МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

Кафедра інженерії програмного забезпечення

КУРСОВА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

з дисципліни: «Моделювання та аналіз програмного забезпечення»

на тему:

Розробка комп'ютерної гри у жанрі метроїдванія «Dark Watcher»

студента 4 курсу групи ІПЗ-19-1
спеціальності 121 «Інженерія програмного забезпечення»

Козакевич Наталії Олександрівни
(прізвище, ім’я та по-батькові)

Керівник: ___зав. каф. КН, к.т.н, доцент
_____Сугоняк І.І_____

Дата захисту: " __ " _____ 20__ р.
Національна шкала _____
Кількість балів: _____
Оцінка: ECTS _____

Члени комісії _____ Сугоняк І.І.
(підпис) (прізвище та ініціали)

_____ _____
(підпис) (прізвище та ініціали)

_____ _____
(підпис) (прізвище та ініціали)

Житомир – 2022

РЕФЕРАТ

Завданням на курсову роботу було дослідження особливостей моделювання та аналізу програмних комплексів з використанням програмних засобів, що підтримують процеси створення і супроводу інформаційної системи, включаючи аналіз і формулювання вимог, проектування прикладного ПЗ (програмного забезпечення) і баз даних, генерацію коду, документування, конфігураційне керування, а також інші процеси.

Пояснювальна записка до курсового проекту (роботи) на тему «Розробка комп'ютерної гри у жанрі метроїдванія "Dark Watcher"» складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків.

Текстова частина викладена на 37 сторінках друкованого тексту.

Пояснювальна записка має 18 сторінок додатків. Список використаних джерел містить 16 найменувать і займає 2 сторінки. В роботі наведено 18 рисунків. Загальний обсяг роботи — 55 сторінок.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

2D – 2-dimentional

ПК – Персональний комп’ютер

ОЗУ – оперативна пам’ять

ОС – операційна система

ПЗ – програмне забезпечення

ЗМІСТ

ВСТУП

У даній курсовій роботі наведено основні нотації уніфікованої мови моделювання, або, скорочено, мови UML, яка призначена для опису, візуалізації і документування об’єкноорієнтованих систем та бізнес-процесів з орієнтацією на їх подальшу реалізацію у вигляді програмного забезпечення.

Метою курсової роботи є дослідження особливостей моделювання та аналізу програмних комплексів за темою курсової роботи «Розробка комп’ютерної гри у жанрі метроїдванія “Dark Watcher”» напрямком з використанням програмних засобів, що підтримують процеси створення і супроводу інформаційної системи, включаючи аналіз і формулювання вимог, проектування прикладного ПЗ (додатків) і баз даних, генерацію коду, документування, конфігураційне керування, а також інші процеси.

Завданням на курсову роботу є:

- аналіз теоретичних засад моделювання програмного забезпечення;
- аналіз та опис вимог користування;
- методи модулювання функцій та поведінки системи;
- проектування об’єктної структури системи;
- фізичне моделювання програмних комплексів;
- кодогенерація із моделей;

Об’єктом дослідження є методи та засоби проектування програмного забезпечення та уніфікація процесу проектування.

Предметом дослідження виступають можливості застосування CASE-засобів з метою автоматизації виконання всіх етапів концептуального, логічного та фізичного проектування архітектури програмних додатків.

1. АНАЛІЗ КОРИСТУВАЧА ТА КОНЦЕПТУАЛЬНЕ МОДЕЛЮВАННЯ

1.1 Технічне завдання на розробку системи

1. Назва системи:

Повне найменування системи: комп’ютерної гра у жанрі метроїдванія “Dark Watcher”

Коротке найменування системи: «Dark Watcher», гра, програмний комплекс, програма, комп’ютерна гра.

2. Призначення системи:

Програмний комплекс, спрямований на інтерактивну взаємодію з користувачем та донесення сюжетної складової через виконання заздалегідь підготовленими завданнями, інтерактивністю з віртуальним середовищем та через анімаційні сюжети. Перегляд та реєстрація нового гравця, а також обробка статистики.

3. Вимоги до системи:

Програмний продукт повинен володіти наступними функціональними характеристиками:

- Доступ до системи за допомогою перевірки ліцензії ігровим ігровим пусковим пристроєм;
- Експорт статистики та прогресу гри на віддалену базу даних Firebase;
- Віддалене оновлення ігрового додатку;
- Можливість роботи з ігровим проектом методами базових інструментів вводу/виводу, а також інструментів виводу для ігрових консолей;
- Основна частина ігрового процесу Dark Watcher зав’язана на дослідженні світу гри, який включає в себе подолання платформ, пошуки секретів і боїв із ворогами які вам зустрінуться.

- Жанри та теги: метроїдванія, однокористувацька гра, дослідження світу, глибокий сюжет, темне фентезі, двовимірний платформер, піксельна графіка.

4. Вимоги до програмного забезпечення:
   a. Вимоги до інтерфейсу:

Програмний комплекс має бути побудований за методологією сепаративного розміщення додатків.

Ігровий лаунчер – має бути побудований за допомогою використання технології .Net або ж node.js та поєднуватись з віддаленою базою даних Firebase методами асинхронних HTTP-запитів, на інтерфейсі має бути можливість перегляду рейтингових таблиць за допомогою окремих вкладених екранів. Купівля ігрового додатку, а також перегляд власної статистики.

Ігровий додаток «Dark watcher» - матиме візуальні підказки, навігацію за допомогою видимих інтерактивних кнопок, інтерфейс який відображає стан гравця, а також стан неігрових персонажів. Панель налаштувань матиме стандартні принципові позиції налаштувань: роздільна здатність екрану та можливість ввімкнення повноекранного режиму, регулювання гучності, зміна мови, зміна рівня графіки, зміна ігрових клавіш.

До інтерфейсів користувача висуваються наступні вимоги:

- мінімально можливий час відгуку системи;
- оптимізація інтерфейсів та процесів програмного комплексу з точки зору максимальної роботи із системою;
- підтвердження виконання критичних дій в системі;

5. Етапи розробки системи:

Етапи розробки системи було проаналізовано та описано у табл.1.1.

Таблиця 1.1 – Етапи розробки системи

Етап	Зміст робіт	Результат
1	Створення UML-моделі системи	Готова повна UML-модель системи та згенеровані частини коду
2	Проектування інтерфейсів та розробка графічного дизайну всіх модулів	Спрайти та Sprite Sheets (таблиці спрайтів)
3	Проектування інтерфейсів та розробка XAML – розмітка інтерфейсу	XAML.cs файли з розміченим програмним інтерфейсом
4	Проектування нереляційної системи збереження даних Firebase	Груповані JSON-дерева у вигляді словникових файлів.
5	Розробка класової структури проекту	Створено та наповнено функціональні компоненти класовим підходом та патернами проектування, C#.

1.2 Обґрунтування вибору засобів моделювання

Необхідно розглянути доступні варіанті поширених UML редакторів, які у них переваги та недоліки, обрати за результатом аналізу найбільш задовільний.

Для початку розберемо що таке UML – це уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування, є мовою широкого профілю, що використовує графічні позначення для створення абстрактної моделі системи, яка називається UML-моделлю.

Мова UML одночасно є простим і потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних і графічних моделей складних систем самого різного цільового призначення. Конструктивне використання мови UML ґрунтується на розумінні загальних принципів моделювання складних систем та особливостей процесу об'єктно-орієнтованого проектування (ООП) зокрема. Вибір засобів для побудови моделей складних систем зумовлює ті завдання, які можуть бути вирішені з використанням даних моделей.

Тому для порівняння було обрано такі засоби для створення UML-моделей:

- StarUML;
- Diagrams.net;
- Dbdiagram.io.

StarUML – це інструмент розробки програмного забезпечення для системного моделювання з використанням уніфікованої мови моделювання, а також мови моделювання систем і класичних нотацій моделювання.

Переваги:

- Кодогенерація
- Відомий інтерфейс
- Шаблони проектування
- Документація забезпечена вказівками
- Швидкий, гнучкий і може бути розширений для розміщення коду у діаграмі

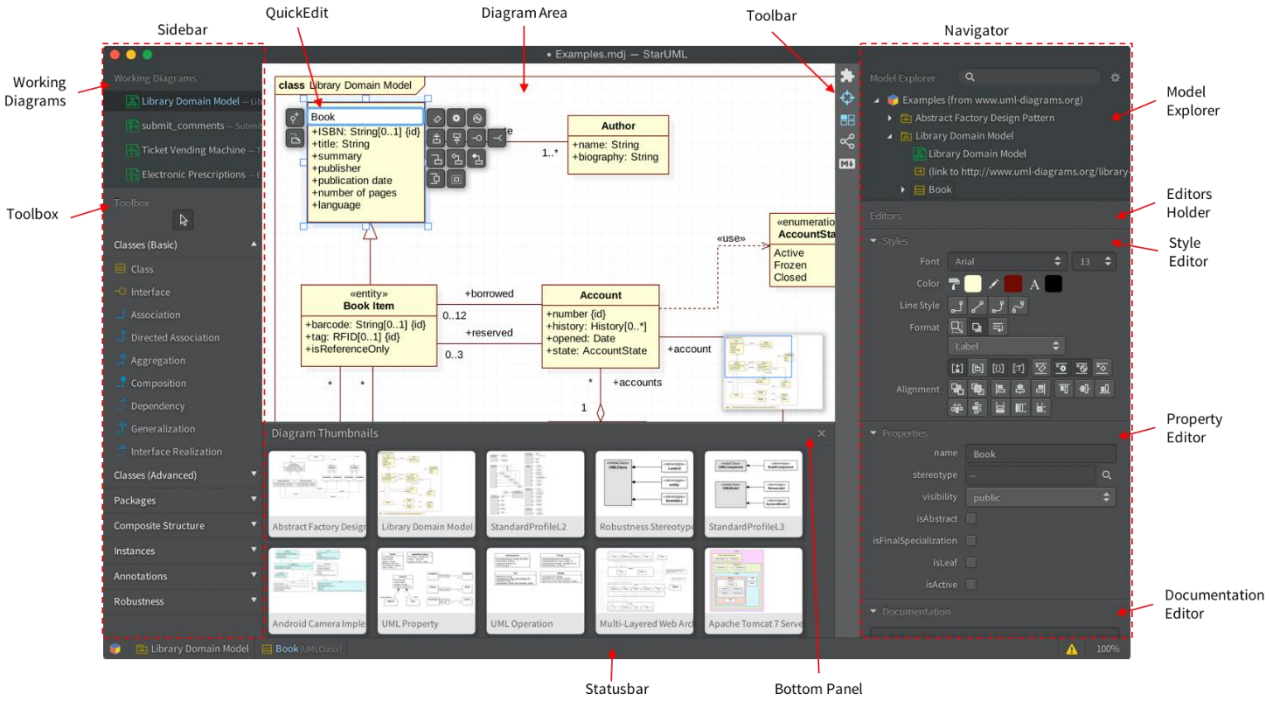

Рисунок 1.1 – Інтерфейс StarUML

Diagrams.net – це безкоштовне міжплатформне програмне забезпечення

для малювання графіків із відкритим кодом, розроблене на HTML5 і JavaScript. Його інтерфейс можна використовувати для створення діаграм, таких як блок-схеми, каркасні схеми, діаграми UML, організаційні діаграми та мережеві діаграми.

