МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ІНДІ-ГРИ “FLAMEL” З ВИКОРИСТАННЯМ ІГРОВОГО РУШІЯ UNITY ТА C#»**

Виконав: студент 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
А.С. Шапка
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Мильцев О.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри комп’ютерних наук, к.т.н. Добровольський Г.А.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Шапці Андрію Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інді-гри “Flamel” з використанням ігрового рушія Unity та C#

керівник роботи Мильцев Олександр Михайлович, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Проєктування гри.
2. Реалізація основних підсистем.
3. Тестування та оцінка результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	19.12.2025	
2.	Збір вихідних даних.	15.01.2026	
3.	Обробка методичних та теоретичних	17.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	13.04.2026	
5.	Розробка третього розділу.	20.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	16.06.2026	

Студент _____ А.С. Шапка
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.М. Мильцев
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка інді-гри “Flamel” з використанням ігрового рушія Unity та C#»: 58 с., 27 рис., 6 джерел, 5 додатків.

АЛХІМІЯ, ІГРОВИЙ РУШІЙ, ІНДІ-ГРА, ІНТЕРАКТИВНИЙ ІНТЕРФЕЙС, МОВА ПРОГРАМУВАННЯ C#, ПРОЄКТУВАННЯ ІГОР.

Об’єкт дослідження – процес розробки інді-гри жанру симулятора алхіміка за використанням ігрового рушія Unity та мови програмування C#.

Мета роботи: спроєктувати та реалізувати гру що поєднує діагностику, приготування зілля, ігрову економіку, систему прогресії та збереження ігрового стану.

Методи дослідження – аналіз предметної області, порівняння ігрових механік, об’єктно-орієнтоване проєктування, моделювання архітектури програмної системи, програмна реалізація та функціональне тестування.

Програмну реалізацію виконано засобами Unity з використанням мови C#. Для побудови логіки гри використано компонентний підхід, окремі контролери підсистем, сервіси керування станом гри та структури даних для опису пацієнтів, хвороб, симптомів, рецептів, зіль, інвентарю та прогресу. Для реалізації користувацького інтерфейсу застосовано стандартні UI-засоби Unity та TextMeshPro, а для збереження ігрового стану – серіалізацію даних у формат JSON.

У результаті створено цілісний програмний продукт, який демонструє можливість використання Unity та C# для розробки інді-гри з кількома взаємопов’язаними підсистемами. Практичне значення роботи полягає у створенні прототипу гри, що може слугувати основою для подальшого розширення функціоналу, ускладнення ігрової логіки та вдосконалення візуальної й технічної складової.

SUMMARY

Bachelor’s qualifying paper «Indie Game “Flamel” Development Using the Unity Game Engine and C#»: 58 pages, 27 figures, 6 references, 5 supplements.

ALCHEMY, GAME ENGINE, INDIE GAME, INTERACTIVE INTERFACE, C# PROGRAMMING LANGUAGE, GAME DESIGN.

The object of research is the process of developing an indie game of the alchemist simulator genre using the Unity game engine and the C# programming language.

The aim of the study is to design and implement a game using a combination of diagnostics, potion making, game economy, progression system, and game state saving.

The methods of research are analysis of the subject area, comparison of game mechanics, object-oriented design, modeling of the architecture of the software system, software implementation, and functional testing.

The software implementation was performed using Unity tools using the C# language. To build the game logic, a component approach was used, separate subsystem controllers, game state management services, and data structures for describing patients, diseases, symptoms, recipes, potions, inventory, and progress. Standard Unity and TextMeshPro UI tools were used to implement the user interface, and data serialization in JSON format was used to save the game state.

As a result, a complete software product was created that demonstrates the possibility of using Unity and C# to develop an indie game with several interconnected subsystems. The practical significance of the work lies in creating a game prototype that can serve as the basis for further expanding the functionality, complicating the game logic, and improving the visual and technical components.

ЗМІСТ

Завдання на кваліфікаційну роботу..... 2
Реферат 4
Summary 5
Вступ 8
1 Проєктування гри 10
1.1 Концепція гри, жанрові особливості та ігровий цикл 10
1.2 Очікуваний результат розробки 11
1.3 Проєктування структури даних гри 12
1.4 ER-модель 13
1.5 Проєктування архітектури програмної системи 13
1.6 Аналіз функціональності та сценаріїв використання 17
1.7 Аналіз прецедентів 18
1.8 Sequence Diagram 20
1.9 Class Diagram 21
1.10 Діаграма активності ігрового циклу 21
1.11 Діаграма компонентів 23
1.12 Діаграма розгортання 24
2 Реалізація основних підсистем гри «Flamel» 26
2.1 Реалізація системи генерації пацієнтів і відображення їхніх характеристик 26
2.2 Реалізація механіки діагностики захворювань 30
2.3 Реалізація алхімічної системи створення зіль 32
2.4 Реалізація системи лікування пацієнтів 32
2.5 Реалізація інвентарю та внутрішньоігрової економіки 33
2.6 Реалізація системи покращень майстерні 34
2.7 Реалізація журналу знань 35
2.8 Реалізація механізму збереження та завантаження гри 36

3 Тестування та оцінка результатів розробки 38
3.1 Організація процесу тестування 38
3.2 Аналіз основного інтерфейсу гри 38
3.3 Аналіз допоміжних вікон та модулів 39
3.4 Результати тестування основних підсистем 42
3.5 Аналіз можливостей подальшого покращення проєкту 43
Висновки 44
Перелік посилань 45
Додаток А Логіка запуску та виконання алхімічної сесії 46
Додаток Б Структура роботи процесу лікування 50
Додаток В Структура TreatmentResolver 53
Додаток Г Структура зміни ресурсів в інвентарі 55
Додаток Д Формування пропозицій та купівлі інгредієнтів 57

ВСТУП

Індустрія відеоігор є однією з найдинамічніших галузей сучасної IT-сфери, що поєднує програмну інженерію, комп'ютерну графіку, дизайн взаємодії, штучний інтелект і мультимедійні технології. Упродовж останніх років особливої популярності набув сегмент інді-ігор, який характеризується незалежністю розробників, творчим підходом до ігрових механік, гнучкістю у прийнятті технічних рішень та можливістю створення конкурентоспроможних продуктів невеликими командами або навіть окремими розробниками. Саме інді-ігри часто стають платформою для реалізації оригінальних ідей, експериментів із жанрами та створення нестандартного ігрового досвіду.

Актуальність теми зумовлена тим, що розробка сучасної комп'ютерної гри є комплексним завданням, яке вимагає не лише створення візуального контенту, а й побудови цілісної програмної архітектури, реалізації ігрової логіки, систем взаємодії користувача з інтерфейсом, збереження та відновлення стану гри, керування внутрішньоігровими ресурсами, а також забезпечення зручності подальшого розширення функціоналу. Ігровий рушій Unity у поєднанні з мовою програмування C# є одним із найбільш поширених і ефективних інструментів для створення інді-проєктів завдяки кросплатформеності, наявності розвиненої екосистеми, підтримці 2D- та 3D-графіки, а також зручними засобами для побудови компонентної архітектури застосунку.

У межах кваліфікаційної роботи розглядається розробка інді-гри «Flamel», яка реалізує ігровий процес у середньовічному антуражі. Концепція гри ґрунтується на поєднанні елементів симулятора, рольової взаємодії та менеджменту ресурсів. Гравець виступає в ролі алхіміка-цілителя, до якого звертаються персонажі з різними симптомами та захворюваннями. Основу ігрового процесу становлять аналіз симптомів, встановлення діагнозу, приготування лікувального зілля за визначеним рецептом, застосування

лікування, отримання винагороди, розвиток майстерні та поступове відкриття нових можливостей. Такий підхід дозволяє поєднати програмну складову з продуманою системою ігрових механік, що робить тему роботи практично значущою та цікавою з точки зору програмної інженерії.

Особливість розроблюваної гри полягає у необхідності побудови взаємопов’язаних підсистем: системи генерації пацієнтів, системи діагностики, алхімічної системи створення зіль, системи інвентарю, внутрішньоігрової економіки, апгрейдів, журналу знань, інтерфейсу користувача, а також механізмів збереження та завантаження прогресу. Усе це потребує продуманої модульної структури програмного коду, яка забезпечує зручність підтримки, тестування та подальшої модернізації проєкту.

1 ПРОЄКТУВАННЯ ГРИ

1.1 Концепція гри, жанрові особливості та ігровий цикл

Кожен пацієнт має певний архетип, приховане захворювання та набір видимих симптомів. На основі цієї інформації гравець повинен обрати правильний діагноз. Далі необхідно скористатися алхімічною системою, вибрати рецепт та покроково виконати процес створення потрібного зілля. Результат лікування залежить від правильності діагнозу, відповідності зілля захворюванню та якості його приготування [1].

Основний ігровий цикл гри можна подати у вигляді такої послідовності: поява пацієнта, аналіз симптомів, вибір діагнозу, підготовка зілля, лікування пацієнта, отримання результату та перехід до наступного випадку. Після успішного завершення лікування гравець отримує внутрішньоігрову винагороду у вигляді золота та репутації. Накопичені ресурси можуть бути використані для придбання інгредієнтів, відкриття нових рецептів та покращення майстерні.

Гра має виражену атмосферичну складову, побудовану навколо середньовічної алхімічної майстерні. Візуальне оформлення, моделі пацієнтів, стилізація інтерфейсу та система знань у вигляді журналу підтримують загальну концепцію проєкту.

З точки зору програмної реалізації, концепція гри передбачає поділ системи на окремі функціональні підсистеми: систему пацієнтів,діагностики, алхімії, лікування, інвентарю, економіки, покращень, журналу знань, збереження та інтерфейсу. Такий підхід дозволяє будувати програмний продукт модульно та забезпечує зручність подальшого розширення функціоналу [6].

1.2 Очікуваний результат розробки

Кінцевим результатом роботи має стати працездатний прототип інді-гри «Flamel», розроблений засобами Unity та C#, який реалізує цілісний ігровий цикл і забезпечує взаємодію гравця з основними підсистемами гри [2].

До основних функціональних вимог розроблюваного програмного продукту належать:

– генерація пацієнтів із визначеним архетипом, симптомами та захворюванням;
– відображення симптомів пацієнта та можливість вибору діагнозу;
– реалізація алхімічної системи з вибором рецепта та покроковим приготуванням зілля;
– застосування лікування з урахуванням правильності діагнозу, обраного зілля та якості його приготування;
– ведення обліку золота, репутації та внутрішньоігрових ресурсів;
– реалізація інвентарю для зберігання інгредієнтів;
– підтримка ринку для придбання ресурсів;
– реалізація системи покращень майстерні;
– ведення журналу знань про симптоми, хвороби та рецепти;
– забезпечення можливості збереження та завантаження прогресу гри;
– реалізація зручного користувацького інтерфейсу для доступу до всіх основних механік.

До нефункціональних вимог належать:

– зрозуміла та логічна структура інтерфейсу;
– модульна архітектура програмної системи;
– узгоджена робота окремих підсистем між собою;
– можливість подальшого розширення гри новими сутностями та механіками;
– стабільне виконання основних сценаріїв проходження;
– стилістична цілісність візуального оформлення.

Таким чином, у результаті розробки необхідно отримати програмний продукт, який не лише демонструє основні ігрові механіки, а й має цілісну архітектуру, придатну до подальшого розвитку, тестування та вдосконалення.

1.3 Проєктування структури даних гри

У структурі даних гри можна виділити кілька основних груп сутностей: пацієнти, симптоми та захворювання; інгредієнти, рецепти та зілля; інвентар, економіка та покращення. Такий поділ відповідає загальній логіці проєкту та відображає основні підсистеми, з якими взаємодіє гравець у процесі проходження [3].

