та натиснути “Застосувати”.
3. Вийти з налаштувань та запустити нову гру, щоб перевірити застосування скина.

Очікуваний результат:

- У вкладці “Скіни” відображаються всі доступні варіанти.
- Після натискання “Застосувати” обраний скин відразу використовується в наступній сесії.

Фактичний результат: успіх, візуальна частина фішки змінилася. І новий вигляд зразу застосувався і працює в новій сесії рисунок 4.6.

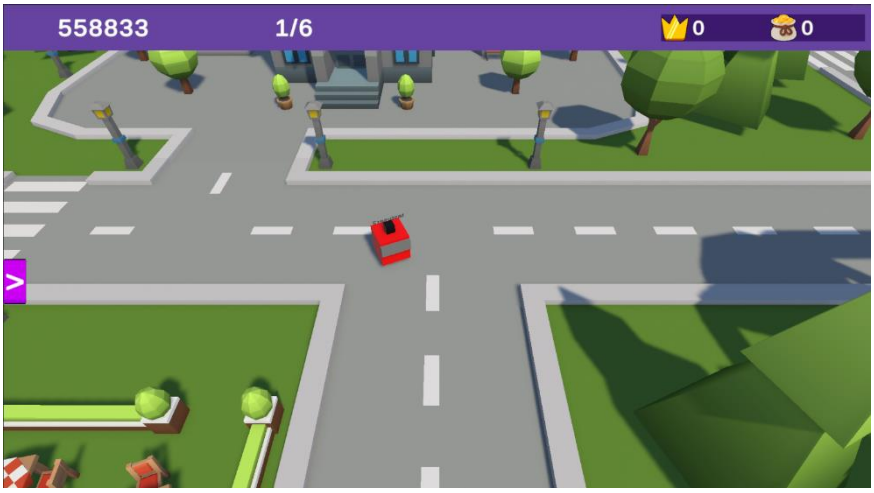


Рис. 4.6 Новий зовнішній вигляд фішки

4.2.6 UC6: Вихід з гри

Кроки тестування:

1. У головному меню обрати пункт “Вихід”.
2. Підтвердити діалог «Ви впевнені, що хочете вийти?»(Див. рисунок 4.7).
3. Дочекатися завершення процесу гри.

Очікуваний результат:

- З’являється підтверджувальний діалог без графічних артефактів.
- Після підтвердження гра закривається без помилок.

Фактичний результат: Успіх, меню виходу з’явилося, при підтверджені гра закривається без помилок.

Рис. 4.7 Виход з гри

4.3 Перевірка записів у Firestore

Під час тестування було підтверджено, що у Firestore успішно створюються такі типи документів:

Документ користувача. Після реєстрації гравця в колекції з'являється запис із полями Login, Nickname, Hash, Salt, Iterations, Gold, CountWinSessions, можна побачити на рисунку 4.8.

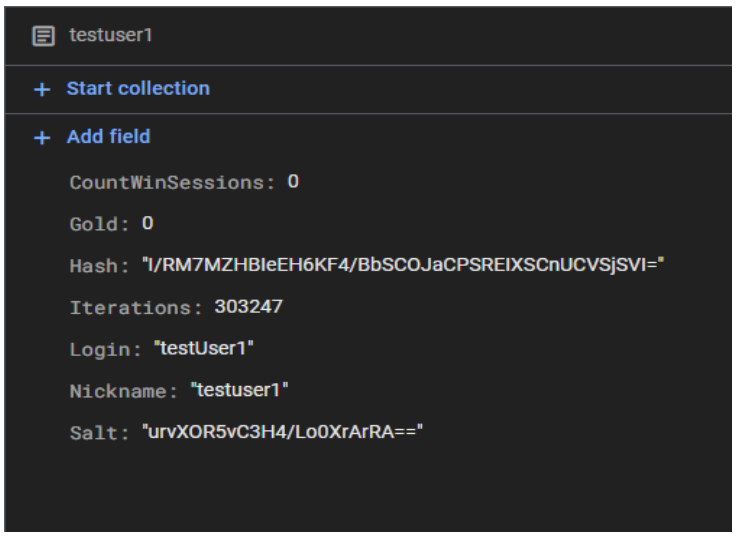


Рис. 4.8 Файл користувача у FireStore

Документ аналітики. Під час гри збираються дані про мережну затримку (ping) й зберігаються у колекції. Кожен документ містить поля UserName, SessionId, Ping, FPS, DateTime (див. рисунок 4.9).