Переваги:

- Резервне копіювання та відновлення даних
- Захист даних для ваших файлів
- Інтеграція з багатьма програмами, платформами та інструментами

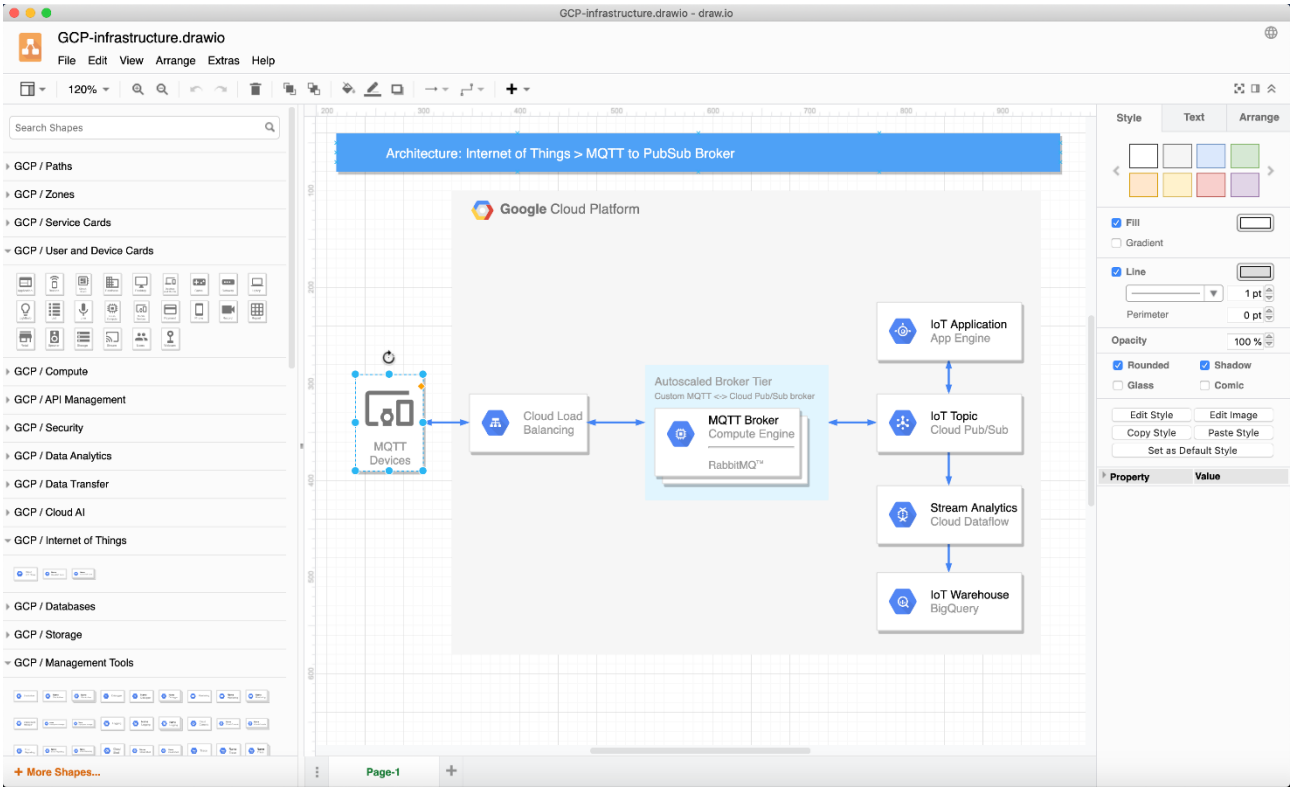

Рисунок 1.2 – Інтерфейс Diagrams.net

Dbdiagram.io – це швидкий та простий безкоштовний інструмент, який допоможе намалювати діаграми відносин з базою даних та працюватиз ними.

Переваги:

- Імпорт і експорт SQL
- Повернення до попередньої версії
- Загальнодоступні та вбудовані діаграми

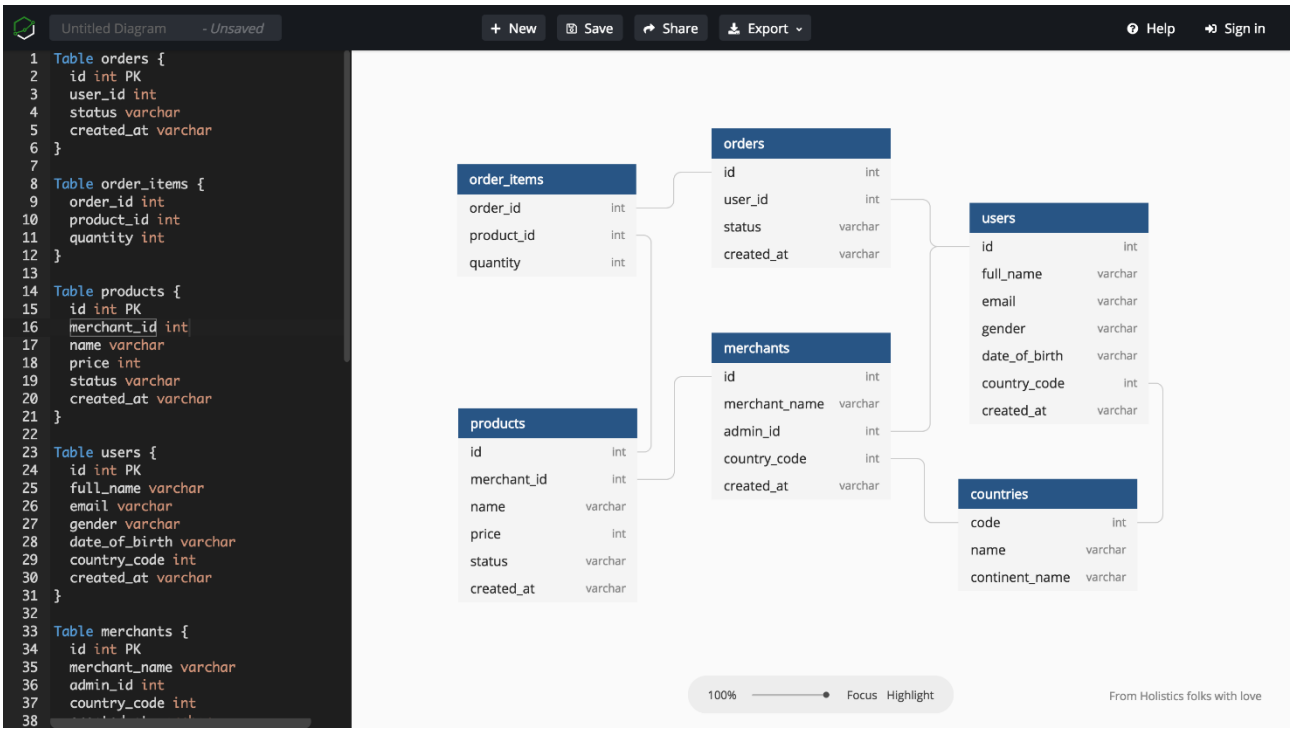

Рисунок 1.3 – Інтерфейс Dbdiagram.io

1.3 Аналіз вимог до програмного продукту

Даний підрозділ (узагальнені і більш детальні ніж у підрозділі 1.1 першого розділу) описує високо рівневі вимоги, якими повинна володіти ігрова система, щоб мати можливість добре виконувати покладені на неї функції, тим самим відповідно задовольнивши всі зазначені вимоги та інші формальні документи. Зазначені умови, яких повинен дотримуватись користувач для роботи з даною ігровим додатком та вказані основні переваги даної програми.

Бізнес-вимоги:

1. Концептуальний опис: Основна частина ігрового процесу Dark Watcher зав’язана на дослідженні світу гри, який включає в себе подолання платформ, пошуки секретів і боїв із ворогами які вам зустрінуться. Гравці досліджують величезний цілісний світ використовуючи особливі поліпшення для пересування і прийоми з бойової системи. У багатьох зонах є перешкоди, які можна подолати лише, якщо гравець отримає перемогу над певними босами і отримавши поліпшення, що випали з них. Таким чином гравцям доводиться відвідувати вже досліджені зони для пошуків секретів або просування по

сюжету. Деякі зони змінюються по ходу сюжету, а тому повторне їх відвідування іноді привносить сюрпризи. Кожна зона з'єднана з іншою зоною декількома шляхами, а тому проходити гру можна по-різному.

2. Система керування: Основна частина ігрового процесу Dark Watcher зав'язана на дослідженні світу гри, який включає в себе подолання платформ, пошуки секретів і боїв із ворогами які вам зустрінуться. Гравці досліджують величезний цілісний світ використовуючи особливі поліпшення для пересування і прийоми з бойової системи. У багатьох зонах є перешкоди, які можна подолати лише перемігши певних босів і отримавши поліпшення, що випали з них. Таким чином гравцям доводиться відвідувати вже досліджені зони для пошуків секретів або просування по сюжету. Деякі зони змінюються по ходу сюжету, а тому повторне їх відвідування іноді привносить сюрпризи. Кожна зона з'єднана з іншою зоною декількома шляхами, а тому проходити гру можна по-різному.

3. Бойова система: Основна частина ігрового процесу Dark Watcher зав'язана на дослідженні світу гри, який включає в себе подолання платформ, пошуки секретів і боїв із ворогами які вам зустрінуться. Гравці досліджують величезний цілісний світ використовуючи особливі поліпшення для пересування і прийоми з бойової системи. У багатьох зонах є перешкоди, які можна подолати лише перемігши певних босів і отримавши поліпшення, що випали з них. Таким чином гравцям доводиться відвідувати вже досліджені зони для пошуків секретів або просування по сюжету. Деякі зони змінюються по ходу сюжету, а тому повторне їх відвідування іноді привносить сюрпризи. Кожна зона з'єднана з іншою зоною декількома шляхами, а тому проходити гру можна по-різному.

4. Вимоги до сюжету: Основна частина ігрового процесу Dark Watcher зав'язана на дослідженні світу гри, який включає в себе подолання платформ, пошуки секретів і боїв із ворогами які вам зустрінуться. Гравці досліджують величезний цілісний світ використовуючи особливі поліпшення для пересування і прийоми з бойової системи. У багатьох зонах є перешкоди, які можна подолати лише перемігши певних босів і отримавши поліпшення, що випали з них. Таким чином гравцям доводиться відвідувати вже досліджені зони для пошуків секретів

або просування по сюжету. Деякі зони змінюються по ходу сюжету, а тому повторне їх відвідування іноді привносить сюрпризи. Кожна зона з'єднана з іншою зоною декількома шляхами, а тому проходити гру можна по-різному.