До першої групи належать сутності, що описують пацієнта та його стан. Пацієнт у грі характеризується архетипом, захворюванням, набором видимих симптомів, а також поточним станом проходження лікування. Для цього використовується окрема структура, яка зберігає ідентифікатор конкретного пацієнта.

Друга група сутностей відповідає за алхімічну систему гри. До неї належать інгредієнти, рецепти, зілля та кроки приготування. Інгредієнт описує назву ресурсу, його базові властивості та спосіб використання в рецептах. Зілля виступає кінцевим результатом алхімічного процесу і містить інформацію про свою назву та лікувальний ефект. Рецепт пов’язує зілля з конкретним набором інгредієнтів і послідовністю дій, необхідних для його створення.

Третя група сутностей стосується керування ресурсами та прогресом гравця. Для цього використовуються структури, що описують інвентар, внутрішньоігрові пропозиції ринку, стан економіки та покращення майстерні. Інвентар зберігає інформацію про доступні ресурси, їхню кількість та обмеження місткості. Економічна частина включає значення золота, репутації, а також об’єкти, що описують ринкові пропозиції та вартість інгредієнтів.

Система покращень містить перелік можливих апгрейдів, їх вартість та вплив на ігровий процес.

1.4 ER-модель

ER-модель даних гри «Flamel» відображає основні сутності програмної системи та зв’язки між ними (див. рис. 1.1 та 1.2, стор. 14).

Центральними сутностями є пацієнт, архетип персонажа, захворювання, симптом, рецепт, інгредієнт, зілля, інвентар, ринкова пропозиція, покращення, журнал знань і стан ігрової сесії.

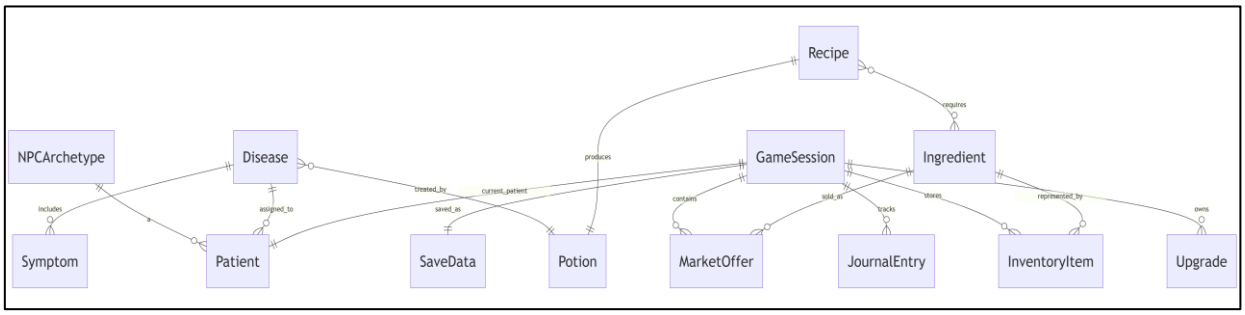


Рисунок 1.1 – Спрощена ER-модель

Побудована модель дозволяє формалізувати логічну структуру даних гри та слугує основою для подальшого проєктування класів і програмної архітектури.

1.5 Проєктування архітектури програмної системи

В основу програмної побудови гри покладено компонентний підхід, характерний для ігрового рушія Unity. Кожна ключова функціональна частина реалізована у вигляді окремих класів і контролерів, що взаємодіють між собою через чітко визначені зв’язки.

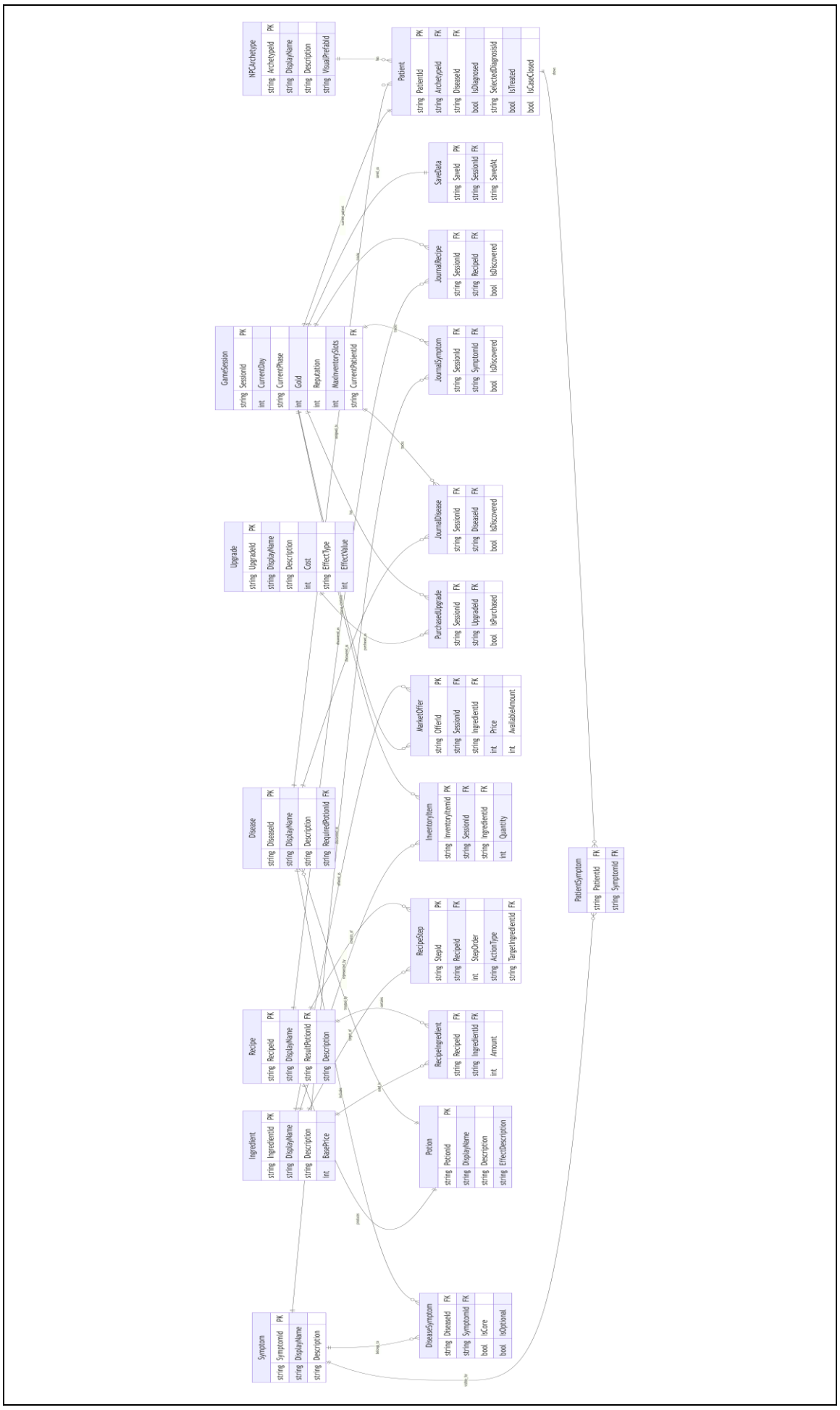


Рисунок 1.2 – Повна ER-модель

Такий підхід дозволяє ізолювати окремі механіки, уникати надмірної залежності між компонентами та забезпечувати кращу структурованість проєкту. При цьому кожен модуль відповідає за одну логічно завершену функцію: генерацію пацієнтів, оновлення інтерфейсу, роботу інвентарю, керування алхімічними сесіями або збереження прогресу гри [5].

У межах архітектури можна виділити кілька основних рівнів. *Перший* рівень становлять дані гри. *Другий* рівень утворюють логічні сервіси та менеджери, які працюють із цими даними та реалізують внутрішню бізнес-логіку гри. *Третій* рівень представлений контролерами інтерфейсу користувача, що забезпечують взаємодію гравця з ігровими системами та відображення поточного стану гри.

Центральне місце в архітектурі займає підсистема керування ігровим процесом, яка відповідає за послідовність зміни фаз гри. Для цього використовується контролер потоку дня, який визначає етапи взаємодії з пацієнтом.

Підсистема пацієнтів має бути реалізована через поєднання фабрики створення пацієнтів, контролера взаємодії з пацієнтом та компонента відображення моделі на сцені. Фабрика відповідає за створення нового пацієнта на основі доступних даних про архетипи, симптоми та захворювання. Контролер взаємодії виконує логіку прив’язки пацієнта до інтерфейсу, передає дані в панелі діагностики й лікування, а також координує перехід до наступного пацієнта. Окремий спавнер візуалізації відповідає за появу відповідної моделі персонажа на сцені залежно від архетипу пацієнта.

Підсистема діагностики повинна бути побудована таким чином, щоб ізолювати логіку вибору захворювання від логіки відображення симптомів. Панель діагностики відповідає за показ доступних симптомів і варіантів діагнозу, а також за фіксацію вибору гравця. Результат діагностики далі використовується системою лікування. Це дозволяє уникнути дублювання логіки й робить механіку діагнозу окремим завершеним модулем.

Алхімічна підсистема є однією з ключових у грі й реалізується через поєднання опису рецептів, контролера алхімії, сесії приготування та логіки

оцінювання результату. Контролер алхімії відповідає за вибір рецепта, запуск процесу варіння, виконання кроків і відображення поточного стану сесії в інтерфейсі. Дані рецепта зберігаються окремо від логіки виконання, що дозволяє легко розширювати гру новими рецептами без зміни базової архітектури системи. Якість приготування зілля та правильність виконаних дій передаються далі у систему лікування.

Підсистема лікування має об’єднувати вибраний діагноз, підготовлене зілля та характеристики поточного пацієнта. Окремий механізм обробки результату лікування оцінює, наскільки правильною була послідовність дій гравця, і повертає результат у вигляді зміни золота, репутації та текстового повідомлення. Це дозволяє зробити лікування самостійним логічним етапом із чітко визначеним входом і виходом.

Для керування ресурсами використаємо підсистему інвентарю та внутрішньоігрової економіки. Інвентарний менеджер зберігає інформацію про доступні інгредієнти, їх кількість та обмеження місткості. Окремі контролери відповідають за показ вмісту інвентарю, ринок, придбання ресурсів та покращення майстерні. Поділ на логічні сервіси та UI-контролери дозволяє відокремити обробку даних від їхнього представлення на екрані.

Окрему роль в архітектурі відіграють журнал знань і система збереження гри. Журнал має надати гравцеві доступ до довідкових даних через інтерфейс. Система збереження та завантаження відповідає за фіксацію поточного стану гри та його відновлення після повторного запуску. Ці підсистеми не залежать напряму від конкретної фази гри, але інтегруються з іншими модулями через спільні дані.

Архітектура інтерфейсу користувача повинна побудована на використанні окремих панелей, кожна з яких відповідає за власну функціональну область. Такий підхід дозволяє зручно організувати взаємодію користувача з грою та уникнути надмірного ускладнення окремих UI-компонентів.

1.6 Аналіз функціональності та сценаріїв використання

Основним користувачем системи є гравець, який взаємодіє з грою через інтерфейс користувача та послідовно проходить етапи ігрового циклу (див. рис. 1.3).