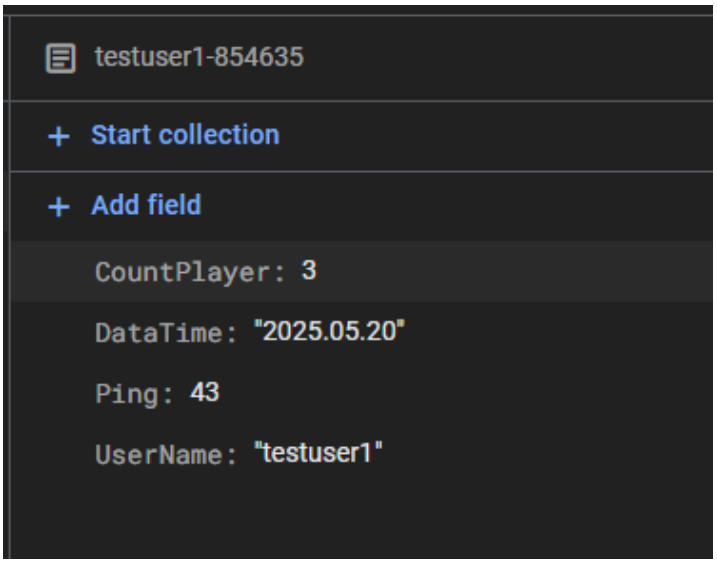


Рис. 4.9 Аналітика гравця

Документ звіту про баг. Якщо гравець надсилає bug report, у колекції bugReports створюється документ із полями TitleBug, DescriptionBug, UserName та DateTime, приклад репорту можна переглянути на рисунку 4.10.

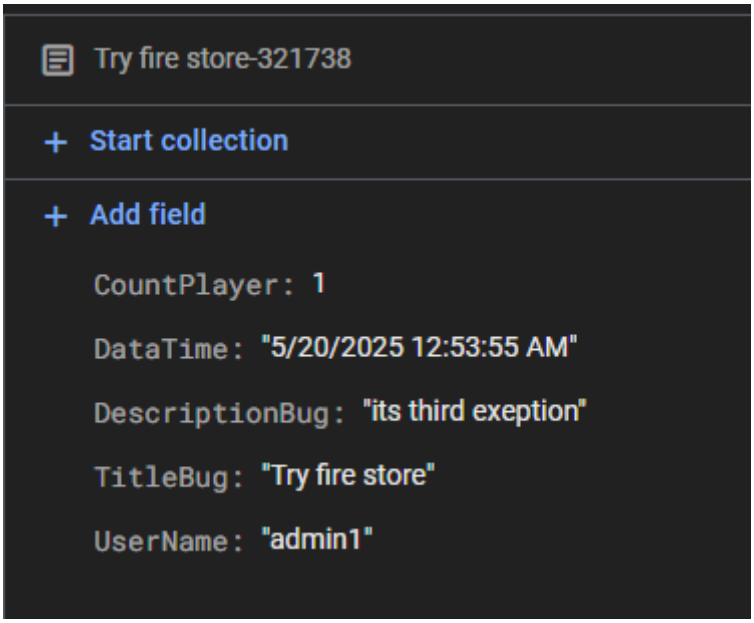


Рис. 4.10 Bug Report

Висновок:

Тестування ключових сценаріїв реєстрації та входу, створення й приєднання до ігрової сесії, виконання ходів із кидком кубика, мережної синхронізації, повернення до меню з налаштуванням скина та виходу з гри засвідчило стабільність і надійність роботи додатку за навантаженням при трьох гравцях, без розбіжностей у стані між клієнтами й із затримками ≤ 100 мс, водночас коректне збереження даних аналітики та звітів про помилки в Firestore підтвердило відповідність функціональних і нефункціональних вимог; для підвищення загальної продуктивності та покращення користувацького досвіду рекомендовано оптимізувати мережеві виклики та посилити обробку помилок у процесі запису аналітики.

ВИСНОВКИ

У результаті виконаної роботи створено інтерактивну мультиплеєрну покрокову економічну гру на рушії Unity із мережевою взаємодією через Photon PUN 2 та хмарним збереженням даних у Firebase Firestore, що дозволяє гейміфіковано моделювати реальні фінансові механізми й підвищувати фінансову грамотність користувачів.

Грунтовний аналіз жанрових аналогів, зокрема Monopoly, Pummel Party та Cashflow, дав змогу виокремити ключові переваги та недоліки наявних рішень і сформувати чіткі вимоги до функціональності (від хешування паролів PBKDF2 до підтримки багатомовного інтерфейсу) та продуктивності (затримки синхронізації не більше 100 мс, кросплатформеність Windows/macOS).

Обґрунтування архітектурних рішень із використання патернів Dependency Injection та MVP у поєднанні з Photon PUN 2 і Firebase Firestore забезпечило високу модульність, тестованість і можливість масштабування проєкту.

Прототип клієнтської частини на Unity реалізує всі ключові ігрові механіки — кидок кубика, рух фішки, купівлю активів і виплату оренди — з коректною синхронізацією між клієнтами й динамічним інтерфейсом завдяки TextMesh Pro та Cinemachine.

Ретельне функціональне та продуктивне тестування підтвердило стабільність роботи сценаріїв реєстрації, формування лобі і здійснення ходів із затримками до 100 мс, а також надійне збереження даних у Firestore із формуванням звітів про помилки. Таким чином, досягнуто мети роботи — створено інструмент для ефективного навчання фінансовим процесам у форматі інтерактивної гри.

Робота пройшла апробацію. За її результатами опубліковано тези доповідей:

1. Цуман О.С., Дібрівний О.А. Визначення бібліотеки для реалізації мережевої взаємодії в покроковій онлайн-грі на Unity. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.2025, ДУІКТ, Київ, с. 222–223.

2. Цуман О.С., Дібрівний О.А. Використання програмного патерну model–view–presenter (MVP) в ігровому проєкті на Unity. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.2025, ДУІКТ, Київ, с. 224–225.

3. Цуман О.С. Визначення вимог безпеки даних гравців у покроковій онлайн-грі. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.2025, КСУ імені Бориса Грінченка, Київ, с. 337–338.