Вимоги користувачів:

- Користувач, гравець:
  1. Користувач має зареєструватись в ігровому лаунчері
  2. Користувач має змогу завантажити оновлення
  3. Користувач має змогу переглядати статистику гри
  4. Користувач має змогу адаптації та налаштувати під себе зовнішній вигляд персонажу та його предмети в інвентарі
  5. Гравець має змогу переглядати сюжетні елементи
  6. Гравець має змогу вступити в битву з неігровими персонажами
  7. Гравець має змогу обміняти та придбати об'єкти інвентарю

Функціональні вимоги:

1. Відображення ігрового світу: у грі має відображатися ігровий світ, включаючи будь-яких персонажів, об'єкти та елементи середовища, у 2D-перспективі
2. Рух і контроль: гра повинна дозволяти гравцеві контролювати рух і дії свого персонажа за допомогою введення з клавіатури або контролера, окрім моментів анімаційних сюжетів
3. Виявлення зіткнень: гра має точно виявляти та реагувати на зіткнення між персонажами, об'єктами та оточенням
4. Ігрова механіка: гра має реалізовувати механізми та елементи ігрового процесу, які є центральними для дизайну гри, наприклад бої, головоломки, управління ресурсами тощо
5. Аудіо та візуальні ефекти: гра має містити відповідні аудіо та візуальні ефекти для покращення досвіду гравця, такі як музика, звукові ефекти

6. Інтерфейс користувача: гра повинна мати інтерфейс користувача, який дозволяє гравцеві отримувати доступ і взаємодіяти з різними функціями та параметрами, такими як меню, інвентар і налаштування
7. Збереження та завантаження: Гра повинна дозволяти гравцеві зберігати свій прогрес і завантажувати збережені ігри
8. Продуктивність і оптимізація: гра має працювати плавно та ефективно, з мінімальними затримками та іншими проблемами продуктивності

Не функціональні вимоги:

1. Сприйняття:
   - Час, необхідний для навчання звичайних користувачів – 10 хвилин, для адаптації досвідчених користувачів – 3 хвилини
   - Час відгуку для типових задач - не більше 0.01 секунд
   - Час завантаження гри – не більше 20 секунд
   - Час завантаження ігрового екрану – не більше 1 секунди
2. Надійність:
   - Доступність - час, потрібний для обслуговування системи не повинен перевищувати 5 годин на тиждень
   - Середній час безперервної роботи — 5 робочих днів
   - Максимальна норма помилок та дефектів в роботі програми - 1 помилка на 1000 запитів користувача
3. Продуктивність:
   - Швидке отримання результатів при здійсненні функцій програми
   - Система повинна раціонально використовувати ресурси комп’ютера, не перенавантажувати його
4. Можливість експлуатації:
   - Оновлення версій – Оновлення версій повинно здійснюватися в автоматичному режимі через ігровий лаунчер

Системні вимоги:

1. Вимоги до середовища виконання: Система повинна задовільнять вимогам на комп’ютері, що знаходиться в наступній мінімальній комплектації:
    - ОС: Windows 7
    - Процесор: Intel Core 2 Duo E5200
    - Оперативна пам’ять: 4 GB ОП
    - Відеокарта: GeForce 9800GTX+ (1GB)
    - DirectX: версії 10
    - Місце на диску: 2 GB доступного місця
    - Додаткові примітки: 1080p, 16:9 (рекомендовано)

Бізнес правила:

Система повинна бути валідною, та відповідати рейтингам ігрових обмежень. Також система повинна бути user-friendly.

Антивимоги:

1. Не використовує інформацію, заборонену нормами моралі та законодавством
2. У грі має бути чітка політика щодо використання ресурсів та вмісту гри
3. У грі має бути відсутнім будь-який вид дискримінації

Бізнес логіка:

Система має пакетну версію та продається в електронних магазинах відеоігор таких як, steam, epic games store, тощо.

Відповідно до детального аналізу вимог ми можемо побудувати діаграму варіантів використання (Use Case Diagram) комп’ютерної гри у жанрі метроїдванія “Dark Watcher. Але для початку детальніше розглянемо поняття діаграми варіантів використання та її компоненти.

Діаграма варіантів використання - це UML діаграма за допомогою якої в графічному вигляді можна зобразити вимоги до розроблюваної системі, а також відношення між акторами та прецедентами в системі.

Діаграма має такі компоненти:

- Варіант використання (прецедент) – це послідовність дій, які повинні бути виконані системою при взаємодії з відповідним актором
- Актор – це зовнішня до системи сутність, яка взаємодіє з нею для досягнення певних цілей

Відношення:

- Асоціація (association) – вказує конкретну роль, яку грає актор при взаємодії з варіантом використання
- Розширення (extend) – визначає зв'язок екземплярів одного варіантів використання з більш загальним
- Узагальнення (generalization) – вказує що варіант використання А може бути узагальненим до варіанту використання В; А – є спеціалізацією або нащадком, В- пращур
- Включення (include) – вказує що деяка поведінка варіанту використання являється складовою поведінки іншого варіанту використання

Позначення компонентів та відношень діаграми варіантів використання можна побачити на рис.1.4.

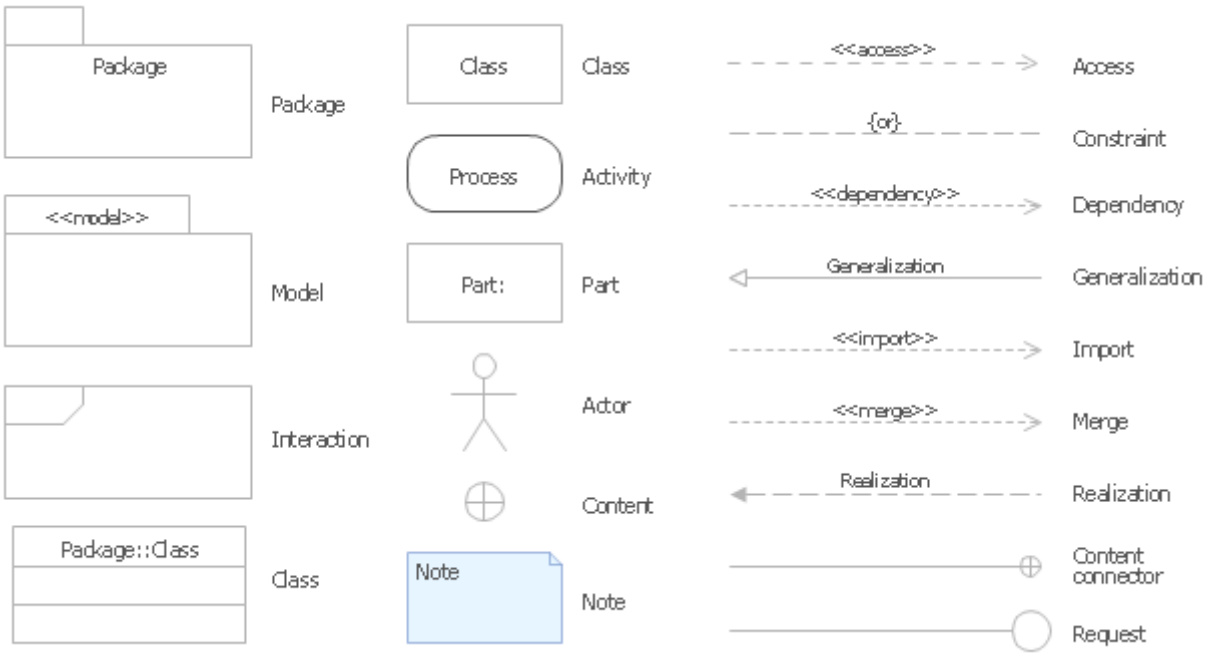

Рисунок 1.4 – Компоненти та відношення діаграми використання

Відповідно до усього вище згаданого була побудована діаграма варіантів використання зображена на рис.1.5.

Детальний опис варіантів використання наведено в додатку А.

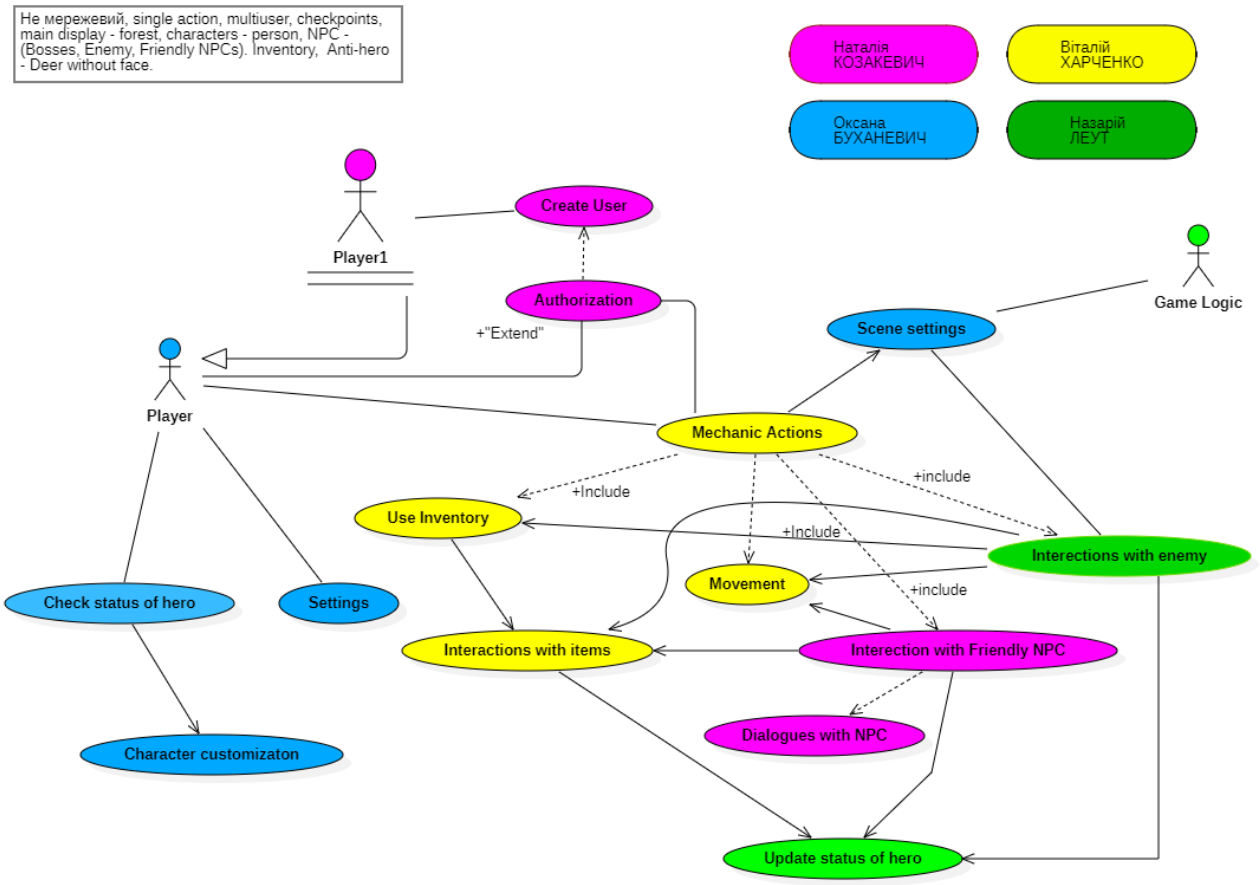

Рисунок 1.5 – Діаграма варіантів використання

Висновки до розділу 1

У цьому розділі було досліджено варіанти UML-редакторів, та обрано кращий з них для подальшої розробки діаграм курсового проектування. Було проаналізовано вимоги до майбутнього продукту, його можливості та побудовано діаграму варіантів використання (Use Case Diagram).

2. РОЗРОБКА МОДЕЛІ ПРОГРАМНОГО КОМПЛЕКСУ НА ЛОГІЧНОМУ РІВНІ

2.1 Алгоритм роботи та стани програмної системи

На даному етапі розробки наведемо загальний алгоритм програми. Задля цього побудуємо діаграму активності (англ. activity diagram), та ознайомимося з її можливостями структуризації системи програми.