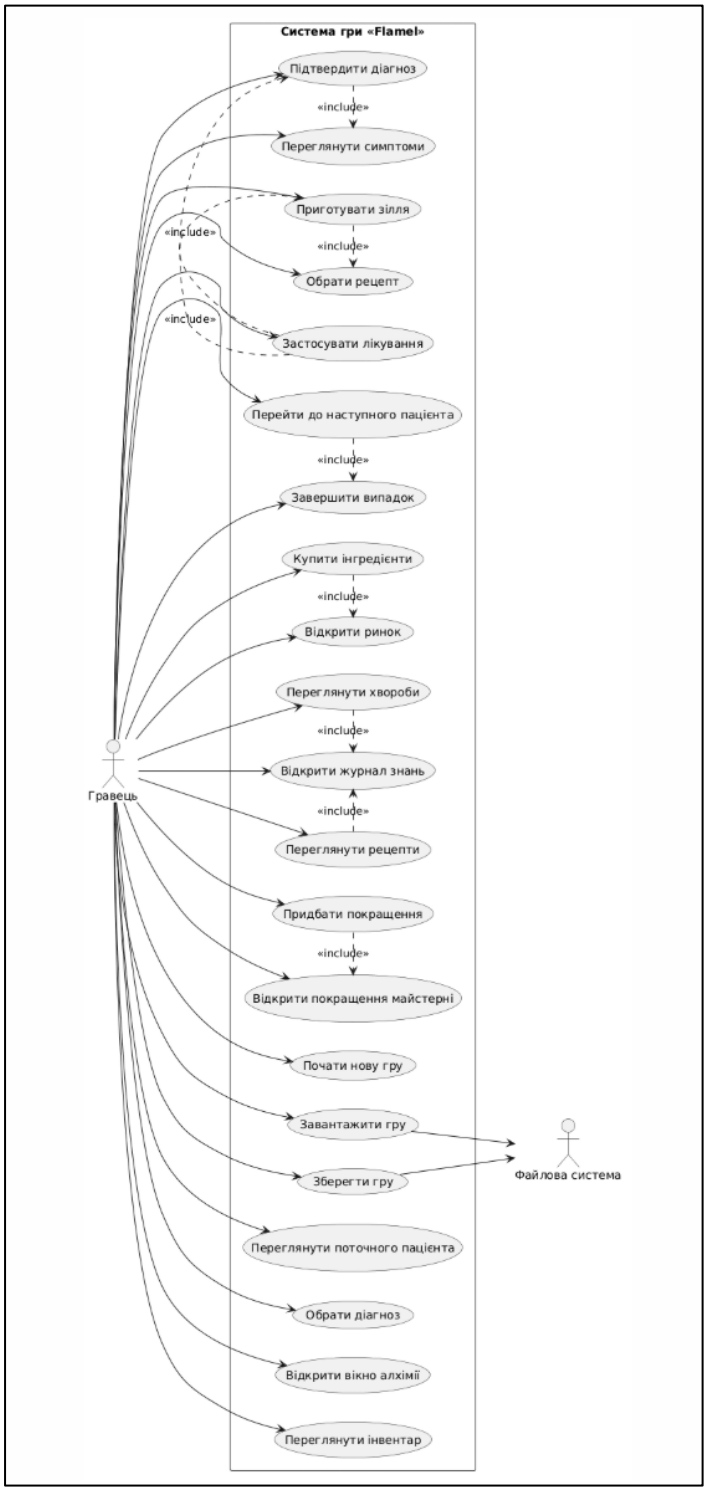


Рисунок 1.3 – Use Case Diagram

На відміну від класичних інформаційних систем, у грі сценарії використання будуються не навколо введення й обробки службових даних, а навколо взаємодії з ігровими механіками. Тому функціональність гри «Flamel» доцільно розглядати як набір взаємопов’язаних прецедентів, що забезпечують проходження від появи пацієнта до завершення лікування та подальшого розвитку майстерні.

До основних функціональних можливостей системи належать:

– початок нової гри;
– завантаження збереженого прогресу;
– генерація нового пацієнта;
– перегляд симптомів пацієнта;
– вибір і підтвердження діагнозу;
– відкриття алхімічного вікна та вибір рецепта;
– приготування зілля;
– застосування лікування до пацієнта;
– завершення поточного випадку та перехід до наступного пацієнта;
– перегляд інвентарю;
– придбання інгредієнтів на ринку;
– придбання покращень майстерні;
– перегляд журналу знань;
– збереження поточного стану гри.

1.7 Аналіз прецедентів

Для формалізації функціональних вимог до гри нижче представлено детальний аналіз прецедентів, що описують послідовність дій гравця та реакцію системи під час ключових геймплейних процесів.

Прецедент «Лікування пацієнта»:

– система створює нового пацієнта;

– гравець переглядає симптоми;
– гравець обирає та підтверджує діагноз;
– гравець відкриває алхімічне вікно;
– гравець обирає рецепт і готує зілля;
– гравець застосовує лікування;
– система обчислює результат;
– система оновлює ресурси та переводить гру до наступної фази.

Прецедент «Придбання інгредієнтів»:
– гравець відкриває ринок;
– система показує доступні інгредієнти та їхню вартість;
– гравець обирає товар;
– система перевіряє наявність золота та місця в інвентарі;
– система виконує покупку;
– інвентар і hud оновлюються.

Прецедент «Придбання покращення майстерні»:
– гравець відкриває вікно покращень;
– система показує список доступних апгрейдів;
– гравець обирає покращення;
– система перевіряє можливість придбання;
– система застосовує ефект покращення;
– оновлюються відповідні параметри гри.

Прецедент «Збереження гри»:
– гравець обирає команду збереження;
– система формує структуру збереження;
– дані записуються у файл;
– гравець отримує повідомлення про успішне збереження.

Прецедент «Завантаження гри»:
– гравець обирає команду завантаження;
– система читає файл збереження;
– система відновлює стан гри.

1.8 Sequence Diagram

Розглянемо діаграму послідовності між основними компонентами програмної системи гри «Flamel» у межах сценарію лікування пацієнта (див. рис. 1.4).

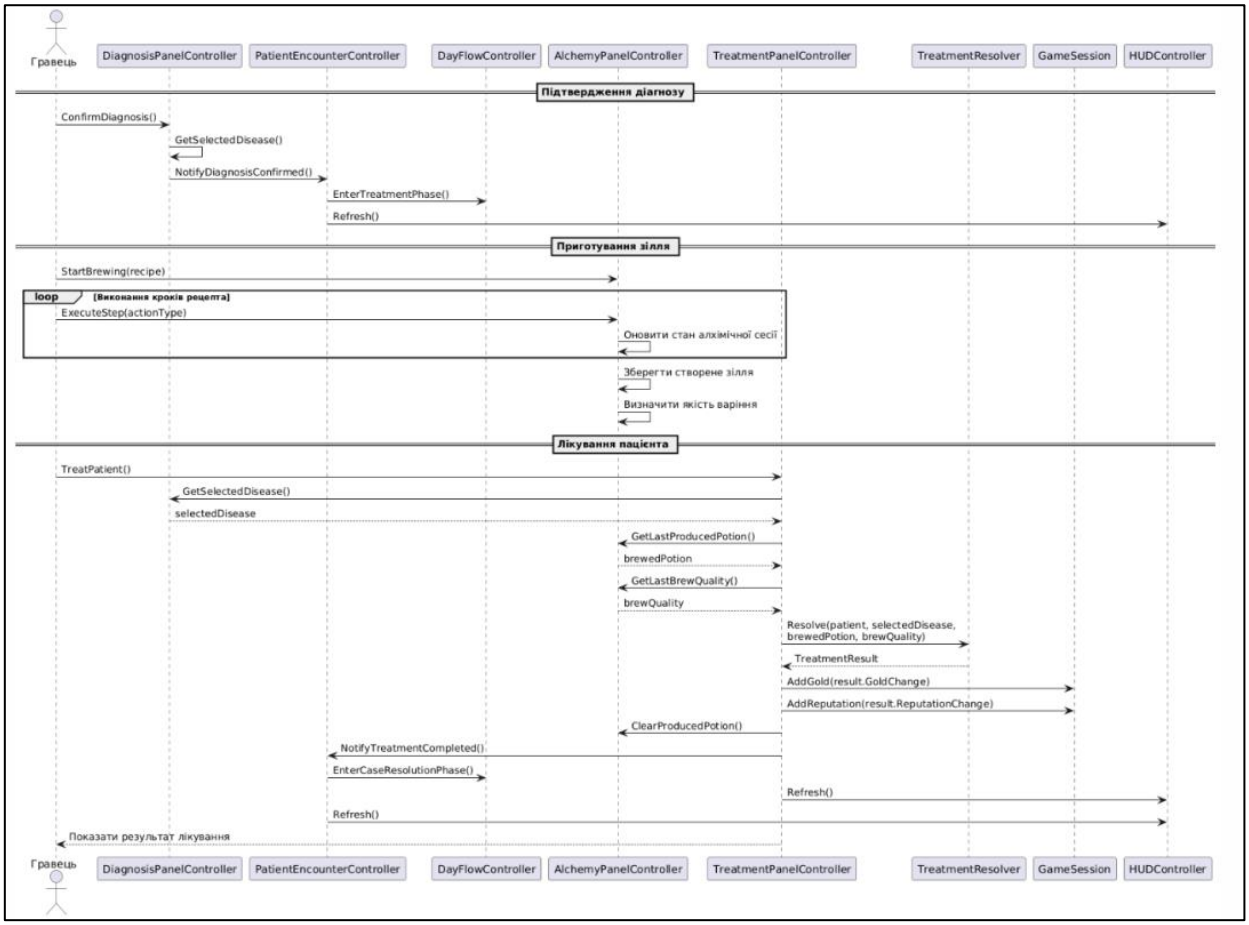


Рисунок 1.4 – Діаграма послідовності сценарію «Лікування пацієнта»

Спочатку гравець підтверджує діагноз, після чого система переводить гру до фази лікування. Далі гравець у підсистемі алхімії готує зілля, виконуючи послідовність кроків рецепта.

Під час лікування контролер лікування отримує обраний діагноз, приготовлене зілля та його якість, передає ці дані до компонента TreatmentResolver, який формує результат лікування. Після цього оновлюються параметри ігрової сесії, очищується результат алхімії, змінюється фаза гри та оновлюється інтерфейс користувача.

1.9 Class Diagram

UML-діаграма класів відображає основні сутності, сервіси, контролери та зв’язки між ними в програмній системі гри «Flamel» (див. рис. 1.5). На діаграмі представлено доменні класи, що описують ігрові сутності, зокрема пацієнтів, хвороби, симптоми, рецепти, зілля та інгредієнти, а також службові класи, які реалізують логіку генерації пацієнтів, діагностики, алхімії, лікування, керування ресурсами, журналу знань і механізму збереження гри.