Апробація підтвердила актуальність і практичну значущість розробленого програмного забезпечення, що дозволяє рекомендувати його для використання в освітніх та тренінгових програмах із підвищення фінансової грамотності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Unity Technologies. Unity User Manual (Long Term Support). Version 2020.3 LTS. Unity Technologies, 2020. URL: https://docs.unity3d.com/Manual/
2. Microsoft. C# Programming Guide. Microsoft Docs. Microsoft Corporation, 2021. URL: https://docs.microsoft.com/dotnet/csharp/
3. Exit Games GmbH. Photon PUN 2 *SDK* Documentation. Photon Engine, 2021. URL: https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro
4. Google LLC. Cloud Firestore Documentation. Firebase, 2021. URL: https://firebase.google.com/docs/firestore
5. Unity Technologies. TextMesh Pro Manual. Unity Asset Store, 2019. URL: https://docs.unity3d.com/Packages/com.unity.textmeshpro@2.1/manual/index.html
6. Unity Technologies. Cinemachine Manual. Unity Asset Store, 2020. URL: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.6/manual/index.html
7. Fowler, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley, 2003. – 552 p.
8. Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 3rd ed. Prentice Hall, 2004. – 576 p.
9. Martin, Robert C. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002. – 472 p.
10. Object Management Group. Unified Modeling Language (UML*), Version 2.5*. OMG, 2015. URL: https://www.omg.org/spec/UML/2.5/
11. Hasbro, Inc. Monopoly – Official Rules. Hasbro, 2018. URL: https://www.hasbro.com/monopoly
12. Rebuilt Games. Pummel Party [videogame]. Rebuilt Games, 2019. URL: https://www.pummelparty.com
13. Kiyosaki, Robert T. Cashflow Quadrant: Rich Dad's Guide to Financial Freedom. Warner Business Books, 1998. – 240 p.

14. Berserk Games. Tabletop Simulator [videogame]. Berserk Games, 2015. URL: https://store.steampowered.com/app/286160/Tabletop_Simulator/

15. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. – 395 p.

16. Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, 1996. – 544 p.

17. Richter, Jeffrey. *CLR via C#*. 4th ed. Microsoft Press, 2012. – 1000 p.

18. OMG. Systems Modeling Language (SysML) v1.6 – Specification. 2019. URL: https://www.omg.org/spec/SysML/1.6/

19. Petrillo, F., Lombardi, M., & Apperley, T. "Gamification in Financial Education: A Systematic Literature Review". Journal of Educational Technology & Society, 23(4), 2020, pp. 12–25.

20. Deterding, Sebastian; Dixon, Dan; Khaled, Rilla; Nacke, Lennart. "From Game Design Elements to Gamefulness: Defining 'Gamification'". Proceedings of the 15th International Academic MindTrek Conference, 2011, pp. 9–15.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка кооперативної гри в жанрі економічної стратегії

Виконав студент 4 курсу
групи ПД-43
Цуман Олександр Сергійович
Керівник роботи
Доктор філософії (PhD), доцент кафедри ІПЗ Дібрівний Олесь Андрійович

Київ – 2025

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проаналізувати жанрові аналоги (Monopoly, Pummel Party).
2. Сформувати функціональні та нефункціональні вимоги гри.
3. Вибрати архітектурний підхід, патерни і мережеву бібліотеку.
4. Спроєктувати геймплейний цикл, UI та схему авторизації.
5. Реалізувати гру на C# / Unity з підтримкою мультиплеєра.
6. Провести тестування й балансування ігрових механік.
7. Оцінити відповідність ПЗ визначеним вимогам.

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** – підвищити рівень фінансової грамотності користувачів шляхом моделювання реальних економічних механізмів у гейміфікованому кооперативному середовищі на рушії Unity.
- **Об'єкт дослідження** – процес взаємодії гравців і механізми прийняття стратегічних економічних рішень у мультиплеєрних покрокових іграх.
- **Предмет дослідження** – розробка покрокової кооперативної економічної стратегії на Unity, що симулює реальні економічні механізми.

АНАЛІЗ АНАЛОГІВ

Критерій	**Monopoly**	**Pummel Party**	**Cashflow**	**Tabletop Simulator**	**Grift Together**
Кількість гравців	2–6	2–8	2–6	довільна (залежить від модифікацій)	2–6
Тип сесії	Офлайн, нескінченні раунди	Онлайн, фіксовані раунди + міні-ігри	Онлайн, раунди до цілі	Онлайн, залежить від гри	Онлайн, гнучкі раунди
Автоматизація розрахунків	Відсутня	Повна	Часткова	Відсутня/часткова	Повна
Глибина фінмеханік	Базова	Відсутня	Середня	Залежить від гри	Базвова
Навчальний потенціал	Низький	Середній	Високий	Середній	Високий
Платформа	Настільна	ПК (Steam)	Онлайн	ПК (Steam)	Крос-платформова (Windows, macOS)
Навантаження на систему	Відсутнє	Високе	Мінімальне	Мінімальне-високе	Оптимізоване

ВИМОГИ ДО ДОДАТКУ

Функціональні вимоги:

- Реєстрація та авторизація гравців.
- Збереження та відновлення прогресу гравця.
- Створення/підключення до ігрової кімнати.
- Налаштування черги ходів гравців.
- Синхронізація стану карти та економічних дій.
- Механізм кастомізації та налаштувань.