Уніфікована мова моделювання включає кілька підмножин діаграм, включаючи діаграми структури, діаграми взаємодії та діаграми поведінки. Діаграми активності разом із діаграмами варіантів використання та кінцевих автоматів вважаються діаграмами поведінки, оскільки вони описують те, що має статися в системі, що моделюється.

Діаграма активності — це UML-діаграма, що зображує потік керування та послідовність дій. Діаграма показує послідовність дій, рішення вузли, цикли та навіть одночасні дії. Діаграми активності широко використовуються в моделюванні робочого процесу – наприклад, для сервіс-орієнтованих програм.

Перш ніж почати складати діаграму діяльності, спочатку потрібно зрозуміти з чого вона складається. Деякі з найпоширеніших компонентів діаграми активності включають такі дії:

- Action: етап діяльності, на якому користувачі або програмне забезпечення виконують певне завдання. У StarUML дії символізуються прямокутниками із заокругленими краями.
- Decision node: умовна гілка в потоці, представлена ромбом. Він включає один вхід і два або більше виходів.
- Control flows: інша назва з’єднувачів, які показують потік між кроками на схемі.
- Start node: символізує початок діяльності. Початковий вузол представлений чорним кругом.
- End node: представляє останній крок у дії. Кінцевий вузол

представлений чорним кругом окресленим колом.

Згідно з діаграмою прецедентів потрібно побудувати діаграми активності «Авторизація», «Взаємодія з неігровими персонажами» та «Діалог з неігровими персонажами»

Рисунок 2.1 – Діаграма активності «Авторизація»

Діаграма «Авторизація» (див. рис. 2.1) відображає взаємодію користивачу з файлами збереження.

Розяснення діаграми «Авторизація»:

- при відкритті меню відбувається відображення доступних файлів збереження, що були створені користувачем
- відбувається вибір необхідного файлу
  - файл з попередньо збереженим ігровим прогресом
  - «пустий» файл, що відповідає за створення нової гри
- запуск обраного файлу з відображенням стартових або збережених данних

Рисунок 2.2 – Діаграма активності «Взаємодія з неігровими персонажами»

На діаграмі «Взаємодія з неігровими персонажами» зображено інтерактивний процес що призводить до отримання певного результату, а саме ворожого чи дружнього.

Розяснення діаграми «Взаємодія з неігровими персонажами»:

- ігровий персонаж входить в радіус взаємодії з неігровими персонажами
    - з’являється можливість співдії
    - вихід з радіусу взаємодії
- після зчитування чи відбулася взаємодія розрізняється якою вона була
    - при дружній взаємодії (розмова, торгівля) відкривається діалогове вікно
    - при ворожій взаємодії (образа, атака) неігровий персонаж переходить у стан ворога

- відповідно до отриманих результатів взаємодії продовжується рух за сценарієм гри

Рисунок 2.3 – Діаграма активності «Діалог з неігровими персонажами»

На рисунку 2.3 зображено різновид дружньої взаємодії з неігровими персонажами, розмову між ними та ігровим персонажем.

Розяснення діаграми «Діалог з неігровими персонажами»:

- після ініціювання взаємодії перед гравцем розгортається діалогове інформаційне вікно
- при діалозі у первний момент розгортається меню вибору відповідей що вплине на подальшу розмову з неігровим персонажем
    - добір відповідей призводить до таких взаємодій як: обмін інформацією, торгівля та прийняття певних завдань
    - при виборі специфічної відповіді діалог завершиться
- взаємодія продовжється до вибору спецефічної відповіді або примусового виходу з діалогу

Розглянемо діаграму класів. Діаграма класів — це тип діаграми в уніфікованій мові моделювання (UML), яка показує структуру системи шляхом зображення класів, атрибутів і зв’язків між об’єктами.

На діаграмі класів класи представлені у вигляді блоків, які містять назву класу, а також його атрибути та методи. Атрибути перераховані всередині поля класу та зазвичай позначені типом даних. Методи також перераховані всередині поля класу та зазвичай позначені типом, які повертають значення.

Діаграми класів зазвичай використовуються в розробці програмного забезпечення для представлення структури системи та проектування нових систем. На діаграмі класів є три основні типи зв’язків, які можна зобразити між класами: асоціація, агрегація та успадкування.

Асоціація: асоціація — це зв’язок між двома класами, де один клас використовує інший. Це представлено на діаграмі класів лінією з відкритою стрілкою, що вказує від класу, який використовує інший клас, до класу, який використовується. Асоціація може бути односторонньою (односпрямованою) або двосторонньою (двонаправленою). В односторонній асоціації тільки один клас використовує інший клас, тоді як у двосторонній асоціації обидва класи використовують один одного.

Агрегація: агрегація – це зв’язок між двома класами, де один клас є частиною іншого. Це представлено на діаграмі класів лінією з ромбовидною формою на кінці, яка вказує від усього класу до частини класу. Агрегація — це спосіб представлення зв’язку між класами, де весь клас має частину класу як атрибут. Клас частини може існувати незалежно від цілого класу і може використовуватися кількома цілими класами.

Успадкування: успадкування – це зв’язок між класами, де один клас (підклас) успадковує характеристики від іншого класу (суперкласу). Це представлено на діаграмі класів лінією із замкнутою стрілкою, що вказує від підкласу до суперкласу. Спадкування — це спосіб створення нового класу, який є модифікованою версією існуючого класу. Підклас успадковує всі атрибути та методи суперкласу, а також може мати власні додаткові атрибути та методи.

Існують також інші типи відносин, які можуть бути зображені на діаграмі класів, наприклад залежність і реалізація. Залежність — це зв’язок між двома класами, де один клас залежить від іншого для своєї реалізації. Реалізація — це зв'язок між класом та інтерфейсом, де клас реалізує інтерфейс.

Для даного програмного продукту було реалізовано різноманітні діаграми класів, нижче наведено приклад двох із них. Кожна з цих діаграм є частиною загальної діаграми комп’ютерної гри.

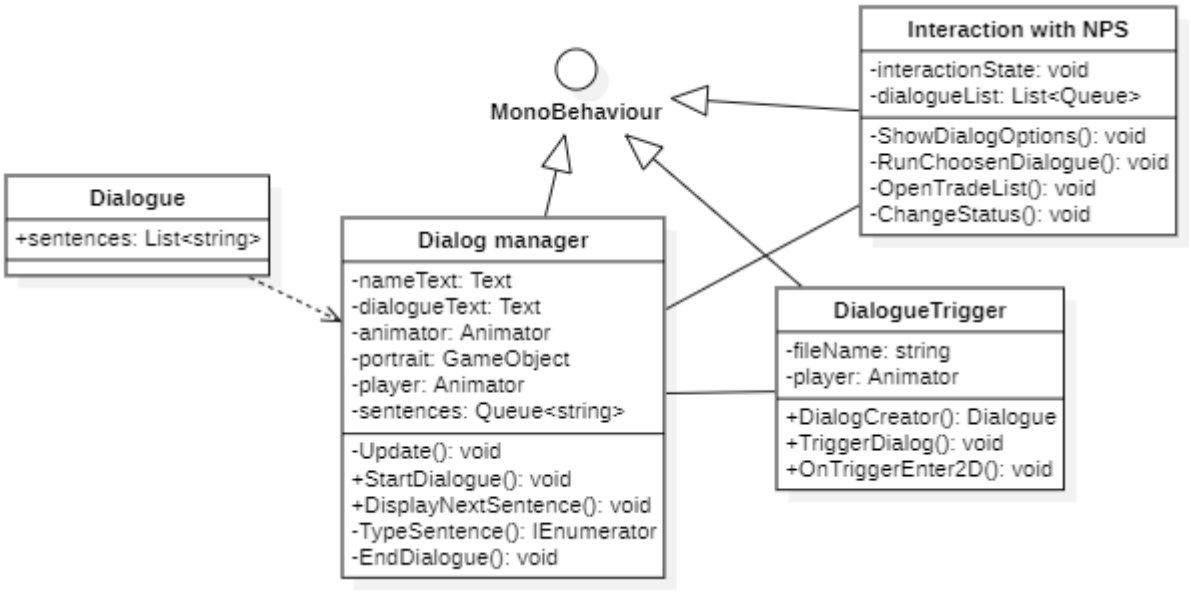

Рисунок 2.4 – Діаграма класів «Dialog»

Розглянемо першу наведену у приклад діаграму класів (див. рис. 2.4), що відповідає за взаємодію гравця з неігровими дружніми персонажами. У назвах класів, полів та методів використано англійську мову та верблюжий регістр (camelCase).

- Клас MonoBehaviour є базовим класом, з якого походить кожен сценарій Unity, є батківським класом для Dialog manager, DialogueTrigger та Interaction with NPS
- Клас Dialog manager містить методи Update(), StartDialogue(), DisplayNextSentence(), TypeSentence() та EndDialogue(), які відповідають за коректне програвання діалогу, його початок, оновлення та завершення.

- Клас Dialogue отримує данні з файлу та передає їх у класс Dialog manager
- Клас DialogueTrigger містить методи DialogCreator(), TriggerDialod() та OnTriggerEnter2D(), які зчитують потрапляння персонажу до області взаємодії та створюють діалог. Також цей клас асоціюється з класом Dialog manager.
- Клас Interaction with NPS містить методи ShowDialogOptions(), RunChosenDialogue(), OpenTradeList() та ChangeStatus(), які відповідають за відображення опцій у діалозі, магазину та передають отриману з обраних опцій інформацію. Цей клас асоціюється з класом Dialog manager.

Наступна діаграма класів, що ми розглянемо (див. рис. 2.5), зображує створення користувачу та збір статистики по уже існуючому.