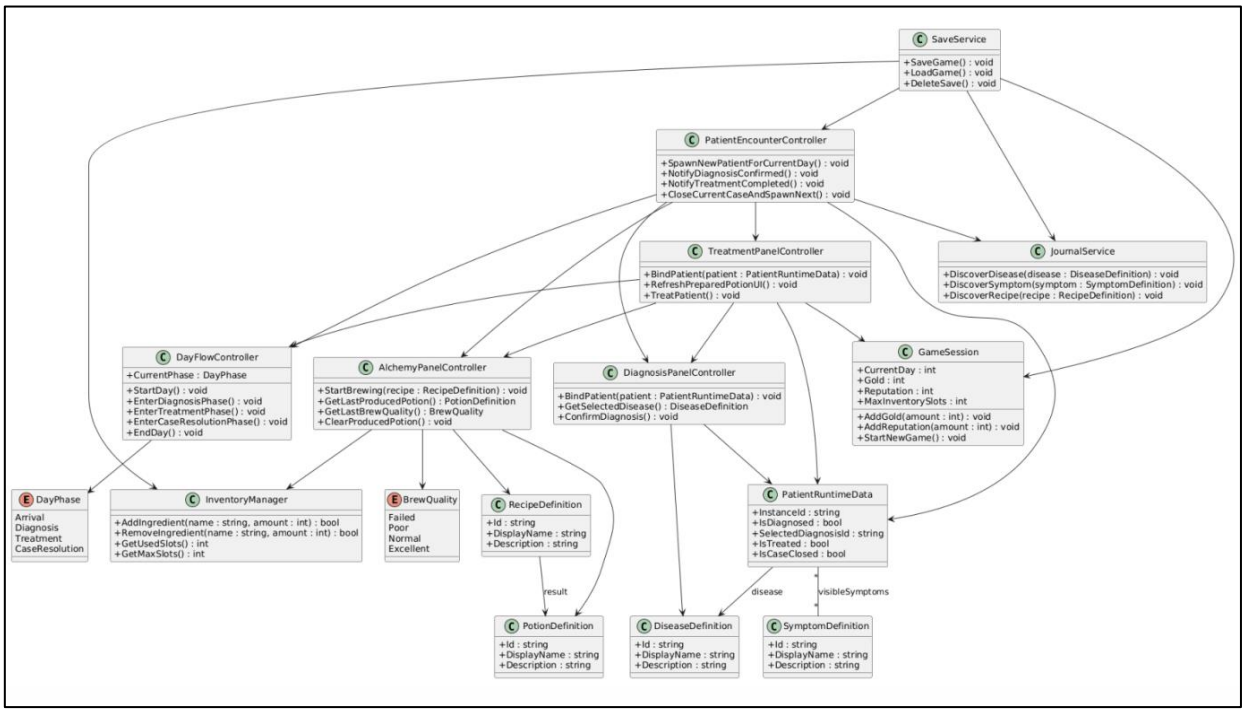


Рисунок 1.5 – Class Diagram

Побудована модель демонструє модульну структуру програмного продукту та взаємодію між його основними підсистемами.

1.10 Діаграма активності ігрового циклу

Діаграма активності відображає послідовність дій у межах основного ігрового циклу гри «Flamel» (див. рис. 1.6).

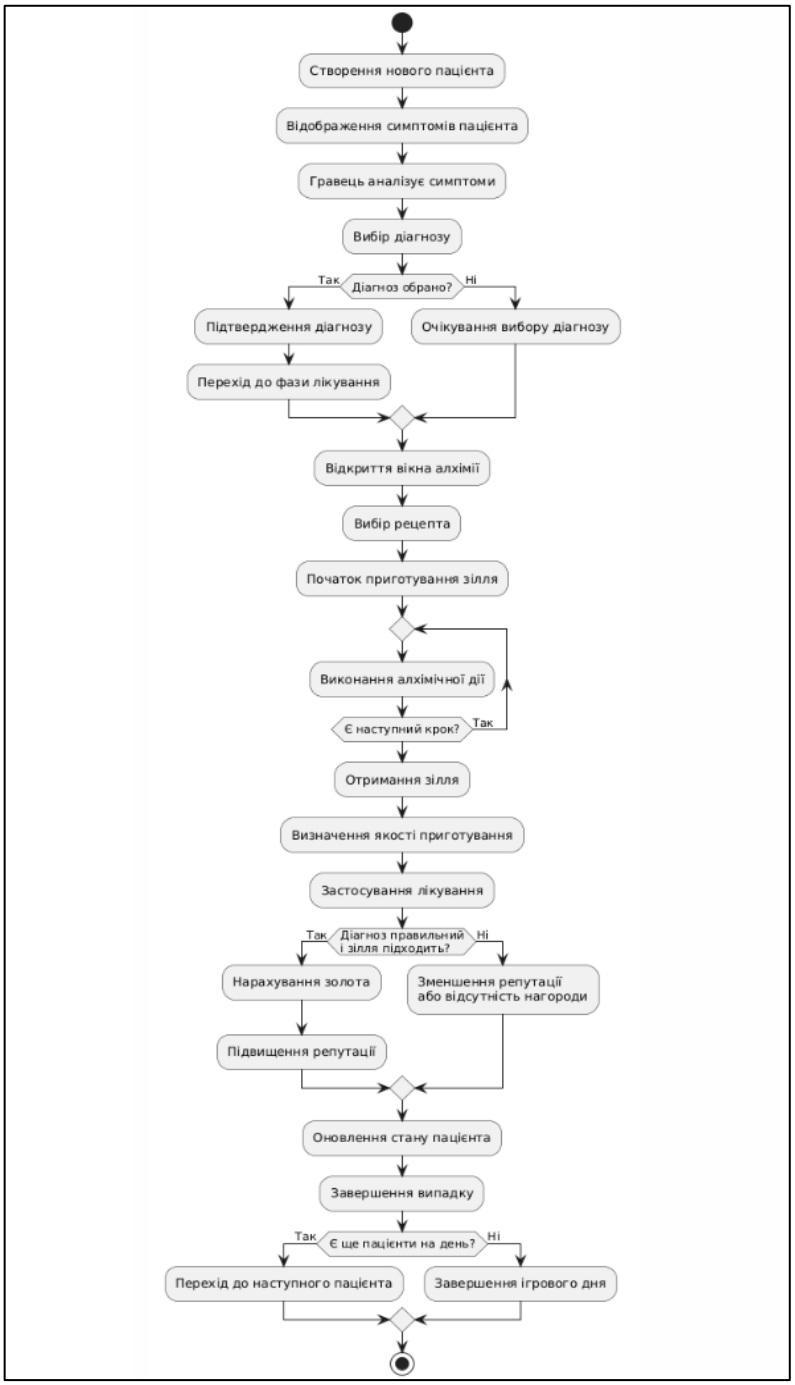


Рисунок 1.6 – Діаграма активності ігрового циклу

Процес починається зі створення нового пацієнта та відображення його симптомів. Далі гравець аналізує симптоми, обирає діагноз і переходить до алхімічної системи, де виконує послідовність дій для приготування зілля. Після цього відбувається лікування пацієнта, перевірка правильності дій гравця та оновлення ігрових параметрів, зокрема золота і репутації. Завершення сценарію супроводжується переходом до наступного пацієнта або завершенням ігрового дня.

1.11 Діаграма компонентів

Діаграма компонентів відображає модульну структуру програмної системи гри «Flamel» та взаємозв’язки між її основними підсистемами (див. рис. 1.7).

Центральним елементом взаємодії є користувацький інтерфейс, через який гравець отримує доступ до функціональних можливостей гри. Основні компоненти системи охоплюють підсистеми керування пацієнтами, діагностики, алхімії, лікування, інвентарю, ринку, покращень, журналу знань, збереження та завантаження гри, а також компоненти керування ігровою сесією і фазами дня.

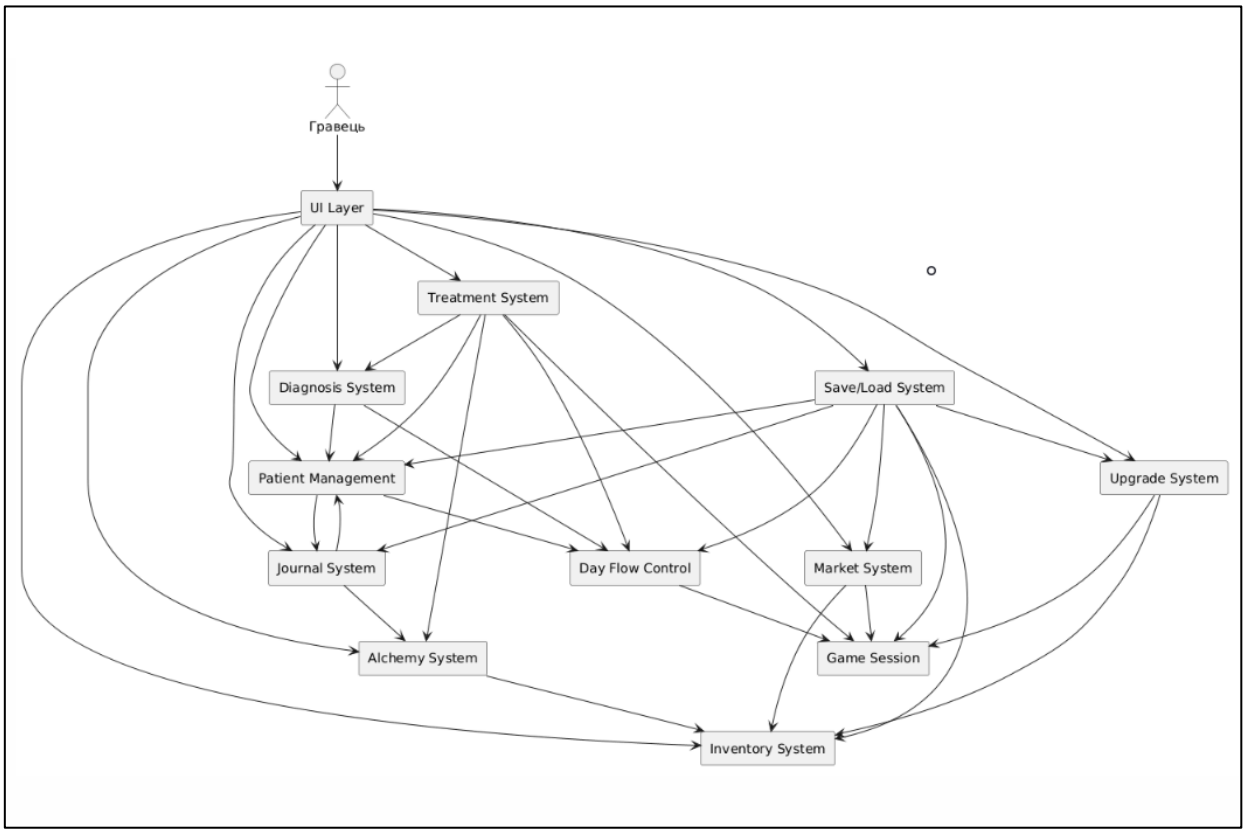


Рисунок 1.7 – Діаграма компонентів програмної системи гри «Flamel»

Побудована діаграма демонструє, що програмний продукт реалізовано за модульним принципом, а окремі компоненти взаємодіють між собою через чітко визначені зв’язки.

1.12 Діаграма розгортання

Діаграма розгортання відображає фізичну структуру розміщення програмного продукту «Flamel» та пов’язаних із ним програмних артефактів (див. рис. 1.8). Оскільки розроблена гра є локальним застосунком, її виконання відбувається на персональному комп’ютері користувача без використання окремого серверного середовища.

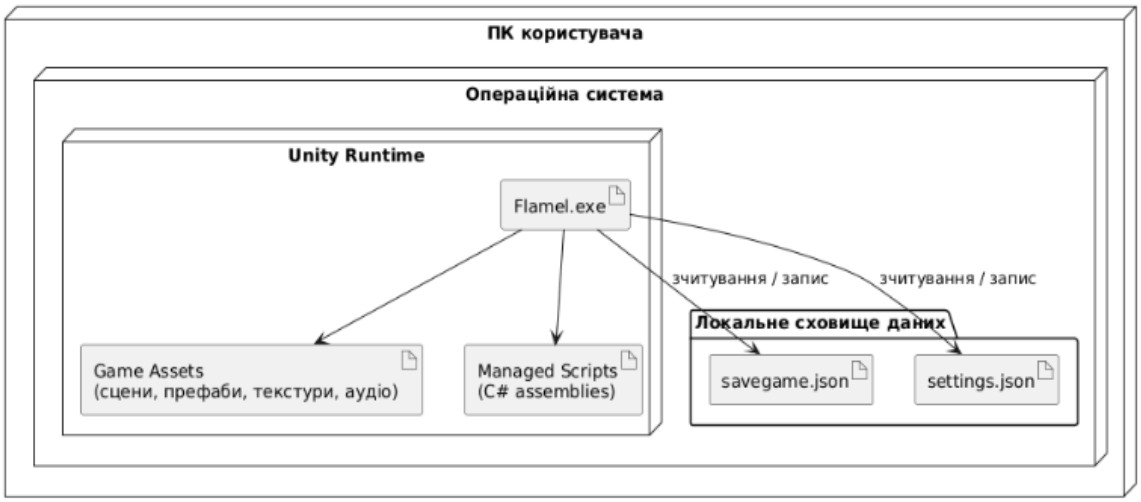


Рисунок 1.8 – Діаграма розгортання

Основним фізичним вузлом системи є персональний комп’ютер користувача, на якому встановлена операційна система та виконується зібраний застосунок гри. У межах цього вузла функціонує середовище Unity Runtime, яке забезпечує запуск і роботу ігрового продукту. Основним програмним артефактом є виконуваний файл Flamel.exe, що містить логіку запуску гри та забезпечує взаємодію з іншими компонентами [4].