Нефункціональні вимоги:

- Паролі гравців хешуються.
- Мережеві взаємодії забезпечують затримку не більше 100 мс при трьох гравцях.
- Гра працює на Windows та macOS.
- Інтерфейс доступний українською та англійською мовами.

КОНЦЕПТ ГРИ

GriftTogether — це покрокова кооперативна економічна стратегія, де гравці змагаються за прибутковість активів у світі. Гра моделює сучасні фінансові механізми (інвестиції, криптобіржі) та вчить приймати стратегічні рішення в колективному ігровому середовищі.

- Сеттинг: сучасне урбаністичне середовище з економічною варіативністю.
- Платформи: Windows, macOS.
- Керування: мишка та клавіатура

Цільова аудиторія: люди віком від 12 років, які зацікавлені в кооперативних симуляторах, любителі настільних ігор, та друзям для гри в онлайн-ком’юніті.

ГЕЙМПЛЕЙ ТА МЕХАНІКИ

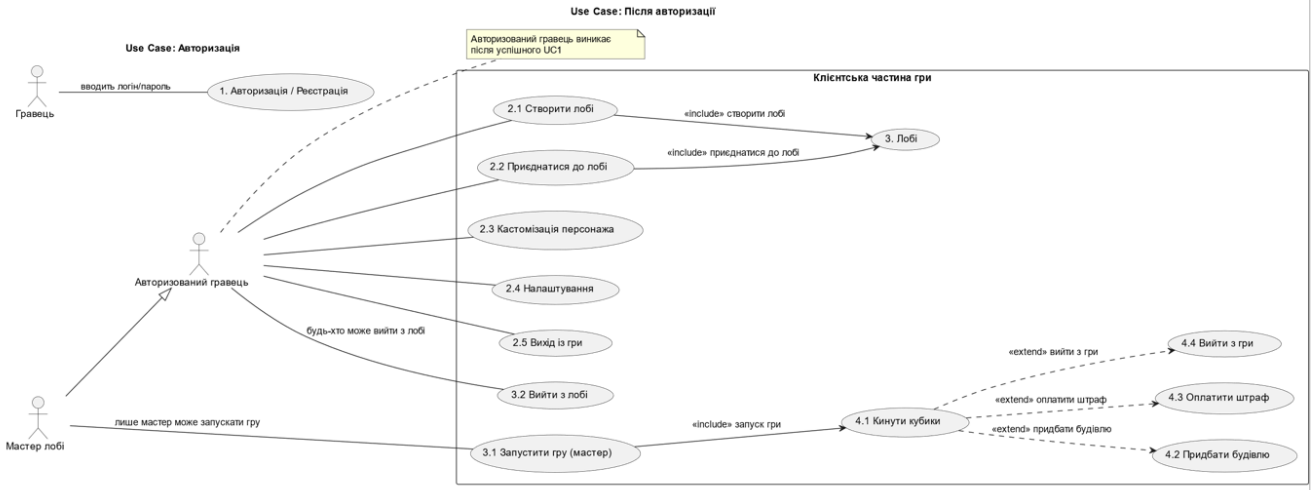
Геймплей: Гравці кидають кубик, рухаються по мапі, купують/продають/оплачують, чекають хід іншого.

Ігрові механіки:

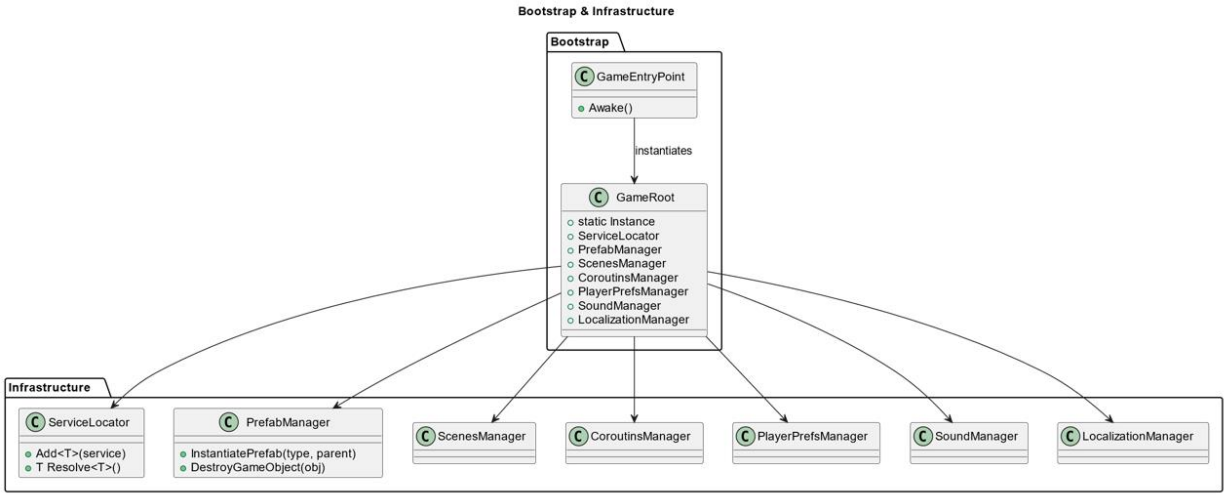
- Динамічні активи, які змінюють свої властивості від різних факторів(як приклад: множення тарифів, котре змінюється за умовами гри). (USP)
- Купівля активів, виплата оренди, біржі(USP), банки(USP).
- Кастомізація зовнішнього вигляду гравців.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