- Клас IFireBaseClient є батківським класом для HttpJSONAdapter
- Клас HttpJSONAdapter містить методи FetchData(), JSONToDictionary(), SetData(), DictionaryToJSON(), які отримують файли, конвертують JSON файли та записують їх.
- Клас User містить методи UserCreation(), UserAuthorization(), GetConectionToDb(), SaveGameProggres(), SaveGameTime(), SaveLocationProggress(), SaveBossProggress(), SaveDeathStatistic(), SaveHitsStatistic(), що відповідають за створення та авторизацію користувачу, підключення до бази данних та збереження часу прохоження, кулькості отриманих ударів, прогресу локації, гри, смертей та босів
- Клас GameProggress містить методи GetFile() та SetFile(), які відповідають за отримання та відправлення інформації про прогрес гри
- Клас LocationProggress містить методи CalculateProggress() та UpdateDeathStatistic(), що обраховують прогрес дослідження локації та оновлення кількості смертей

- Клас BossProggress містить методи GetHitsCount(), GetBossDeath() та GetTimeInBattle(), які отримують інформацію про кількість отриманих ударів та смертей під час прохождення босу, а також затрачений на нього час
- Клас HitsStatistic містить метод CalculateHits(), що відповідає за підрахування кількості отриманих ударів
- Клас DeathStatistic містить метод CalculateDeathCount(), що відповідає за підрахування кількості програшів

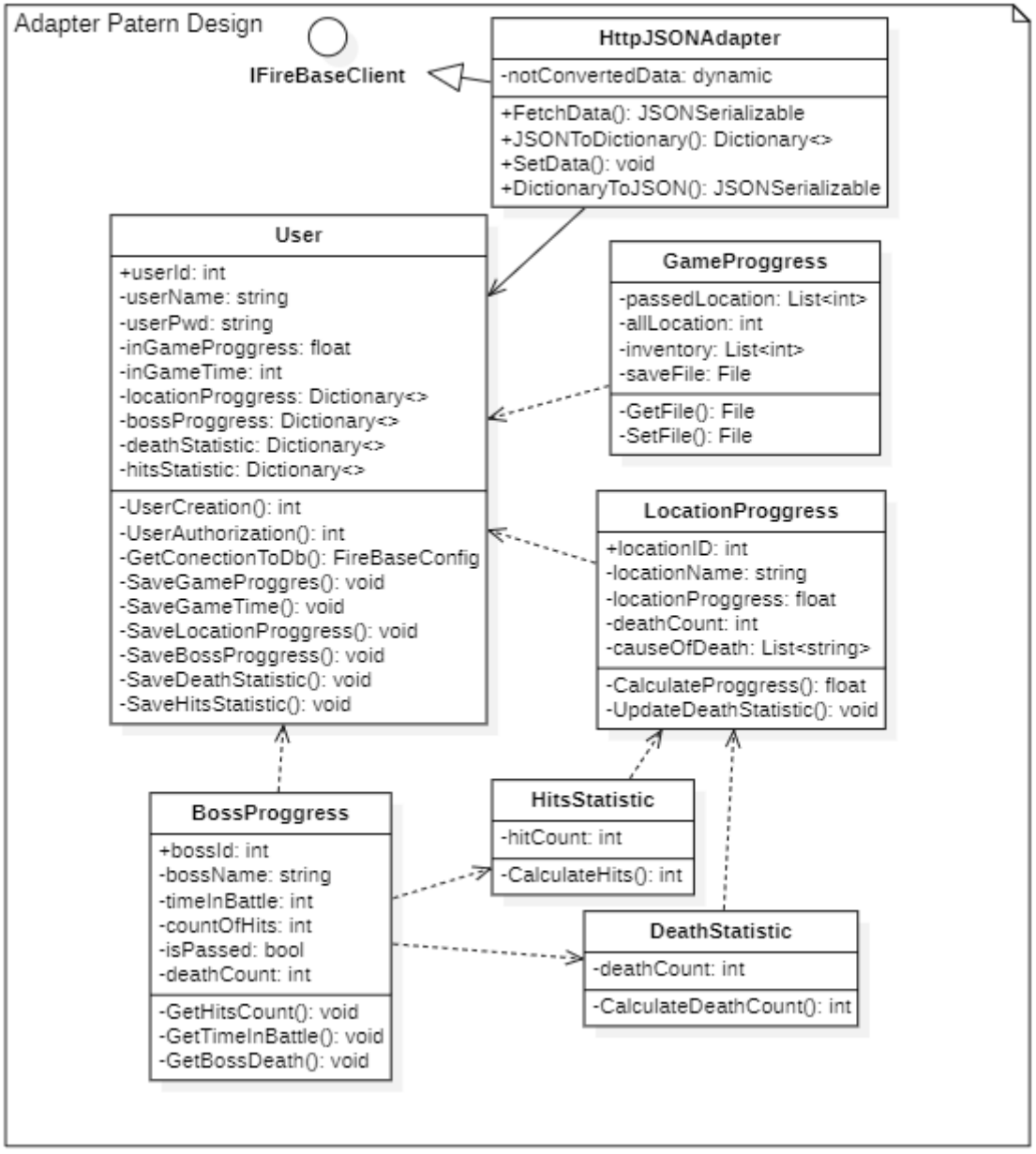

Рисунок 2.5 – Діаграма класів «Користувач»

На рисунку 2.6 зображено загальну командну діаграму класів.

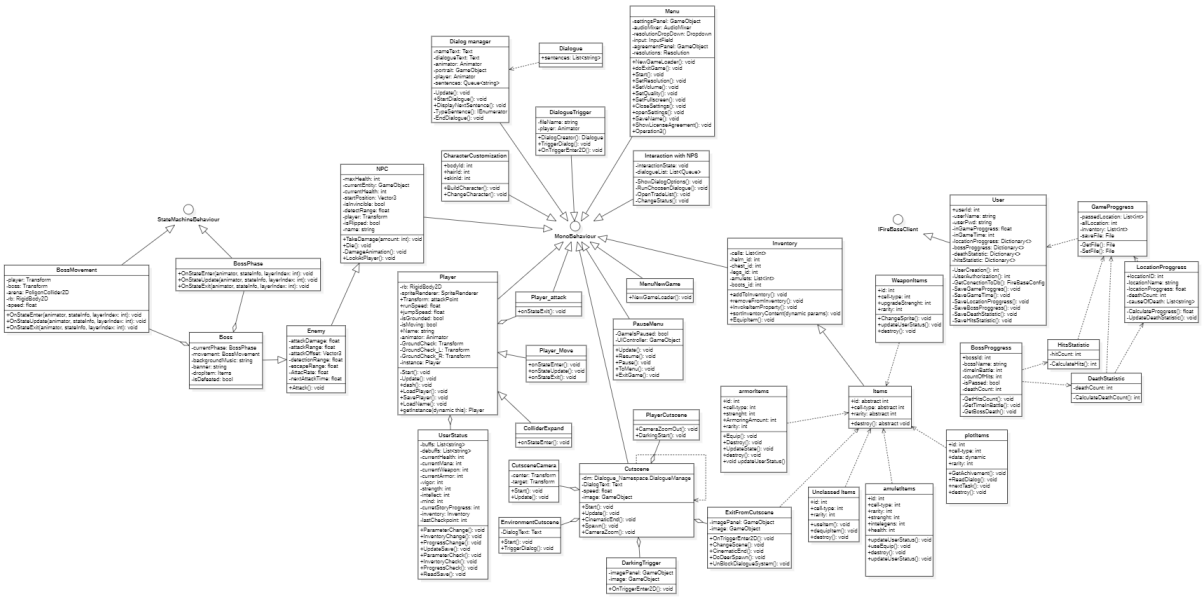

Рисунок 2.6 – Загальна діаграма класів

Далі ми розглянемо один з патернів що використовувався при створенні діаграми класів, але спочатку згадаємо що таке патерн.

Патерн — схема-образ, що діє як посередник уявлення, або чуттєве поняття, завдяки якому в режимі одночасності сприйняття і мислення виявляються закономірності. Патерн дизайну не є готовим проектом, який можна трансформувати безпосередньо в код. Це опис або шаблон того, як вирішити проблему, який можна використовувати в багатьох різних ситуаціях.

При створенні діаграми класів «Користувач» (див. рис. 2.5) було використано патерн проєктування адаптер – це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом.

Схема роботи патерну адаптер:

- Адаптер має інтерфейс, сумісний з одним із об’єктів.
- Тому цей об’єкт може вільно викликати методи адаптера.

- Адаптер отримує ці виклики та перенаправляє їх іншому об’єкту, але вже в тому форматі та послідовності, які є зрозумілими для цього об’єкта.

Згідно цього патерну прослідковується така логіка: при взаємодії з IFireBaseClient клас HttpJSONAdapter конвертує отримуваний фал та передає його у клас User. Аналогічно патерн працює в зворотньому порядку.

2.2 Об’єктно-орієнтована модель системи

Діаграма послідовності — це тип діаграми, який показує взаємодію між об’єктами або компонентами протягом певного періоду часу. Він використовується для моделювання поведінки системи або для опису конкретної взаємодії між об’єктами в системі.

На діаграмі послідовності об’єкти представлені горизонтальними лініями, а взаємодії між об’єктами – вертикальними лініями, які називаються «лініями життя». Взаємодія між об’єктами відображається як повідомлення, які передаються між об’єктами. Повідомлення можуть бути синхронними (це означає, що відправник повідомлення чекає відповіді, перш ніж продовжити), або асинхронними (це означає, що відправник повідомлення не чекає відповіді, перш ніж продовжити).

Діаграми послідовності зазвичай використовуються для моделювання взаємодії між об’єктами в системі в контексті конкретного сценарію або випадку використання. Їх можна використовувати для моделювання взаємодії між об’єктами в системі на різних рівнях абстракції, від оглядів високого рівня до детальних описів низького рівня.

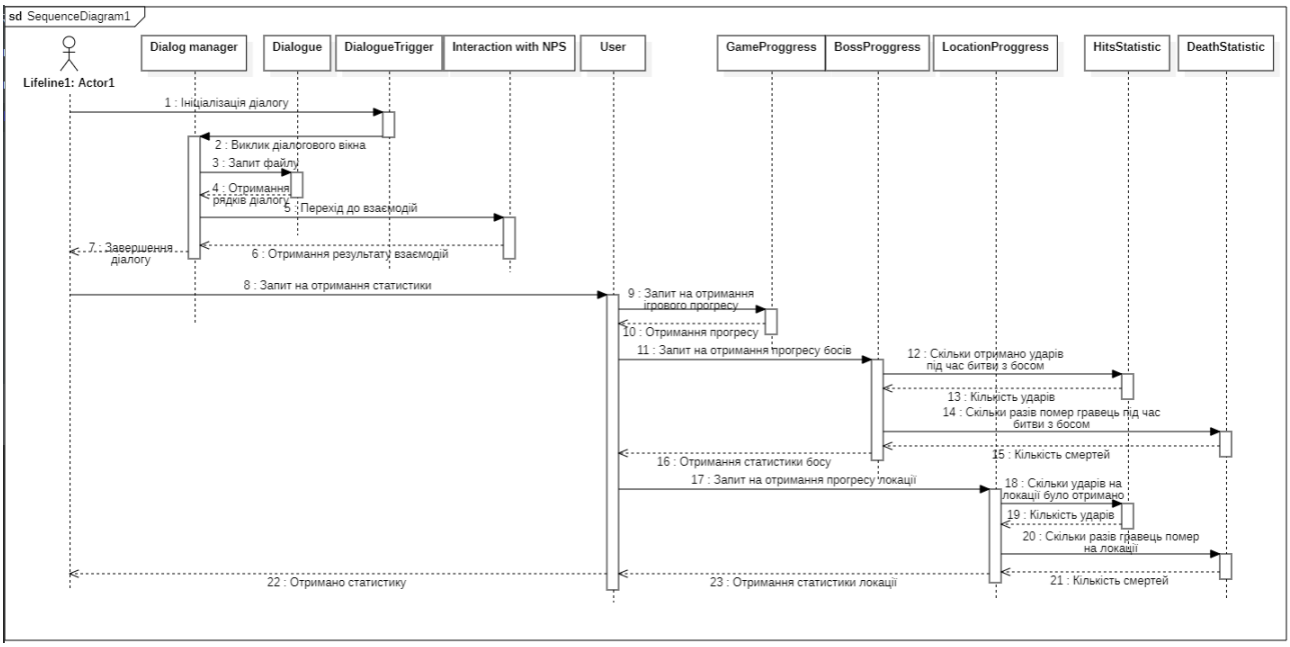

Рисунок 2.7 – Діаграма послідовності

Діаграма комунікації, також відома як діаграма співпраці або діаграма послідовності зв’язку, — це тип діаграми, який показує взаємодію між об’єктами або компонентами в системі. Він використовується для моделювання поведінки системи або для опису конкретної взаємодії між об’єктами в системі.