Для коректної роботи застосунку використовуються також додаткові артефакти, зокрема ігрові ресурси (сцени, префаби, текстури, аудіофайли) та скомпільовані C#-скрипти, які реалізують основні механіки гри. Окремо в локальному сховищі користувача розміщуються файли savegame.json, що використовується для збереження ігрового прогресу, та settings.json, у якому можуть зберігатися параметри налаштувань гри.

Отже, діаграма демонструє, що програмний продукт є автономною локальною системою, яка виконується на комп’ютері користувача та використовує локальні файли для збереження даних і налаштувань.

2 РЕАЛІЗАЦІЯ ОСНОВНИХ ПІДСИСТЕМ ГРИ «FLAMEL»

У цьому розділі наведено детальний опис структури реалізації основних ігрових систем та їхньої взаємодії між собою і з гравцем.

2.1 Реалізація системи генерації пацієнтів і відображення їхніх характеристик

Для представлення поточного пацієнта використовується окрема структура PatientRuntimeData, яка містить у собі динамічний стан конкретного ігрового випадку (див. рис. 2.1).

```
using System.Collections.Generic;

[System.Serializable]
public class PatientRuntimeData
{
    public string InstanceId;
    public NPCArchetypeDefinition Archetype;
    public DiseaseDefinition Disease;

    public List<SymptomDefinition> VisibleSymptoms = new List<SymptomDefinition>();

    public bool IsDiagnosed;
    public string SelectedDiagnosisId;

    public bool IsTreated;
    public bool IsCaseClosed;
}
```

Рисунок 2.1 – Структура PatientRuntimeData

Створення нового пацієнта виконується за допомогою спеціального класу-фабрики PatientFactory (див. рис. 2.2). Його використання дозволяє централізовано формувати нового пацієнта на основі ігрової бази даних. Під час генерації обираються архетип пацієнта, захворювання та набір симптомів, що мають бути відображені гравцеві.

```
public PatientRuntimeData CreateRandomPatient()
{
    if (gameDatabase == null)
    {
        Debug.LogError("GameDatabase is null");
        return null;
    }

    if (gameDatabase.NPCArchetypes.Count == 0)
    {
        Debug.LogError("Database does not contain NPC archetypes");
        return null;
    }

    List<DiseaseDefinition> availableDiseases = GetAvailableDiseases();

    if (availableDiseases.Count == 0)
    {
        Debug.LogError("No available diseases found for current progression");
        return null;
    }

    NPCArchetypeDefinition randomArchetype =
        gameDatabase.NPCArchetypes[UnityEngine.Random.Range(0, gameDatabase.NPCArchetypes.Count)];

    DiseaseDefinition randomDisease =
        availableDiseases[UnityEngine.Random.Range(0, availableDiseases.Count)];

    PatientRuntimeData patient = new PatientRuntimeData
    {
        InstanceId = Guid.NewGuid().ToString(),
        Archetype = randomArchetype,
        Disease = randomDisease,
        IsDiagnosed = false,
        SelectedDiagnosisId = string.Empty,
        IsTreated = false,
        IsCaseClosed = false
    };

    List<SymptomDefinition> visibleSymptoms = new List<SymptomDefinition>();
    visibleSymptoms.AddRange(randomDisease.CoreSymptoms);

    int diagnosisBonus = 0;

    if (WorkshopUpgradeService.Instance != null)
    {
        diagnosisBonus = WorkshopUpgradeService.Instance.GetTotalDiagnosisBonus();
    }

    for (int i = 0; i < randomDisease.OptionalSymptoms.Count; i++)
    {
        bool includeOptional = UnityEngine.Random.value > 0.5f;

        if (i < diagnosisBonus)
        {
            includeOptional = true;
        }

        if (includeOptional)
        {
            visibleSymptoms.Add(randomDisease.OptionalSymptoms[i]);
        }
    }

    patient.VisibleSymptoms = visibleSymptoms;

    return patient;
}
```

Рисунок 2.2 – Структура створення пацієнта

Керування поточним пацієнтом у межах сцени виконує PatientEncounterController (див. рис. 2.3). Саме цей компонент відповідає за запуск створення нового пацієнта, передачу даних у відповідні панелі інтерфейсу та координацію переходів між фазами ігрового процесу. Після генерації нового пацієнта контролер передає його до панелі діагностики та панелі лікування, а також оновлює загальний стан інтерфейсу.

```
public void SpawnNewPatientForCurrentDay()
{
    if (dayFlowController == null)
    {
        Debug.LogError("DayFlowController is missing");
        return;
    }

    dayFlowController.EnterPatientArrivalPhase();

    currentPatient = patientFactory.CreateRandomPatient();

    if (JournalService.Instance != null && currentPatient != null)
    {
        JournalService.Instance.DiscoverDisease(currentPatient.Disease);

        for (int i = 0; i < currentPatient.VisibleSymptoms.Count; i++)
        {
            JournalService.Instance.DiscoverSymptom(currentPatient.VisibleSymptoms[i]);
        }
    }

    if (currentPatient == null)
    {
        Debug.LogError("Failed to create patient");
        return;
    }

    if (currentPatientView != null)
    {
        Destroy(currentPatientView.gameObject);
    }

    currentPatientView = patientSpawner.SpawnPatient(currentPatient);

    if (alchemyPanelController != null)
    {
        alchemyPanelController.ClearProducedPotion();
    }

    if (diagnosisPanelController != null)
    {
        diagnosisPanelController.BindPatient(currentPatient);
    }

    if (treatmentPanelController != null)
    {
        treatmentPanelController.BindPatient(currentPatient);
    }

    dayFlowController.EnterDiagnosisPhase();

    if (hudController != null)
    {
        hudController.Refresh();
    }
}
```

Рисунок 2.3 – Структура створення та відображення пацієнта

Для відображення пацієнта на сцені використовується окремий спавнер PatientSpawner (див. рис. 2.4). Його завдання полягає у створенні візуального представлення пацієнта на основі архетипу. Для цього реалізовано бібліотеку відповідності між типом пацієнта та prefab-моделлю, що дозволяє замість одного універсального об’єкта відображати різні моделі.

```
public PatientView SpawnPatient(PatientRuntimeData patient)
{
    ClearCurrentPatientVisual();

    if (patient == null)
    {
        Debug.LogWarning("SpawnPatient called with null patient");
        return null;
    }

    GameObject prefabToSpawn = null;

    string archetypeId = GetArchetypeId(patient);
    Debug.Log("PatientSpawner archetypeId = " + archetypeId);

    if (visualLibrary != null && !string.IsNullOrWhiteSpace(archetypeId))
    {
        prefabToSpawn = visualLibrary.GetPrefabByArchetypeId(archetypeId);
    }

    if (prefabToSpawn == null)
    {
        Debug.LogWarning("No prefab found for archetype: " + archetypeId + ". Using fallback.");
        prefabToSpawn = fallbackPrefab;
    }

    if (prefabToSpawn == null)
    {
        Debug.LogWarning("No patient visual prefab found and no fallback prefab assigned.");
        return null;
    }

    Transform targetSpawnPoint = spawnPoint != null ? spawnPoint : transform;

    currentSpawnedPatient = Instantiate(
        prefabToSpawn,
        targetSpawnPoint.position,
        targetSpawnPoint.rotation
    );

    currentSpawnedPatient.name = prefabToSpawn.name;

    PatientView patientView = currentSpawnedPatient.GetComponent<PatientView>();

    if (patientView == null)
    {
        patientView = currentSpawnedPatient.GetComponentInChildren<PatientView>();
    }

    return patientView;
}
```

Рисунок 2.4 – Структура PatientSpawner

Окремий компонент PatientView використовується для представлення моделі пацієнта та інтеграції її в загальну структуру сцени (див. рис. 2.5).

```
using UnityEngine;

public class PatientView : MonoBehaviour
{
   public PatientRuntimeData RuntimeData { get; private set; }

   public void Bind(PatientRuntimeData runtimeData)
   {
      RuntimeData = runtimeData;

      if (runtimeData != null && runtimeData.Archetype != null)
      {
         gameObject.name = "Patient_" + runtimeData.Archetype.DisplayName;
      }
   }
}
```

Рисунок 2.5 – Структура PatientView

Важливим аспектом реалізації є інтеграція системи пацієнтів із фазами проходження гри. Після створення нового пацієнта система переводить гру у фазу прибуття, а далі – у фазу діагностики. Це означає, що генерація нового пацієнта не є ізольованою функцією, а входить до загальної моделі керування ігровим процесом.

2.2 Реалізація механіки діагностики захворювань

Відображення даних, пов’язаних з діагностикою, реалізовано в окремому контролері інтерфейсу DiagnosisPanelController (див. рис. 2.6). Цей компонент відповідає за прив’язку поточного пацієнта до панелі діагностики, показ його симптомів, заповнення списку доступних захворювань і виведення результату вибору.

Після появи нового пацієнта дані автоматично передаються в цей контролер, який оновлює відповідні текстові поля та елементи вибору.

```
public void ConfirmDiagnosis()
{
    if (currentPatient == null)
    {
        Debug.LogError("No current patient assigned");
        return;
    }

    if (dayFlowController != null && dayFlowController.CurrentPhase != DayPhase.Diagnosis)
    {
        diagnosisResultText.text = "Result: diagnosis is not allowed in the current phase.";
        return;
    }

    if (patientEncounterController != null)
    {
        patientEncounterController.NotifyDiagnosisConfirmed();
    }

    int selectedIndex = diagnosisDropdown.value;

    if (selectedIndex < 0 || selectedIndex >= availableDiseases.Count)
    {
        Debug.LogError("Invalid selected diagnosis index");
        return;
    }

    DiseaseDefinition selectedDisease = availableDiseases[selectedIndex];
    DiagnosisResult result = diagnosisService.EvaluateDiagnosis(currentPatient, selectedDisease);

    if (result.IsCorrect)
    {
        diagnosisResultText.text = "Result: Correct diagnosis!\n";

        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayDiagnosisConfirm();
        }
    }
    else
    {
        diagnosisResultText.text = "Result: Wrong diagnosis!\nSelected: " + result.SelectedDiseaseName +
                                   "\n";
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayError();
        }
    }

    if (hudController != null)
    {
        hudController.Refresh();
    }
}
```

Рисунок 2.6 – Структура методу підтвердження діагнозу

2.3 Реалізація алхімічної системи створення зіль

Для реалізації процесу приготування зілля використовується окремий контролер алхімії AlchemyPanelController, який відповідає за взаємодію гравця з алхімічним вікном (див. додаток А). Цей компонент забезпечує відображення доступних рецептів, запуск нового процесу варіння, обробку дій гравця та оновлення інформації про поточний стан алхімічної сесії. Через нього гравець обирає рецепт, ініціює початок варіння та послідовно виконує алхімічні кроки.