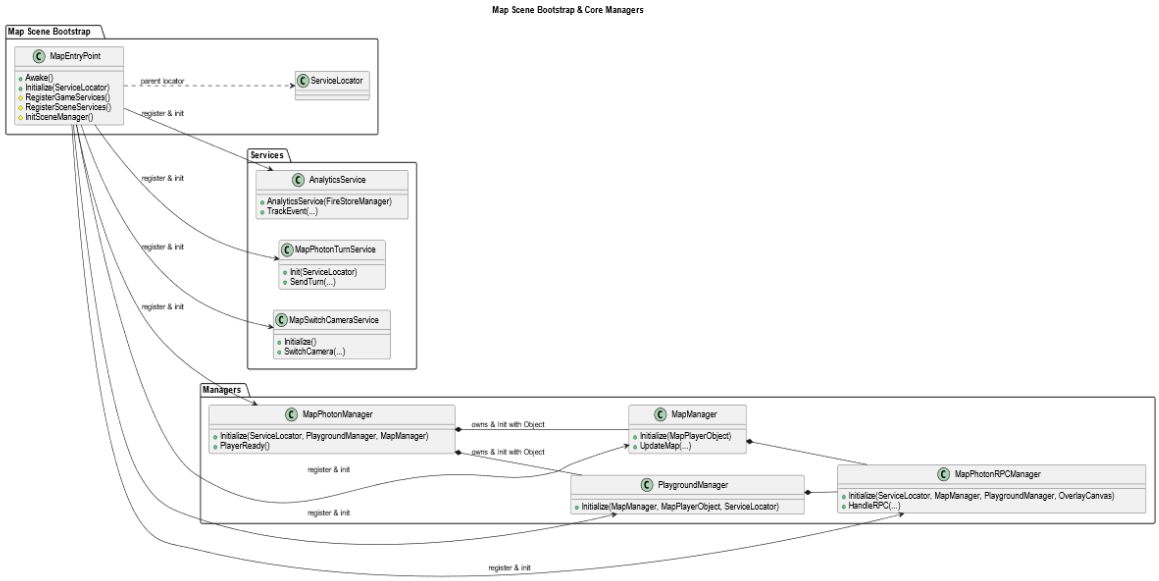
ДІАГРАМА ВИКОРИСТАННЯ



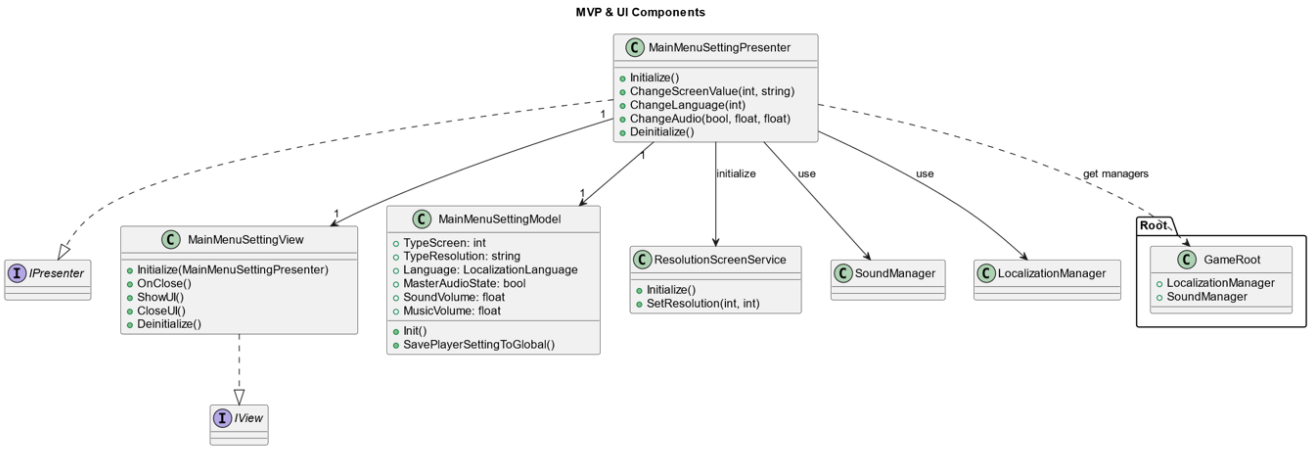
ДІАГРАМА КЛАСІВ



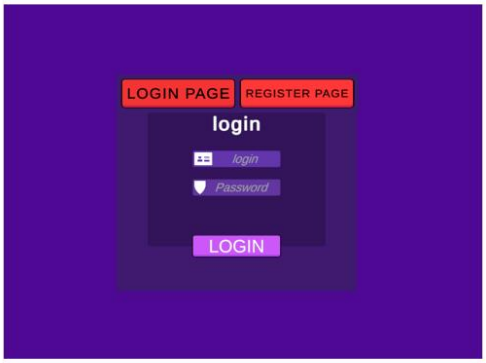
ДІАГРАМА КЛАСІВ



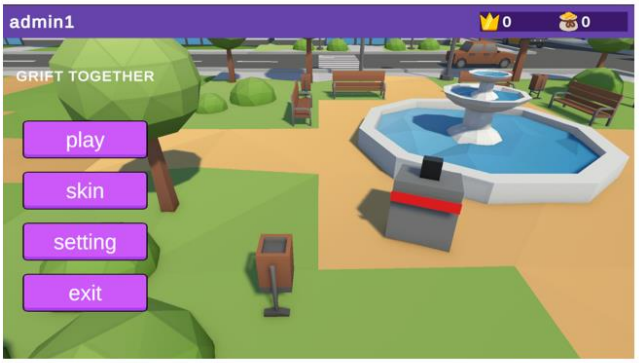
ДІАГРАМА КЛАСІВ



ЕКРАННІ ФОРМИ



Авторизація



Головне меню

ЕКРАННІ ФОРМИ

Лобі

Ігрове поле

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Цуман О.С., Дібрівний О.А., Визначення бібліотеки для реалізації мережевої взаємодії в покроковій онлайн-грі на Unity. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, с.222-223.

2. Цуман О.С., Дібрівний О.А., Використання програмного патерну model–view–presenter (MVP) в ігровому проєкті на Unity. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, с.224-225.

3. Цуман О. С., Визначення вимог безпеки даних гравців у покроковій онлайн-грі. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с. 337-338.

ВИСНОВКИ

1. Проаналізовані традиційні покрокові економічні ігри (Monopoly, Pummel Party, Cashflow) та виокремлені їхні сильні й слабкі сторони в частині балансування економіки й мережевої взаємодії.
2. Сформовані функціональні та нефункціональні вимоги гри, що охоплюють хешування паролів, затримки мережевої взаємодії не більше ніж 100 мс при трьох гравцях, крос-платформеність (Windows, macOS) та підтримку української й англійської мов інтерфейсу.
3. Вибрано архітектурний підхід із використанням Dependency Injection та патерну MVP, а також мережеву бібліотеку Photon PUN 2 для синхронізації стану гри між клієнтами.
4. Спроєктовано геймплейний цикл, інтерфейс користувача та схему авторизації з хешуванням паролів і захищеним обміном даних через Firestore.
5. Реалізовано мультиплеєрну покрокову економічну стратегію на рушії Unity за допомогою C#, що забезпечує синхронізацію подій, ходів та оновлення позицій ігрових фішок у реальному часі.
6. Проведено тестування й балансування ігрових механік, для виявлення й виправлення критичних проблем.
7. Оцінено відповідність програмного забезпечення визначеним функціональним та нефункціональним вимогам, підтверджено ефективність моделювання реальних економічних механізмів у кооперативному середовищі.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Вихідний код файлу GameEntryPoint