На діаграмі зв’язку об’єкти представлені прямокутниками, а взаємодія між об’єктами — лініями, що з’єднують коробки. Взаємодія між об’єктами відображається як повідомлення, які передаються між об’єктами. Повідомлення можуть бути синхронними (це означає, що відправник повідомлення чекає відповіді, перш ніж продовжити), або асинхронними (це означає, що відправник повідомлення не чекає відповіді, перш ніж продовжити).

Діаграми зв’язку зазвичай використовуються для моделювання взаємодії між об’єктами в системі в контексті конкретного сценарію або варіанту використання. Їх можна використовувати для моделювання взаємодії між об’єктами в системі на різних рівнях абстракції, від оглядів високого рівня до детальних описів низького рівня.

Комунікаційні діаграми є корисним інструментом для візуалізації та документування поведінки системи, і вони часто використовуються в розробці

програмного забезпечення для розробки та розуміння взаємодії між об’єктами в системі.

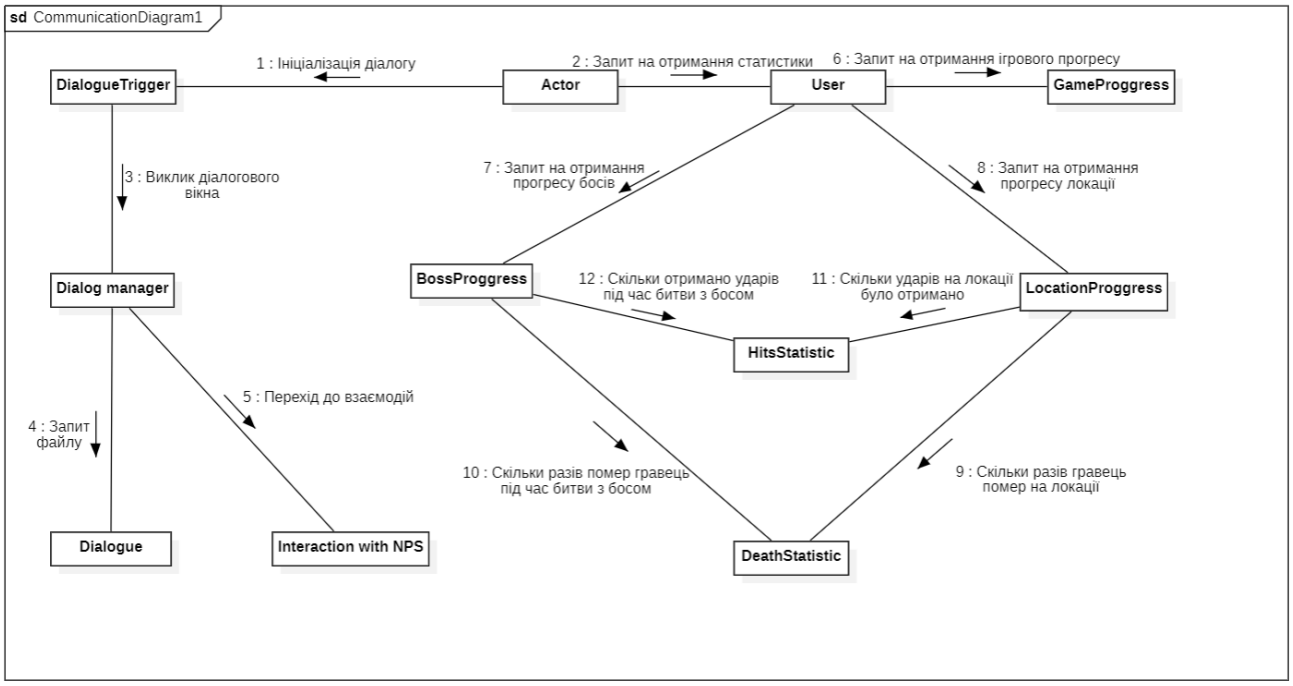

Рисунок 2.8 – Діаграма комунікації

Висновки до розділу 2

У цьому розділі було сформовано алгоритм роботи програми та побудовано діаграми активностей, також було розглянуто діаграми класів та побудовані необхідні для роботи додатку. Було застосовано птерни (design patterns). Сформовано діаграму послідовності та комунікації.

РОЗДІЛ 3. ФІЗИЧНА МОДЕЛЬ ТА ПРОТОТИП ПРОГРАМНОГО КОМПЛЕКСУ

3.1 Взаємодія компонентів системи

Для фізичного представлення моделі системи використовується діаграма реалізації, яка складається з двох діаграм UML: діаграми компонентів і діаграми розгортання.

Діаграми компонентів дозволяють визначити архітектуру системи, що розробляється. Основними графічними елементами діаграми компонентів є компоненти, інтерфейси та залежності між ними.

Компонент:

Компонент (component) – тернім який застосовується у UML для представлення фізичних об’єктів.

Компонент призначений для представлення фізичної організації елементів моделі, з якими він пов’язаний. Також компонент може мати текстовий стереотип і теговане значення, а деякі компоненти - власні графічні зображення.

Компонент може реалізувати певний набір інтерфейсів. Прямокутник і два менші прямокутники зліва використовуються для графічного представлення компонента (див. рис. 3.1). Ім'я компонента та додаткова інформація записуються усередині.

Інтерфейси:

Графічно вони зображуються колами, з'єднаними з компонентами лініями без стрілок (див. рис. 3.1), а коло пишеться назва інтерфейсу, яке рекомендовано писати з великої літери «I».

Крім того, інтерфейс на діаграмі компонентів може бути представлений у вигляді прямокутника класу зі стереотипом <<interface>> та розділом для операцій, що підтримуються (див. рис. 3.1). Як правило, ця нотація використовується для представлення внутрішньої структури інтерфейсів.

Під час розробки програмних систем інтерфейси забезпечують як сумісність між різними версіями, а й можливість вносити істотні зміни до однієї частини програми, залишаючи інші частини без змін. Характер використання інтерфейсу окремими компонентами може бути різним.

Існує два способи підключення інтерфейсів та компонентів. Якщо компонент реалізує інтерфейс, цей компонент надає його послугу іншим компонентам.

Залежності між компонентами:

Залежно від діаграм компонентів представлені пунктирними лініями зі стрілками, що вказують від залежного об'єкта до незалежного об'єкта. (див. рис. 3.1).

Залежно можуть представляти файлові відносини вашої програми під час компіляції. Залежності можуть також вказувати на існування в незалежних компонентах описів класів, що використовуються в залежних компонентах.

На діаграмі компонентів залежності дозволяють зв'язати разом компоненти та інтерфейси, які використовуються конкретними компонентами та компонентами різних типів. У цьому випадку ви малюєте стрілку від компонента клієнта до вашого інтерфейсу. Така стрілка означає, що компонент не реалізує інтерфейс, а використовує під час виконання. При цьому на тій самій діаграмі може існувати інший компонент, що реалізує цей інтерфейс. Відносини реалізації інтерфейсу показані на діаграмах компонентів звичайними лініями без стрілок. (див. рис. 3.1).

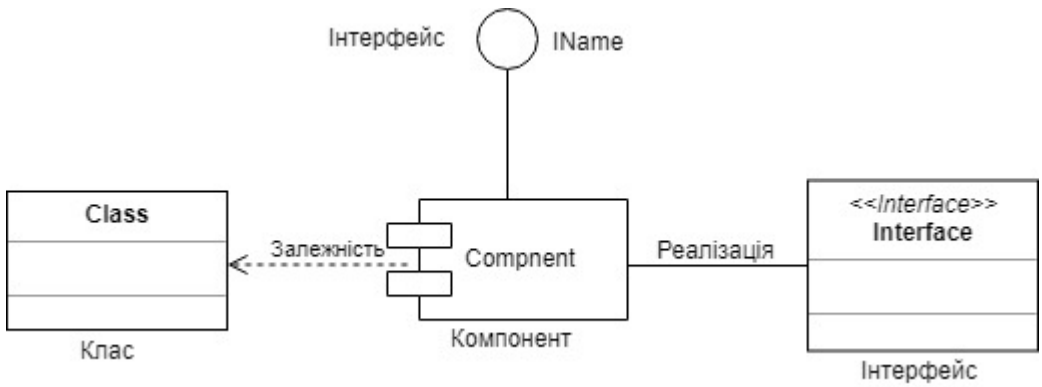

Рисунок 3.1 – Елементи діаграми компонентів

Побудовану діаграму основних компонентів проектованої системи можна побачити на рисунку 3.2.

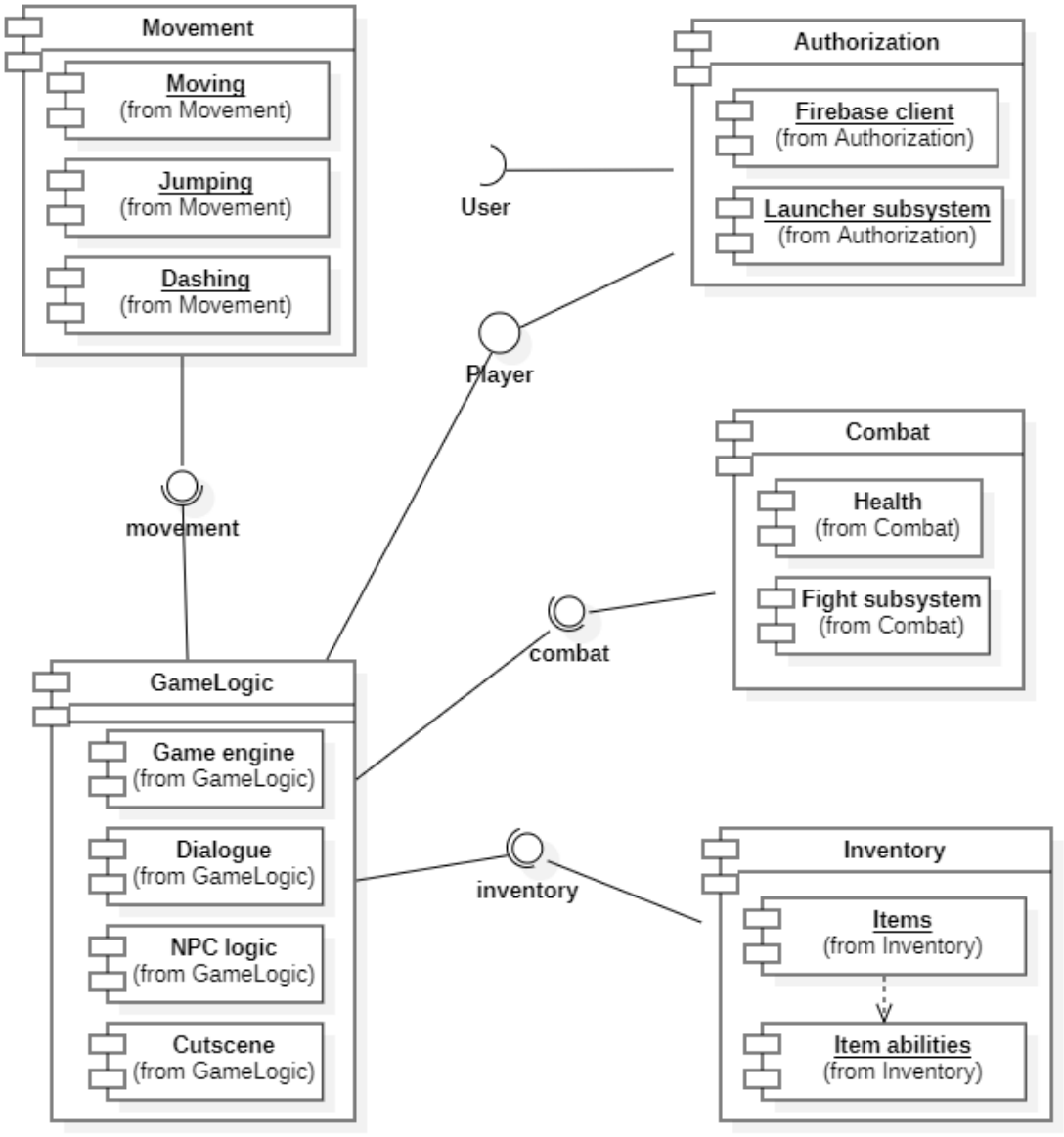

Рисунок 3.2 – Діаграма компонентів

Діаграми розгортання створюються в UML, щоб показати конфігурацію та топологію програмної системи. По суті створення діаграми розгортання — це останній крок у визначенні моделі програмної системи.