2.4 Реалізація системи лікування пацієнтів

За взаємодію з користувачем відповідає TreatmentPanelController (див. додаток Б). Цей компонент приймає поточного пацієнта, відображає інформацію про підготовлене зілля, показує результат лікування та забезпечує запуск відповідної логіки після натискання кнопки застосування лікування. Контролер виконує попередню перевірку умов, за яких лікування може бути проведене.

Безпосередня оцінка лікування реалізована через окремий клас TreatmentResolver, який відповідає за логіку перевірки результату (див. додаток В).

У відповідь система отримує об’єкт TreatmentResult, який містить інформацію про успішність лікування, текстовий результат, зміну золота та репутації (див. послідовно рис. 2.7 та 28).

```
[System.Serializable]
public class TreatmentResult
{
    public bool DiagnosisCorrect;
    public bool PotionCorrect;
    public bool IsSuccessful;

    public string SelectedDiagnosisName;
```

Рисунок 2.7 – Структура TreatmentResult (1 частина)

```
    public string ActualDiseaseName;
    public string SelectedPotionName;
    public string RequiredPotionName;

    public BrewQuality BrewQuality;

    public int GoldChange;
    public int ReputationChange;

    public string OutcomeText;
}
```

Рисунок 2.8 – Структура TreatmentResult (2 частина)

2.5 Реалізація інвентарю та внутрішньоігрової економіки

Для зберігання і керування ресурсами використовується InventoryManager, який відповідає за поточний стан інвентарю (див. додаток Г). У межах цієї реалізації інвентар містить набір ігрових предметів, переважно алхімічних інгредієнтів, а також обмеження на максимальну кількість доступних слотів. Кожен слот відповідає одній одиниці конкретного ресурсу, що спрощує облік предметів і водночас дозволяє реалізувати механіку обмеженої місткості майстерні.

Відображення вмісту інвентарю реалізовано через InventoryPanelController, який відповідає за показ поточних ресурсів у користувацькому інтерфейсі. Панель оновлюється після змін у стані інвентарю, зокрема після придбання інгредієнтів, використання їх у рецептах або скидання стану під час нової гри.

Для реалізації ринку використовується окрема підсистема, представлена сервісом MarketService і контролером інтерфейсу MarketPanelController (див. додаток Д). MarketService відповідає за формування списку поточних пропозицій, тобто інгредієнтів, які доступні для купівлі, а також їхньої вартості. MarketPanelController відображає ці пропозиції в інтерфейсі, дозволяє обрати конкретний товар і виконати покупку. Після придбання

інгредієнта система перевіряє, чи достатньо у гравця золота, чи є вільне місце в інвентарі, після чого оновлює стан інвентарю, економіки та відповідних UI-панелей.

2.6 Реалізація системи покращень майстерні

Керування системою покращень реалізовано через WorkshopUpgradeService, який відповідає за зберігання інформації про вже придбані покращення, перевірку умов придбання та застосування їхнього ефекту (див. рис. 2.9).

```
public bool PurchaseUpgrade(UpgradeDefinition upgrade)
{
    if (upgrade == null)
    {
        Debug.LogError("Upgrade is null");
        return false;
    }

    if (GameSession.Instance == null || GameSession.Instance.PlayerStats == null)
    {
        Debug.LogError("GameSession or PlayerStats is missing");
        return false;
    }

    if (IsUpgradePurchased(upgrade))
    {
        Debug.Log("Upgrade already purchased: " + upgrade.DisplayName);
        return false;
    }

    if (GameSession.Instance.PlayerStats.Gold < upgrade.Cost)
    {
        Debug.Log("Not enough gold for upgrade: " + upgrade.DisplayName);
        return false;
    }

    GameSession.Instance.AddGold(-upgrade.Cost);
    purchasedUpgradeIds.Add(upgrade.Id);

    Debug.Log("Upgrade purchased: " + upgrade.DisplayName);
    return true;
}
```

Рисунок 2.9 – Структура покупки покращення

Відображення доступних покращень і взаємодія з гравцем реалізовані через UpgradesPanelController. Цей контролер формує список доступних апгрейдів, показує їх у відповідному вікні інтерфейсу та забезпечує запуск покупки. Після вибору певного покращення система перевіряє, чи достатньо у гравця золота для його придбання, чи не було це покращення активоване раніше, і лише після цього змінює стан підсистеми.

2.7 Реалізація журналу знань

Відображення довідкової інформації реалізовано через JournalPanelController, який керує вікном журналу та його вкладками (див. рис. 2.10 та 2.11).

```
private void SetTabDiseases(bool rememberState)
{
    currentTab = JournalTabType.Diseases;
    FillDiseasesDropdown();

    if (currentDiseases.Count > 0)
    {
        selectedDiseaseIndex = Mathf.Clamp(selectedDiseaseIndex, 0, currentDiseases.Count - 1);
        entryDropdown.value = selectedDiseaseIndex;
        entryDropdown.RefreshShownValue();
    }

    RefreshDetails();

    if (rememberState)
    {
        SaveCurrentDropdownIndex();
    }
     }

      ...

private void SetTabRecipes(bool rememberState)
{
    currentTab = JournalTabType.Recipes;
    FillRecipesDropdown();

    if (currentRecipes.Count > 0)
    {
        selectedRecipeIndex = Mathf.Clamp(selectedRecipeIndex, 0, currentRecipes.Count - 1);
```

Рисунок 2.10 – Структура відображення інформації в журналі (частина 1)

```
        entryDropdown.value = selectedRecipeIndex;
        entryDropdown.RefreshShownValue();
    }

    RefreshDetails();

    if (rememberState)
    {
        SaveCurrentDropdownIndex();
    }
        }
```

Рисунок 2.11 – Структура відображення інформації в журналі (частина 2)

У межах поточної реалізації журнал містить щонайменше дві основні категорії: хвороби та рецепти. Для вибору конкретного запису використовується випадаючий список, а детальна інформація відображається в текстовій області.

2.8 Реалізація механізму збереження та завантаження гри

Основу цієї підсистеми становить SaveService, який відповідає за формування структури даних збереження, запис цієї структури у файл та її подальше відновлення (див. рис. 2.12).

У межах гри зберігаються не лише загальні показники, такі як золото, репутація або номер дня, а й стан інших важливих підсистем. Зокрема, механізм збереження охоплює поточного пацієнта, доступні ресурси інвентарю, придбані покращення, відкриті елементи журналу знань, а також поточні ринкові пропозиції.

Окрему роль у реалізації відіграє інтеграція механізму збереження з інтерфейсом користувача. Для цього використовується SaveLoadUIController, який надає гравцеві кнопки для збереження, завантаження та початку нової гри (див. рис. 2.13).

Через цей контролер користувач взаємодіє з SaveService, не працюючи безпосередньо з внутрішньою логікою файлового обміну.

```
public void SaveGame()
{
    SaveData data = BuildSaveData();

    string json = JsonUtility.ToJson(data, true);
    File.WriteAllText(SavePath, json);

    Debug.Log("Game saved to: " + SavePath);

}

public bool LoadGame()
{
    if (!File.Exists(SavePath))
    {
        Debug.LogWarning("Save file does not exist: " + SavePath);
        return false;
    }

    string json = File.ReadAllText(SavePath);
    SaveData data = JsonUtility.FromJson<SaveData>(json);

    if (data == null)
    {
        Debug.LogError("Failed to parse save file");
        return false;
    }

    ApplySaveData(data);

    Debug.Log("Game loaded from: " + SavePath);
    return true;
}
```

Рисунок 2.12 – Структура збереження та завантаження гри

```
public void SaveGame()
{
    if (SaveService.Instance != null)
    {
        SaveService.Instance.SaveGame();
        if (ToastMessageController.Instance != null)
        {
            ToastMessageController.Instance.ShowMessage("Game Saved");
        }
    }
}

public void LoadGame()
{
    if (SaveService.Instance != null)
    {
        SaveService.Instance.LoadGame();
        if (ToastMessageController.Instance != null)
        {
            ToastMessageController.Instance.ShowMessage("Game Loaded");
        }
    }
}
```

Рисунок 2.13 – Структура SaveLoadUIController

3 ТЕСТУВАННЯ ТА ОЦІНКА РЕЗУЛЬТАТІВ РОЗРОБКИ

Проведення тестування реалізації поставленої задачі, підбиття підсумків розробки та аналіз можливостей покращення проєкту.

3.1 Організація процесу тестування

На завершальному етапі розробки інді-гри «Flamel» було проведено тестування реалізованих підсистем з метою перевірки відповідності програмного продукту поставленим завданням. Основна увага приділялася перевірці працездатності ключових ігрових механік, коректності взаємодії між окремими модулями та стабільності виконання основних сценаріїв гри.

3.2 Аналіз основного інтерфейсу гри

На головній ігровій сцені гри «Flamel» реалізовано основний робочий простір гравця (рис. 3.1), у межах якого відбувається взаємодія з пацієнтом, постановка діагнозу, лікування, керування інвентарем та доступ до допоміжних підсистем. Інтерфейс побудований таким чином, щоб гравець одночасно бачив поточний стан гри, інформацію про пацієнта, результати своїх дій та швидкий доступ до додаткових вікон.

У лівій частині інтерфейсу розміщена панель пацієнта та діагностики. Вона містить основну інформацію про поточного персонажа: його тип, перелік видимих симптомів, елемент вибору діагнозу та поле для відображення результату. Саме ця панель реалізує діагностичну частину гри. Гравець спочатку аналізує симптоми, після чого у випадаючому списку обирає одне з можливих захворювань і підтверджує свій вибір.



Рисунок 3.1 – Основний інтерфейс гри

У центральній частині сцени знаходиться модель поточного пацієнта, яка виконує роль візуального представлення ігрового випадку.

Праворуч від центральної частини розміщена панель лікування. Вона показує, яке зілля наразі підготовлено, а також результат лікування після його застосування. У цій панелі гравець виконує фінальну дію поточного циклу – натискає кнопку лікування пацієнта. Саме тут відбувається завершальна перевірка всіх попередніх дій: чи правильно було поставлено діагноз, чи підходить обране зілля до захворювання та якою є якість його приготування. Після цього система формує підсумковий результат, що впливає на золото, репутацію та подальший перебіг гри. Також у цій панелі розміщено кнопку завершення справи, яка дозволяє перейти до наступного пацієнта.

У правій частині екрана знаходиться панель інвентарю, яка показує вміст сховища та кількість доступних слотів. Тут відображаються наявні інгредієнти, що використовуються в алхімічній системі.

3.3 Аналіз допоміжних вікон та модулів

У верхній частині екрана розміщено інформаційну панель, яка відображає основні параметри ігрової сесії: кількість золота, рівень репутації,

номер поточного дня та фазу гри. Це потрібно для того, щоб гравець у будь-який момент міг бачити свій прогрес і розуміти, на якому етапі ігрового циклу він перебуває. Наприклад, фаза Diagnosis означає, що поточний етап пов’язаний із діагностикою захворювання, а отже саме зараз гравець має проаналізувати симптоми та вибрати правильний діагноз.

Також на панелі присутні кнопки завершення дня та кнопка для відкриття меню (рис. 3.2), де можна зберегти або завантажити прогрес.



Рисунок 3.2 – Вкладка меню

У нижній правій частині екрана розташовані кнопки доступу до допоміжних підсистем: алхімії, ринку, журналу знань і покращень. Це окремі модулі гри, які відкриваються за потреби та доповнюють основний сценарій. Кнопка Alchemy відкриває алхімічне вікно (рис. 3.3), де гравець може обрати рецепт і виконати процес приготування зілля.

Кнопка Market відкриває вікно ринку, у якому можна купувати відсутні інгредієнти (рис. 3.4).

Кнопка Journal забезпечує доступ до журналу знань про симптоми, хвороби та рецепти (рис. 3.5).