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace GriftTogether {

public class GameEntryPoint {



[RuntimeInitializeOnLoadMethod(RuntimeInitializ
eLoadType.BeforeSceneLoad)]
public static void StartEntryGame() {
new GameEntryPoint().RunGame();
}


private void RunGame() {

BaseUiSystem();
SaveSystem();

GameRoot.ServiceLocator = new
ServiceLocator(null);

SettingSystem();

GameRoot.FireStoreManager = new
FireStoreManager();

GameRoot.PhotonManager = new
GameObject("[PHOTON_MANAGER]").AddCom
ponent<PhotonManager>();
Object.DontDestroyOnLoad(GameRoot.PhotonMa
nager);

GameRoot.ScenesManager.SwitchScene(ScenesMa
nagerConst.LOGIN_SCENE, true);

}

private void BaseUiSystem() {
GameRoot.PrefabManager = new PrefabManager();

GameRoot.CoroutinsManager = new
GameObject("[COROUTINS]").AddComponent<C
oroutinsManager>();
Object.DontDestroyOnLoad(GameRoot.Coroutins
Manager);

GameRoot.ScenesManager = new
ScenesManager();
GameRoot.ScenesManager.ShowLoadingScreen();
}

private void SaveSystem() {
GameRoot.PlayerPrefsManager = new
PlayerPrefsManager();
GameRoot.PlayerGlobalManager = new
PlayerGlobalManager();
GameRoot.PlayerGlobalManager.LoadPPSetting();
}

private void SettingSystem() {

ResolutionScreenService screen = new
ResolutionScreenService();
screen.SetScreenType((FullScreenMode)GameRoot
.PlayerGlobalManager.GetScreenMode);
screen.TrySetScreenSize(GameRoot.PlayerGlobal
Manager.GetResolution);

GameRoot.SoundManager = new
SoundManager(GameRoot.PlayerGlobalManager.G
etMasterSoundState,
GameRoot.PlayerGlobalManager.GetVolumeSoun
d,
GameRoot.PlayerGlobalManager.GetVolumeMusic
);

GameRoot.CoroutinsManager.OnStartUnity +=
LateUpdate;

GameRoot.LocalizationManager = new
LocalizationManager(GameRoot.PlayerGlobalMan
ager.GetLanguage);

GameRoot.ServiceLocator.AddService(screen);
}

private void LateUpdate() {

GameRoot.SoundManager.SetSetting(GameRoot.Pl
ayerGlobalManager.GetMasterSoundState,
GameRoot.PlayerGlobalManager.GetVolumeSoun
d,
GameRoot.PlayerGlobalManager.GetVolumeMusic
);
}
}
}
```

Вихідний код файлй MainMenuManager