Діаграми розгортання використовуються для представлення елементів та компонентів програми, які існують лише на етапі виконання (runtime).

Діаграма розгортання містить графічне уявлення пристроїв, процесів та взаємозв'язків між ними. Створення схеми розгортання супроводжується наступними цілям:

- Розподілити компоненти системи по відповідних фізичних вузлах;
- Зобразити фізичні зв'язки між всіма вузлами системи на етапі виконання;
- Пере-конфігурувати топологію для досягнення вищої продуктивності, якщо це потрібно.

Основним елементом діаграми є вузол. Вузол - це фізично присутній елемент системи з певним обчислювальним ресурсом. Останні версії мови UML розширюють концепцію вузлів, включаючи не тільки обчислювальні пристрої, але й інші механічні або електронні пристрої, такі як принтери, датчики, цифрові камери, модеми, сканери і т.д.

Діаграма розгортання для модельованого програмного комплексу зображено на рисунку 3.3.

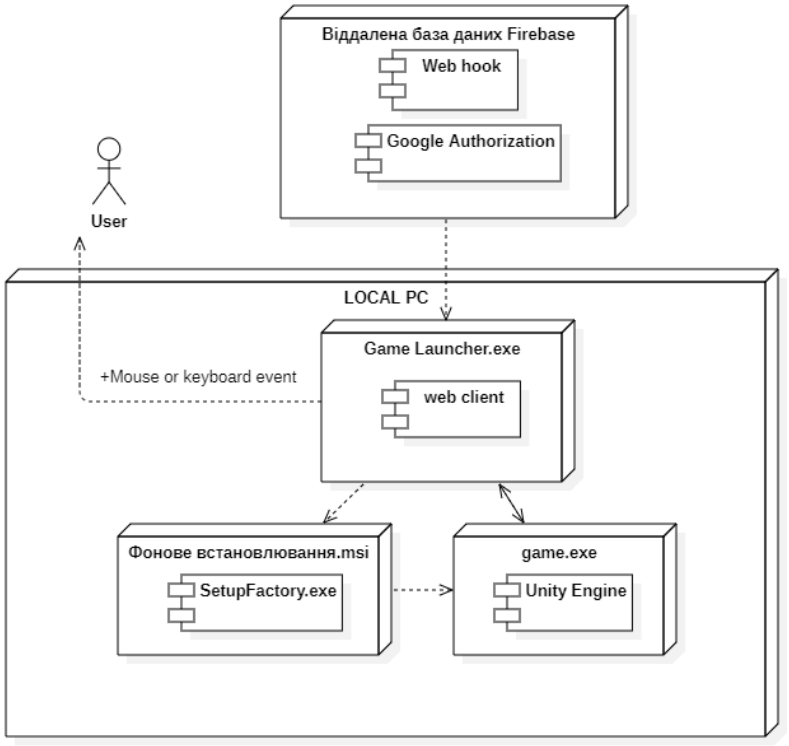

Рисунок 3.3 – Діаграма розгортання

На діаграмі (див. рис. 3.3) можна побачити ігровий вузол, у якому знаходиться компонент «game.exe» (гра), цей компонент позначає саме проектоване програмне забезпечення.

3.2 Генерування програмного коду для прототипу програмного комплексу

На основі діаграми класів, побудованій у другому розділі можна виконати кодогенерацію, яка створить файли класів спроектованої нами системи. Кодогенерація – це процес автоматичного перетворення UML-діаграм у програмний код.

Для генерації програмного коду необхідна діаграма класів, з правильно вказаними зв’язками.

Перед проведенням генерації коду необхідно адаптувати усі елементи під обрану мову програмування: змінити усі використані типи даних на відповідні до обраної мови програмування, назви змінних та методів відповідно до стандартів. Для розробки програми випадку було обрано мову програмування C#.

Для генерації коду на C# у StarUML необхідно виконати наступні кроки:

1. Відкриємо файл зі створеною раніше діаграмою класів;
2. Перейдемо у StarUML – Tools – Extension Manager та встановимо розширення C# (також доступне за посиланням https://github.com/staruml/staruml-csharp);
3. Перезавантажимо додаток StarUML;
4. Перейтдемо у StarUML – Tools – C# – Generate code, оберемо класи код яких потрібно згенерувати та шлях хбереження;
5. Згенерований код з’явиться в обраному каталозі, на цьому генерація коду завершена.

Порядок генерації коду наведено на рис.3.4.

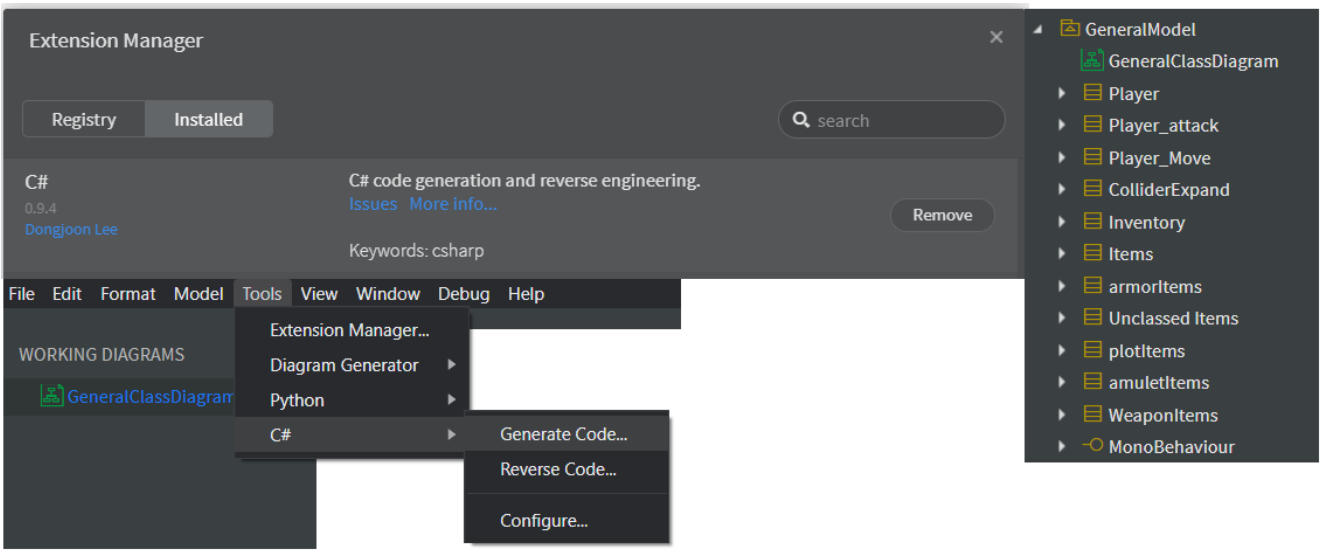

Рисунок 3.4 – Послідовність генерації коду

Згенеровані класи проектованої CRM можна побачити на рис.3.5, а лістинг коду кожного класу наведено в додатку Б.

Ім'я	Дата змінення	Тип	Розмір
Boss.cs	28.12.2022 2:23	C# Source File	1 КБ
Boss1.cs	28.12.2022 2:23	C# Source File	1 КБ
Boss1Movement.cs	28.12.2022 2:23	C# Source File	1 КБ
Boss1Phase.cs	28.12.2022 2:23	C# Source File	1 КБ
Boss2.cs	28.12.2022 2:23	C# Source File	1 КБ
Boss2Movement.cs	28.12.2022 2:23	C# Source File	1 КБ
Boss2Phase.cs	28.12.2022 2:23	C# Source File	1 КБ
BossMovement.cs	28.12.2022 2:23	C# Source File	2 КБ
BossPhase.cs	28.12.2022 2:23	C# Source File	1 КБ
Enemy.cs	28.12.2022 2:23	C# Source File	1 КБ
MonoBehaviour.cs	28.12.2022 2:23	C# Source File	1 КБ
NPC.cs	28.12.2022 2:23	C# Source File	2 КБ
StateMachineBehaviour.cs	28.12.2022 2:23	C# Source File	1 КБ

Рисунок 3.5 – Отримані файли згенерованих класів

На основі згенерованого програмного коду було побудовано прототип ігрового додатку, який містить класи, що базуються на згенерованих і реалізує

частину ігрового процесу. Скриншоти роботи прототипу гри, що відображають логіку роботи, наведено у додатку В.

Висновки до розділу 3

У третьому розділі було проаналізовано взаємодію між компонентами системи та виходячи з цього побудовано діаграми компонетів та розгортання. Також було згенеровано програмний код, що базується на створених раніше діаграмах і реалізовано прототип, який демонструє частину роботи майбутнього ігрового додатку.

ВИСНОВКИ

У першому розділі було досліджено різні варіанти UML-редакторів, та обрано кращий з них для подальшої розробки діаграм курсового проектування.

Було проаналізовано вимоги до майбутнього продукту, його можливості та побудовано діаграму варіантів використання (Use Case Diagram).

У другому розділі розділі було сформовано алгоритм роботи програми та побудовано діаграми активностей, також було розглянуто діаграми класів та побудовані необхідні для роботи додатку. Було застосовано патерни (design patterns). Сформовано діаграму послідовності та комунікації.

У третьому розділі було проаналізовано взаємодію між компонентами системи та виходячи з цього побудовано діаграми компонетів та розгортання. Також було згенеровано програмний код, що базується на створених раніше діаграмах і реалізовано прототип, який демонструє частину роботи майбутнього ігрового додатку.

В результаті виконання курсової роботи було розроблено UML модель системи комп’ютерної гри у жанрі метроїдванія «Dark Watcher», яку буде використано у подільшій розробці повноцінного програмного продукту. В ході її розробки було створено діаграми, що входять до мови моделювання UML. Відповідно до виконаних завдань досліджено особливості моделювання та аналізу програмних комплексів, що є метою курсової роботи.