Рисунок 3.3 – Вкладка алхімії

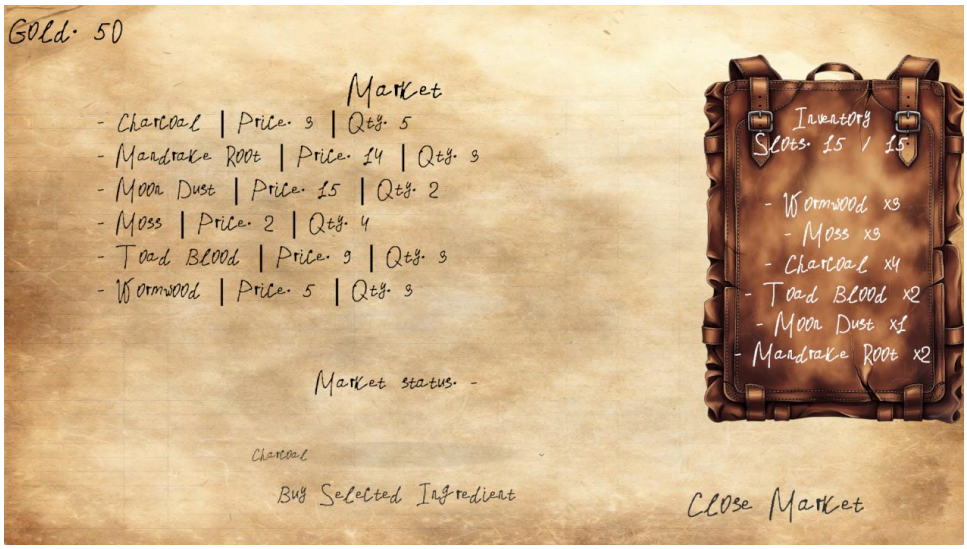


Рисунок 3.4 – Вкладка ринку

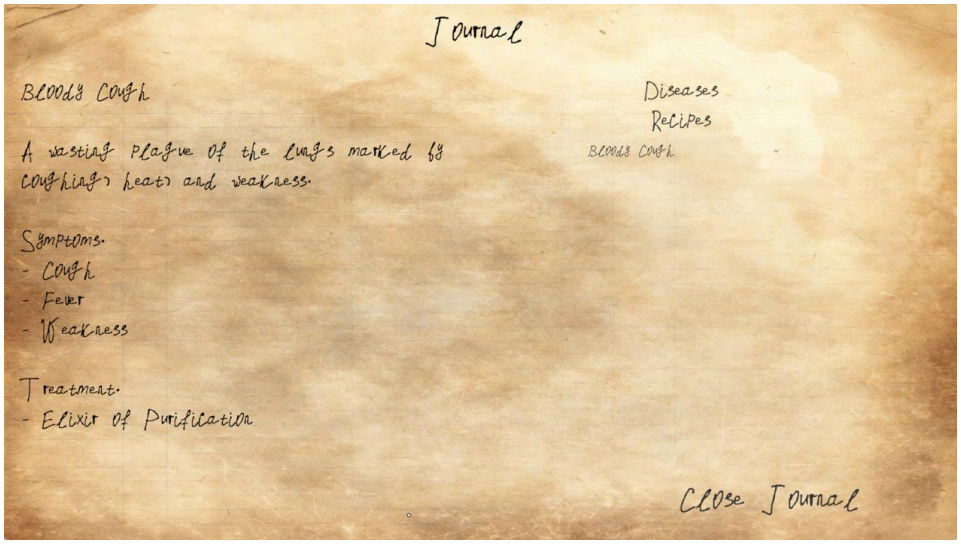


Рисунок 3.5 – Вкладка журналу

Кнопка Upgrades призначена для взаємодії з системою покращень майстерні, яка підтримує довгостроковий розвиток гри (рис. 3.6).

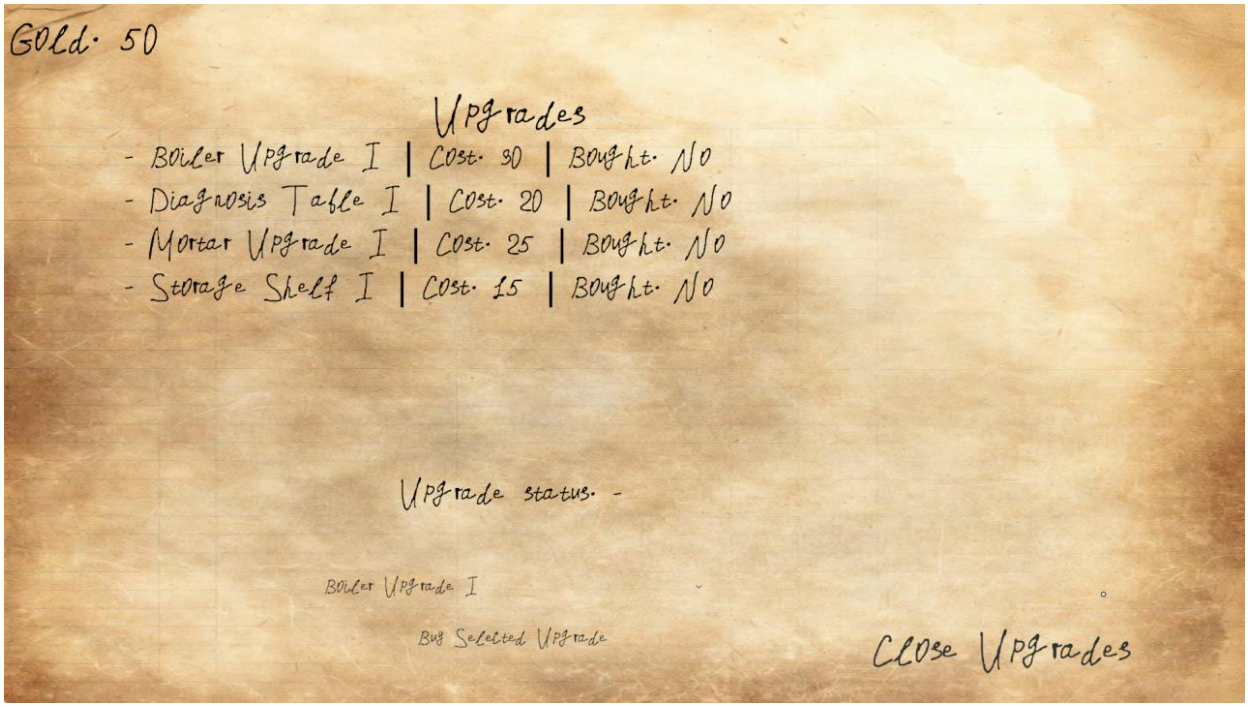


Рисунок 3.6 – Вкладка покращень

У сукупності описані допоміжні вікна та елементи інтерфейсу формують єдину взаємопов’язану систему керування, яка повністю покриває всі функціональні потреби користувача під час ігрової сесії.

3.4 Результати тестування основних підсистем

У процесі тестування було перевірено роботу системи генерації пацієнтів, механіки діагностики, алхімічної підсистеми, лікування, інвентарю, ринку, журналу знань, системи покращень майстерні, а також механізму збереження та завантаження гри.

Окремо було проаналізовано працездатність користувацького інтерфейсу, правильність оновлення даних на панелях і коректність переходів між фазами ігрового процесу.

Перевірка проводилася шляхом послідовного проходження основних сценаріїв гри, а також моделювання типових і помилкових дій користувача.

3.5 Аналіз можливостей подальшого покращення проєкту

Перспективами подальшого розвитку гри є розширення набору пацієнтів, симптомів, хвороб, інгредієнтів і рецептів, ускладнення ігрової логіки, додавання нових типів покращень, покращення візуального оформлення, анімацій та звукового супроводу, а також подальша оптимізація внутрішньої структури програмного коду.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено функціональний прототип інді-гри «Flamel» із використанням ігрового рушія Unity та мови програмування C#. У межах проєкту реалізовано основні механіки, що формують цілісний ігровий цикл.

Практичні результати реалізації підтверджують, що гра коректно відтворює запроєктований сценарій взаємодії. Гравець може послідовно пройти всі ключові етапи.

Розроблений прототип має потенціал для подальшого розвитку. Насамперед можливе розширення контентної складової гри: додавання нових хвороб, симптомів, типів пацієнтів, інгредієнтів, рецептів і зіль. Це дозволить збільшити варіативність ігрових ситуацій та підвищити інтерес до повторного проходження. Перспективним також є поглиблення системи прогресії, зокрема додавання нових типів покращень майстерні, складнішої економіки, нових умов відкриття контенту та системи репутаційного розвитку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ethics and game design: teaching values through play / ed. by K. Schrier, D. Gibson. Hershey PA : Information Science Reference, 2010. 367 p.

2. Fullerton T. Game design workshop: a playcentric approach to creating innovative games : 2nd ed. Amsterdam : Elsevier Morgan Kaufmann, 2008. 470 p.

3. McShaffry M., Graham D. Game coding complete : 4th ed. Boston, MA : Cengage Learning PTR, 2013. 911 p.

4. Geig M. Unity game development in 24 hours, sams teach yourself. Sams Publishing, 2021. 464 p.

5. Nystrom R. Game programming patterns. Apress, 2011. 300 p.

6. Schell J. Art of game design: a book of lenses. Taylor & Francis Group, 2008.

ДОДАТОК А

Логіка запуску та виконання алхімічної сесії

```
public bool StartBrewing(RecipeDefinition recipe)
{
    if (CurrentSession != null && CurrentSession.IsActive)
    {
        Debug.LogWarning("A brewing session is already active");
        return false;
    }

    if (recipe == null)
    {
        Debug.LogError("Recipe is null");
        return false;
    }

    if (InventoryManager.Instance == null)
    {
        Debug.LogError("InventoryManager is missing");
        return false;
    }

    if (!InventoryManager.Instance.HasIngredientsForRecipe(recipe))
    {
        Debug.LogWarning("Not enough ingredients for recipe: " + recipe.DisplayName);
        return false;
    }

    bool consumed = InventoryManager.Instance.ConsumeIngredientsForRecipe(recipe);
    if (!consumed)
    {
        Debug.LogWarning("Failed to consume ingredients for recipe: " + recipe.DisplayName);
        return false;
    }

    CurrentSession = new BrewingSession
    {
        Recipe = recipe,
        CurrentStepIndex = 0,
        IsActive = true,
        IsCompleted = false,
        IsFailed = false,
        ProducedPotion = null,
        Quality = BrewQuality.None,
        MistakeCount = 0
    };
```

```
    LastBrewingResult = null;

    Debug.Log("Brewing started: " + recipe.DisplayName);
    return true;
     }

     ...

public BrewingStepResult PerformAction(AlchemyActionType actionType)
{
    BrewingStepResult stepResult = new BrewingStepResult();

    if (CurrentSession == null || !CurrentSession.IsActive)
    {
        stepResult.Success = false;
        stepResult.Message = "No active brewing session.";
        return stepResult;
    }

    if (CurrentSession.Recipe == null)
    {
        stepResult.Success = false;
        stepResult.Message = "Recipe is missing.";
        FailCurrentSession();
        return stepResult;
    }

    if (CurrentSession.CurrentStepIndex >= CurrentSession.Recipe.Steps.Count)
    {
        stepResult.Success = false;
        stepResult.Message = "All recipe steps are already completed.";
        return stepResult;
    }

    RecipeStepData expectedStep = CurrentSession.Recipe.Steps[CurrentSession.CurrentStepIndex];
    CurrentSession.PerformedActions.Add(actionType);

    bool correctAction = expectedStep.ActionType == actionType;

    if (correctAction)
    {
        stepResult.Success = true;
        stepResult.WasMistake = false;
        stepResult.CompletedStepIndex = CurrentSession.CurrentStepIndex;
        stepResult.Message = "Correct step " + (CurrentSession.CurrentStepIndex + 1) + ": " + actionType;
    }
    else
    {
        CurrentSession.MistakeCount++;

        stepResult.Success = true;
        stepResult.WasMistake = true;
```

```
            stepResult.CompletedStepIndex = CurrentSession.CurrentStepIndex;
            stepResult.Message =
                "Wrong action on step " + (CurrentSession.CurrentStepIndex + 1) +
                ". Expected: " + expectedStep.ActionType +
                ", but got: " + actionType;
        }

        CurrentSession.CurrentStepIndex++;

        if (CurrentSession.CurrentStepIndex >= CurrentSession.Recipe.Steps.Count)
        {
            CompleteCurrentSession();
        }

        return stepResult;
            }

         ...

    public void StartBrew()
    {
        if (brewingController == null)
        {
            Debug.LogError("BrewingController is missing");
            return;
        }

        if (availableRecipes.Count == 0)
        {
            brewingStatusText.text = "Brewing status: no recipes available.";
            return;
        }

        int selectedIndex = recipeDropdown.value;
        if (selectedIndex < 0 || selectedIndex >= availableRecipes.Count)
        {
            brewingStatusText.text = "Brewing status: invalid recipe selection.";
            return;
        }

        RecipeDefinition selectedRecipe = availableRecipes[selectedIndex];

        bool success = brewingController.StartBrewing(selectedRecipe);



        if (!success)
        {
            brewingStatusText.text = "Brewing status: failed to start brewing. Check ingredients.";
            if (AudioManager.Instance != null)
            {
                AudioManager.Instance.PlayError();
```

```
        }
    }
    else
    {
        brewingStatusText.text = "Brewing status: started brewing " + selectedRecipe.DisplayName;
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayBrewStart();
        }
    }

    if (inventoryPanelController != null)
    {
        inventoryPanelController.Refresh();
    }

    RefreshSessionUI();
}
```