```
using UnityEngine;

namespace GriftTogether {
public class MainMenuManager : BaseSceneManager {

private GameObject _mainCanvas;
private GameObject _overlayCanvas;
private ServiceLocator _serviceLocator;

private MainMenuRootUIView _mainMenuRootUIView;
private MainMenuHeaderPresenter _mainMenuHeaderPresenter;
private MainMenuPresenter _mainMenuPresenter;

public MainMenuManager(GameObject mainCanvas, GameObject overlayCanvas, ServiceLocator serviceLocator) {
_mainCanvas = mainCanvas;
_overlayCanvas = overlayCanvas;
_serviceLocator = serviceLocator;
}

public override void Initialize() {
_mainMenuRootUIView = GameRoot.PrefabManager.InstantiatePrefab(MainMenuPrefabType.RootView, _mainCanvas).GetComponent<MainMenuRootUIView>();

_mainMenuHeaderPresenter = new MainMenuHeaderPresenter(_mainMenuRootUIView.GetHeaderParent);
_mainMenuHeaderPresenter.Initialize();

_mainMenuPresenter = new MainMenuPresenter(_mainMenuRootUIView.GetLeftCornerParent, _overlayCanvas, _serviceLocator);
_mainMenuPresenter.Initialize();
}

public override void Deinitialize() {
_mainMenuPresenter.Deinitialize();
_mainMenuHeaderPresenter.Deinitialize();
GameRoot.PrefabManager.DestroyGameObject(_mainMenuRootUIView.gameObject);
}
}
}
```

Вихідний код файлу MainMenuPresenter

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

namespace GriftTogether {

public class MainMenuPresenter : IPresenter {

private GameObject _root;
private GameObject _overlayRoot;

private ServiceLocator _serviceLocator;
private SkinService _skinService;

private MainMenuView _view;

private Dictionary<string, IPresenter> _cashPresenters;

public MainMenuPresenter(GameObject root, GameObject overlayRoot, ServiceLocator serviceLocator) {
_root = root;
_overlayRoot = overlayRoot;
_serviceLocator = serviceLocator;
_serviceLocator.Resolve(out _skinService);
}

public void Initialize() {
_view = GameRoot.PrefabManager.InstantiatePrefab(MainMenuPrefabType.MainView, _root).GetComponent<MainMenuView>();
_cashPresenters = new Dictionary<string, IPresenter>();
_view.Initialize(this);
}

public void PlayUI() {
if (TryShowUI<MainMenuPlayPresenter>()) return;

MainMenuPlayPresenter presenter = new MainMenuPlayPresenter(_root);
presenter.Initialize();
presenter.OnBack += BackMainMenu;
_cashPresenters.Add(typeof(MainMenuPlayPresenter).Name, presenter);

TryShowUI<MainMenuPlayPresenter>();
}

public void SkinUI() {
if (TryShowUI<MainMenuSkinPresenter>()) return;

MainMenuSkinPresenter presenter = new MainMenuSkinPresenter(_root, _skinService);
presenter.Initialize();
presenter.OnBack += BackMainMenu;
_cashPresenters.Add(typeof(MainMenuSkinPresenter).Name, presenter);

if (TryShowUI<MainMenuSkinPresenter>()) return;
```

```
    }

    public void SettingUI() {
    if (TryShowUI<MainMenuSettingPresenter>())
return;

    MainMenuSettingPresenter presenter = new
MainMenuSettingPresenter(_root);
    presenter.Initialize();
    presenter.onBack += BackMainMenu;
    _cashPresenters.Add(typeof(MainMenuSettingPrese
nter).Name, presenter);

    TryShowUI<MainMenuSettingPresenter>();
    }

    public void ExitUI() {

    if (TryShowUI<MainMenuExitPresenter>()) {
    this.ShowUI();
    return;
    }

    MainMenuExitPresenter presenter = new
MainMenuExitPresenter(_overlayRoot);
    presenter.Initialize();
    presenter.OnBack += BackMainMenu;
    _cashPresenters.Add(typeof(MainMenuExitPresente
r).Name, presenter);

    TryShowUI<MainMenuExitPresenter>();
    this.ShowUI();
    }

    private bool TryShowUI<T>() where T : IPresenter {

    string key = typeof(T).Name;

    if(_cashPresenters.ContainsKey(key)) {

    if (_cashPresenters.TryGetValue(key, out IPresenter
presenter)) {

    foreach(var item in _cashPresenters) {
    item.Value.CloseUI();
    }

    presenter.ShowUI();
    this.CloseUI();
    return true;
    }
    }

    return false;
    }

    private void BackMainMenu() {
    foreach (var item in _cashPresenters) {
    item.Value.CloseUI();
    }

    this.ShowUI();
    }

    public void ShowUI() => _view.ShowUI();

    public void CloseUI() => _view.CloseUI();

    public void Deinitialize() {
    foreach (var item in _cashPresenters) {
    item.Value.Deinitialize();
    }

    _view.Deinitialize();
    GameRoot.PrefabManager.DestroyGameObject(_vi
ew.gameObject);
    }
    }

    }
```