ЛІТЕРАТУРА

1. UML classes and relationships [Електронний ресурс] – Режим доступу до ресурсу: http://www.cs.utsa.edu/~cs3443/uml/uml.html.
2. CASE-технології та CASE-засоби проектування [Електронний ресурс] – Режим доступу до ресурсу: https://pidru4niki.com/10760623/informatika/case-tehnologiyi_case-zasobi_proektuvannya.
3. Constructing Software Systems on .Net Platforms [Електронний ресурс] – Режим доступу до ресурсу: https://ecs.syr.edu/faculty/fawcett/handouts/webpages/cse681.htm.
4. StarUML documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.staruml.io/.
5. UML Diagrams: History, Types, Characteristics, Versions, Tools [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/uml-diagrams.html.
6. Hassan Gomaa. Software Modeling and Design / George Mason University, Fairfox, Virginia, 2011.
7. Seidl M, Kappel G, Scholz M, Huemer C. UML classroom. Springer International Publishing, 2015.
8. System Modeling and Analysis: a Practical Approach / Gerrit Muller.
9. Software Modeling and Analysis Using Hierarchical Object-Oriented Petri Nets / Jang Eui Hong and Doo-Hwan Bae.
10. Podeswa H. UML FOR THE IT BUSINESS ANALYST / Howard Podeswa. – United States of America: CEBGAGE LEARNING, 2018. – 400 с. – (978-1-59863-868-4).
11. UML 2.0 in Action: A project-based tutorial: A detailed and practical walk-through showing how to apply – MUMBAI: Packt Publishing Ltd., 2005. – (1). – (1-904811-55-8).
12. Refactoring.Guru [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://refactoring.guru/design-patterns/catalog.

13. Static Modeling using the Unified Modeling Language – Филадельфия: Drexel University, 2022. – 67 с. – (1).
14. UML Tutorial. // Tutorialspoint. – 2010. – №12. – С. 504.
15. UML: історія, Специфікація, Бібліографія. УРЛ: http://it.ridne.net/node/265.
16. Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design / Craig Larman.. – 656 с. – (978-0130925695).

ДОДАТКИ

Додаток А

Таблиця А.1 – Специфікація прецеденту Authorization

Ім’я варіанту використання: Authorization		Рівень важливості: високий
Основний актор: Користувач		
Короткий опис		Користувач може відкрити та переглянути список файлів збереження та обрати один з них
Передумови		Реєстрація
Пост-умови		
Відносини		
Асоціація		
Включення		
Розширення		Create user
Узагальнення		
Основний потік		Користувач відкриває меню вибору файлів збереження, після чого обирає один з них. Відбувається завантаження ігрового процессу з останнього збереження даного файлу.
Альтернативні/виключні потоки:		Create user

Таблиця А.2 – Специфікація прецеденту Create user

Ім’я варіанту використання: Create user		Рівень важливості: високий
Основний актор: Користувач		
Короткий опис		Користувач відкриває меню та створює новий файл збереження вказуючи ім’я героя
Передумови		Реєстрація
Пост-умови		
Відносини		

Асоціація	
Включення	Authorization
Розширення	
Узагальнення	
Основний потік	Користувач відкриває меню та створює новий файл збереження вказуючи ім’я героя після чтого відтворюється процес гри з нульовим прогресом її проходження
Альтернативні/виключні потоки:	

Таблиця А.3 – Специфікація прецеденту Interaction with friendly nps

Ім’я варіанту використання: Interaction with friendly nps		Рівень важливості: середній
Основний актор: Користувач		
Короткий опис		Користувач взаємодіє з неігровими персонажами
Передумови		Реєстрація, Авторизація, натискання кнопки
Пост-умови		
Відносини		
Асоціація		Movement, Interaction with items
Включення		
Розширення		Dialogues with nps
Узагальнення		
Основний потік		Користувач взаємодіє з неігровими персонажами через дружні або ворожі взаємодії. Атакує NPC або обирає відповіді у діалоговому вікні
Альтернативні/виключні потоки:		

Таблиця А.4 – Специфікація прецеденту Dialogues with nps

Ім’я варіанту використання: Dialogues with nps	Рівень важливості: середній
Основний актор: Користувач	

Короткий опис	Розмова з неігровими персонажами
Передумови	Реєстрація, Авторизація, натискання кнопки або взаємодія з тригером
Пост-умови	
Відносини	
Асоціація	
Включення	Interaction with friendly nps
Розширення	
Узагальнення	
Основний потік	Користувач натискає клавішу для розмови або стикається з NPC після чого відбувається відображення діалогового вікна. Обираючи різні відповіді отримується різна інформація або відкривається гілка діалогу.
Альтернативні/виключні потоки:	

Додаток Б

Кодогенерація Класу BossProggress:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class BossProggress {

    public BossProggress() {
    }

    public int bossId;

    private string bossName;

    private int timeInBattle;

    private int countOfHits;

    private bool isPassed;

    private int deathCount;

    /// <summary>
    /// @return
    /// </summary>
    private void GetHitsCount() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void GetTimeInBattle() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void GetBossDeath() {
        // TODO implement here
        return null;
    }

}
```

Кодогенерація Класу DeathStatistic:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class DeathStatistic {

    public DeathStatistic() {
    }

    private int deathCount;

    /// <summary>
    /// @return
    /// </summary>
    private int CalculateDeathCount() {
        // TODO implement here
        return 0;
    }

}
```

Кодогенерація Класу Dialog manager:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class DialogManager : MonoBehaviour {

    public DialogManager() {
    }

    private Text nameText;

    private Text dialogueText;

    private Animator animator;

    private GameObject portrait;

    private Animator player;

    private Queue<string> sentences;

    /// <summary>
    /// @return
    /// </summary>
```

```
    private void Update() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public void StartDialogue() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public void DisplayNextSentence() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private IEnumerator TypeSentence() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void EndDialogue() {
        // TODO implement here
        return null;
    }

}
```

Кодогенерація Класу Dialogue:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class Dialogue {

    public Dialogue() {
    }
```

```
    public List<string> sentences;

}
```

Кодогенерація Класу DialogueTrigger:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class DialogueTrigger : MonoBehaviour {

    public DialogueTrigger() {
    }

    private string fileName;

    private Animator player;

    /// <summary>
    /// @return
    /// </summary>
    public Dialogue DialogCreator() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public void TriggerDialog() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public void OnTriggerEnter2D() {
        // TODO implement here
        return null;
    }

}
```

Кодогенерація Класу GameProggress:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
public class GameProggress {

    public GameProggress() {
    }

    private List<int> passedLocation;

    private int allLocation;

    private List<int> inventory;

    private File saveFile;

    /// <summary>
    /// @return
    /// </summary>
    private File GetFile() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private File SetFile() {
        // TODO implement here
        return null;
    }

}
```

Кодогенерація Класу HitsStatistic:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class HitsStatistic {

    public HitsStatistic() {
    }

    private int hitCount;

    /// <summary>
    /// @return
    /// </summary>
    private int CalculateHits() {
        // TODO implement here
```

```
        return 0;
    }

}
```

Кодогенерація Класу HttpJSONAdapter:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class HttpJSONAdapter : IFireBaseClient {

    public HttpJSONAdapter() {
    }

    private dynamic notConvertedData;

    /// <summary>
    /// @return
    /// </summary>
    public JSONSerializable FetchData() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public Dictionary<> JSONToDictionary() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public void SetData() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    public JSONSerializable DictionaryToJSON() {
        // TODO implement here
        return null;
    }
```

```
}
```

Кодогенерація Класу Interaction with NPS:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class InteractionWithNPS : MonoBehaviour {

    public InteractionWithNPS() {
    }

    private void interactionState;

    private List<Queue> dialogueList;

    /// <summary>
    /// @return
    /// </summary>
    private void ShowDialogOptions() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void RunChoosenDialogue() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void OpenTradeList() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void ChangeStatus() {
        // TODO implement here
        return null;
    }

}
```

Кодогенерація Класу LocationProggress:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class LocationProggress {

    public LocationProggress() {
    }

    public int locationID;

    private string locationName;

    private float locationProggress;

    private int deathCount;

    private List<string> causeOfDeath;

    /// <summary>
    /// @return
    /// </summary>
    private float CalculateProggress() {
        // TODO implement here
        return 0.0F;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void UpdateDeathStatistic() {
        // TODO implement here
        return null;
    }

}
```

Кодогенерація Класу MonoBehaviour:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public interface MonoBehaviour {

}
```

Кодогенерація Класу User:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class User : IFireBaseClient {

    public User() {
    }

    public int userId;

    private string userName;

    private string userPwd;

    private float inGameProggress;

    private int inGameTime;

    private Dictionary<string, dynamic> locationProggress;

    private Dictionary<string, dynamic> bossProggress;

    private Dictionary<string, dynamic> deathStatistic;

    private Dictionary<string, dynamic> hitsStatistic;

    /// <summary>
    /// @return
    /// </summary>
    private int UserCreation() {
        // TODO implement here
        return 0;
    }

    /// <summary>
    /// @return
    /// </summary>
    private int UserAuthorization() {
        // TODO implement here
        return 0;
    }

    /// <summary>
    /// @return
    /// </summary>
    private FireBaseConfig GetConectionToDb() {
```

```
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void SaveGameProggres() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void SaveGameTime() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void SaveLocationProggress() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void SaveBossProggress() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void SaveDeathStatistic() {
        // TODO implement here
        return null;
    }

    /// <summary>
    /// @return
    /// </summary>
    private void SaveHitsStatistic() {
        // TODO implement here
        return null;
```

```
    }

}
```

Рисунок В.1. – Візуалізація ігрового процесу

Рисунок В.2. – Дії переміщення та взаємодії з ворогами

Рисунок В.3. – Вигляд діалогу та взаємодії

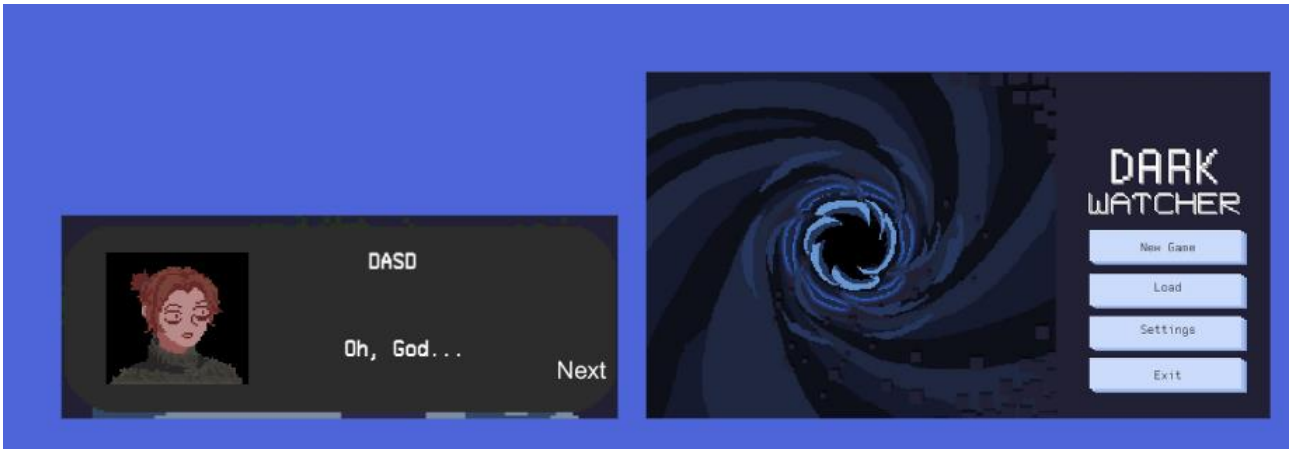

Рисунок В.4. – Головне меню та стартовий діалог