ДОДАТОК Б

Структура роботи процесу лікування

```
public void TreatPatient()
{
    if (currentPatient == null)
    {
        Debug.LogError("No current patient assigned");
        return;
    }

    if (dayFlowController != null && dayFlowController.CurrentPhase != DayPhase.Treatment)
    {
        treatmentResultText.text = "Treatment result: treatment is not allowed in the current phase.";
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayError();
        }
        return;
    }

    if (currentPatient.IsTreated)
    {
        treatmentResultText.text = "Treatment result: this patient has already been treated.";
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayError();
        }
        return;
    }

    if (diagnosisPanelController == null)
    {
        treatmentResultText.text = "Treatment result: diagnosis panel is missing.";
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayError();
        }
        return;
    }

    if (alchemyPanelController == null)
    {
        treatmentResultText.text = "Treatment result: alchemy panel is missing.";
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayError();
        }
```

```
            return;
        }

        DiseaseDefinition selectedDiagnosis = diagnosisPanelController.GetSelectedDisease();
        if (selectedDiagnosis == null)
        {
            treatmentResultText.text = "Treatment result: choose a diagnosis first.";
            if (AudioManager.Instance != null)
            {
                AudioManager.Instance.PlayError();
            }
            return;
        }

        PotionDefinition brewedPotion = alchemyPanelController.GetLastProducedPotion();
        if (brewedPotion == null)
        {
            treatmentResultText.text = "Treatment result: no prepared potion. Brew a potion first.";
            if (AudioManager.Instance != null)
{
    AudioManager.Instance.PlayError();
}
            return;
        }

        BrewQuality brewQuality = alchemyPanelController.GetLastBrewQuality();

        TreatmentResult result = treatmentResolver.Resolve(
            currentPatient,
            selectedDiagnosis,
            brewedPotion,
            brewQuality
        );

        currentPatient.IsTreated = true;

        if (GameSession.Instance != null)
        {
            GameSession.Instance.AddGold(result.GoldChange);
            GameSession.Instance.AddReputation(result.ReputationChange);
        }

        ShowResult(result);

        if (result.IsSuccessful)
        {
            if (AudioManager.Instance != null)
            {
                AudioManager.Instance.PlayTreatmentSuccess();
            }
        }
        else
```

```
    {
        if (AudioManager.Instance != null)
        {
            AudioManager.Instance.PlayTreatmentFail();
        }
    }

    alchemyPanelController.ClearProducedPotion();
    RefreshPreparedPotionUI();

    if (patientEncounterController != null)
    {
        patientEncounterController.NotifyTreatmentCompleted();
    }

    if (hudController != null)
    {
        hudController.Refresh();
    }
}
```

ДОДАТОК В

Структура TreatmentResolver

```
public TreatmentResult Resolve(
    PatientRuntimeData patient,
    DiseaseDefinition selectedDiagnosis,
    PotionDefinition selectedPotion,
    BrewQuality brewQuality)
{
    TreatmentResult result = new TreatmentResult();

    if (patient == null)
    {
        Debug.LogError("Patient is null");
        result.IsSuccessful = false;
        result.OutcomeText = "Treatment failed: patient is missing.";
        return result;
    }

    if (patient.Disease == null)
    {
        Debug.LogError("Patient disease is null");
        result.IsSuccessful = false;
        result.OutcomeText = "Treatment failed: disease data is missing.";
        return result;
    }

    if (selectedDiagnosis == null)
    {
        Debug.LogError("Selected diagnosis is null");
        result.IsSuccessful = false;
        result.OutcomeText = "Treatment failed: no diagnosis selected.";
        return result;
    }

    if (selectedPotion == null)
    {
        Debug.LogError("Selected potion is null");
        result.IsSuccessful = false;
        result.OutcomeText = "Treatment failed: no potion selected.";
        return result;
    }

    result.SelectedDiagnosisName = selectedDiagnosis.DisplayName;
    result.ActualDiseaseName = patient.Disease.DisplayName;
    result.SelectedPotionName = selectedPotion.DisplayName;
    result.RequiredPotionName = patient.Disease.RequiredPotion != null
        ? patient.Disease.RequiredPotion.DisplayName
```

```
            : "Unknown";

        result.BrewQuality = brewQuality;

        result.DiagnosisCorrect = patient.Disease.Id == selectedDiagnosis.Id;
        result.PotionCorrect = patient.Disease.RequiredPotion != null &&
                               patient.Disease.RequiredPotion.Id == selectedPotion.Id;

        bool validPotion = result.PotionCorrect && brewQuality != BrewQuality.Corrupted;
        result.IsSuccessful = result.DiagnosisCorrect && validPotion;

        if (result.IsSuccessful)
        {
            if (brewQuality == BrewQuality.Perfect)
            {
                result.GoldChange = patient.Disease.BaseRewardGold + 5;
                result.ReputationChange = patient.Disease.ReputationReward + 2;
                result.OutcomeText = "Perfect treatment! The patient recovered completely.";
            }
            else if (brewQuality == BrewQuality.Standard)
            {
                result.GoldChange = patient.Disease.BaseRewardGold;
                result.ReputationChange = patient.Disease.ReputationReward;
                result.OutcomeText = "Treatment successful! The patient has been cured.";
            }
            else
            {
                result.GoldChange = patient.Disease.BaseRewardGold / 2;
                result.ReputationChange = patient.Disease.ReputationReward / 2;
                result.OutcomeText = "The patient survived, but the potion was weak.";
            }
        }
        else
        {
            result.GoldChange = 0;
            result.ReputationChange = patient.Disease.ReputationPenalty;

            if (brewQuality == BrewQuality.Corrupted)
            {
                result.OutcomeText = "Treatment failed. The corrupted potion harmed the patient.";
            }
            else
            {
                result.OutcomeText = "Treatment failed. The patient's condition worsened.";
            }
        }

        return result;
    }
```

ДОДАТОК Г

Структура зміни ресурсів в інвентарі

```
public bool AddIngredient(IngredientDefinition ingredient, int amount = 1)
{
   if (ingredient == null)
   {
      Debug.LogError("Ingredient is null");
      return false;
   }

   if (amount <= 0)
   {
      Debug.LogWarning("Add amount must be greater than zero");
      return false;
   }

   if (!HasFreeSlots(amount))
   {
      Debug.LogWarning("Not enough free inventory slots");
      return false;
   }

   for (int i = 0; i < amount; i++)
   {
      ingredientEntries.Add(new InventoryEntry(ingredient));
   }

   Debug.Log("Added ingredient: " + ingredient.DisplayName + " x" + amount);
   return true;
    }

     ...

public bool RemoveIngredient(IngredientDefinition ingredient, int amount = 1)
{
   if (ingredient == null)
   {
      Debug.LogError("Ingredient is null");
      return false;
   }

   if (amount <= 0)
   {
      Debug.LogWarning("Remove amount must be greater than zero");
      return false;
   }
```

```
    if (!HasIngredient(ingredient, amount))
    {
        Debug.LogWarning("Not enough ingredient in inventory: " + ingredient.DisplayName);
        return false;
    }

    int removed = 0;

    for (int i = ingredientEntries.Count - 1; i >= 0; i--)
    {
        if (ingredientEntries[i] != null &&
            ingredientEntries[i].Ingredient != null &&
            ingredientEntries[i].Ingredient.Id == ingredient.Id)
        {
            ingredientEntries.RemoveAt(i);
            removed++;

            if (removed >= amount)
            {
                break;
            }
        }
    }

    Debug.Log("Removed ingredient: " + ingredient.DisplayName + " x" + amount);
    return true;
}
```

ДОДАТОК Д

Формування пропозицій та купівлі інгредієнтів

```
public void GenerateDailyOffers()
{
    currentOffers.Clear();

    if (DatabaseProvider.Instance == null || DatabaseProvider.Instance.GameDatabase == null)
    {
        Debug.LogError("GameDatabase is missing");
        return;
    }

    List<IngredientDefinition> ingredients = DatabaseProvider.Instance.GameDatabase.Ingredients;

    for (int i = 0; i < ingredients.Count; i++)
    {
        IngredientDefinition ingredient = ingredients[i];

        if (ingredient == null)
        {
            continue;
        }

        MarketOffer offer = new MarketOffer
        {
            Ingredient = ingredient,
            Price = CalculatePrice(ingredient),
            Quantity = CalculateQuantity(ingredient)
        };

        currentOffers.Add(offer);
    }

    Debug.Log("Daily market offers generated");
}

...

 public bool BuyIngredient(string ingredientId)
 {
     if (GameSession.Instance == null || GameSession.Instance.PlayerStats == null)
     {
         Debug.LogError("GameSession or PlayerStats is missing");
         return false;
     }

     if (InventoryManager.Instance == null)
```

```
    {
        Debug.LogError("InventoryManager is missing");
        return false;
    }

    MarketOffer offer = currentOffers.Find(x => x.Ingredient != null && x.Ingredient.Id == ingredientId);

    if (offer == null)
    {
        Debug.LogWarning("Offer not found for ingredient id: " + ingredientId);
        return false;
    }

    if (offer.Quantity <= 0)
    {
        Debug.LogWarning("Offer is sold out: " + offer.Ingredient.DisplayName);
        return false;
    }

    if (GameSession.Instance.PlayerStats.Gold < offer.Price)
    {
        Debug.LogWarning("Not enough gold");
        return false;
    }

    bool added = InventoryManager.Instance.AddIngredient(offer.Ingredient, 1);

    if (!added)
    {
        Debug.LogWarning("Could not buy ingredient because inventory is full");
        return false;
    }

    GameSession.Instance.AddGold(-offer.Price);
    offer.Quantity -= 1;

    Debug.Log("Bought ingredient: " + offer.Ingredient.DisplayName);
    return true;
}
```