Вихідний код файлу MapPhotonManager

```
using ExitGames.Client.Photon;
using Photon.Pun;
using Photon.Realtime;
using System;
using UnityEngine;

namespace GriftTogether {

public class MapPhotonManager :
MonoBehaviourPunCallbacks {


    private ServiceLocator _serviceLocator;
    private PlaygroundManager _playground;
    private MapManager _mapManager;

    private bool _leaveGame;

    public MapPlayerObject Initialize(ServiceLocator
serviceLocator, PlaygroundManager playground,
MapManager mapManager) {
    _serviceLocator = serviceLocator;

    _playground = playground;
    _mapManager = mapManager;

    _leaveGame = false;

    return CreateLocalPlayer();
    }

    private MapPlayerObject CreateLocalPlayer() {
    GameObject player =
PhotonNetwork.Instantiate(PhotonPrefabConst.MAP_RP
C_AGENT_PREFAB_PATH, Vector3.zero,
Quaternion.identity);
    MapPhotonRPCService rpcService =
player.GetComponent<MapPhotonRPCService>();
```

```
    _serviceLocator.AddService(rpcService);

    GameRoot.ServiceLocator.Resolve(out
SkinPhotonService slinService);
    int indexPlayer =
Array.IndexOf(PhotonNetwork.PlayerList,
PhotonNetwork.LocalPlayer);
    MapPlayerObject mapPlayer =
slinService.CreatePhotonMapPlayer(PhotonNetwork.Loc
alPlayer, _playground.GetStartRoundPos(indexPlayer));

    return mapPlayer;
    }

    public void PlayerReady() {
    Hashtable props = new Hashtable { {
PhotonManagerConst.PLAYER_STAY_GAME, true } };
    PhotonNetwork.LocalPlayer.SetCustomProperties(p
rops);
    }

    public override void
OnPlayerPropertiesUpdate(Player targetPlayer,
Hashtable changedProps) {
    if
(changedProps.ContainsKey(PhotonManagerConst.PLA
YER_STAY_GAME)) CheckAllReady();
    }

    private void CheckAllReady() {

    var players = PhotonNetwork.PlayerList;

    int readyCount = 0;
    foreach (var p in players) {
    if
(p.CustomProperties.TryGetValue(PhotonManagerConst
.PLAYER_STAY_GAME, out var v) && (bool)v)
    readyCount++;
    }

    if (readyCount ==
PhotonNetwork.CurrentRoom.PlayerCount) {
    GameRoot.ScenesManager.HideLoadingScreen();
    _mapManager.StartGame();
    }
    }

    public void LeaveGame() =>_leaveGame = true;

    public override void OnLeftRoom() {
    base.OnLeftRoom();

    if (_leaveGame)
GameRoot.ScenesManager.SwitchScene(ScenesManager
Const.MENU_SCENE, true);
    }

    }
    }
```

Вихідний код файлу MapPhotonRPCService

```
    using Photon.Pun;
    using System.Threading.Tasks;
    using UnityEngine;

    namespace GriftTogether {

    public class MapPhotonRPCService :
MonoBehaviour, IService {

    private PhotonView _view;
    private MapPhotonRPCManager _manager;

    private void Start() {
    Initialize();
    }

    public void Initialize() {

    _view = GetComponent<PhotonView>();

    MapPhotonRPCContext context =
GameRoot.PhotonManager.CurrentPhotonContext as
MapPhotonRPCContext;

    if (context != null) {
    _manager = context.GetRPCManage;

    } else {
    Debug.LogError("CRITICAL ERROR! Can't
research MapPhotonRPCContext!");
    return;
    }
    }


    public void RPC_SendNextTurn() {
    _view.RPC(nameof(RPC_GetNextTurn),RpcTarget.
All);
    }

    [PunRPC]
    public void RPC_GetNextTurn() {
    _manager.GetNextTurn();
    }


    public void RPC_SendLog(string message) {
    _view.RPC(nameof(RPC_GetLog), RpcTarget.All,
message);
    }

    [PunRPC]
    public void RPC_GetLog(string message) {
    _manager.SpawnFadeLog(message);
    }


    public void RPC_BuyBuild(string message, string
indeficator, int playerIndex) {
```

```
    _view.RPC(nameof(RPC_GetBuyBuild),
RpcTarget.All, message, identifier, playerIndex);
    }

    [PunRPC]
    public void RPC_GetBuyBuild(string message,
string indeficator, int playerIndex) {
    _manager.BuyBuild(indefinator, playerIndex);
    _manager.SpawnFadeLog(message);
    }


    public void RPC_Rent(string message, string
indeficator) {
    _view.RPC(nameof(RPC_GetRent), RpcTarget.All,
message, identifier);
    }

    [PunRPC]
    public void RPC_GetRent(string message, string
indefictor) {
    _manager.PayRent(defector);
    _manager.SpawnFadeLog(message);
    }

    public void RPC_RemoveBuild(string message,
string indeficator, int playerIndex) {
    _view.RPC(nameof(RPC_GetRemoveBuild),
RpcTarget.All, message, identifier, playerIndex);
    }

    [PunRPC]
    public void RPC_GetRemoveBuild(string message,
string indeficator, int playerIndex) {
    _manager.RemoveOwner(indefinator, playerIndex);
    _manager.SpawnFadeLog(message);
    }
    }
    }
```