МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

___________ Олексій Горський
"____"__________2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
МАГІСТРА

Тема: "Методика проектування сервіс-орієнтованих застосунків, використовуючи ігровий рушій Unity3D"

Виконавець: Бездітний В'ячеслав Миколайович

Керівник: к.т.н., доцент Горський Олексій Миколайович

Консультант: д.т.н., професор Чебанюк Олена Вікторівна

Нормоконтролер: ст. викладач Гололобов Дмитро Олександрович

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп’ютерної та програмної інженерії
Кафедра інженерії програмного забезпечення
Освітній ступінь магістр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Програмне забезпечення систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Олексій Горський
"___" _______ 2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи студента
Бездітного В’ячеслава Миколайовича

1. Тема кваліфікаційної роботи: «Методика проектування сервіс-орієнтованих застосунків, використовуючи ігровий рушій Unity3D», затверджена наказом ректора від 29.09.2023 р. № 1994/ст.

2. Термін виконання проекту: з 02.10.2022 р. по 31.12.2023 р.

3. Вихідні дані до роботи: особливості проектування архітектури за допомогою ігрового рушія Unity3D, патерни проектування застосунків та практики їх використання, мова розробки C#.

4. Зміст пояснювальної записки:
1. Аналіз предметної області створення додатків в Unity3D.
2. Методика проектування сервіс-орієнтованих застосунків з використанням патерну «Service Locator».
3. Реалізація сервіс-орієнтованого застосунку в Unity3D.

5. Перелік обов'язкових слайдів презентації:
1. Актуальність дослідження, мета та завдання роботи.
2. Доменний аналіз предметної області проектування застосунків у Unity3D.
3. Методика проектування сервіс-орієнтованих застосунків з використанням патерну «Service Locator».
4. Реалізація та застосування запропонованої методики.
5. Висновки.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Ознайомлення з постановкою задачі та вивчення літератури. Доменний аналіз предметної області. Написання 1 розділу, представлення керівнику.	02.10.2023- 17.10.2023	
2.	Підготовка та написання 2 розділу. Відсилка керівнику	18.10.2023- 01.11.2023	
3.	Підготовка та написання 3 розділу. Відсилка керівнику	02.11.2023- 25.11.2023	
4.	Загальне редагування та друк пояснювальної записки, графічного матеріалу. Відсилка ПЗ для перевірки на плагіат одним файлом.	26.11.2023- 04.12.2023	
7.	Проходження нормо-контролю, переплітт пояснювальної записки. Отримання відгуку керівника. Підготовка презентації та тексту доповіді.	05.12.2023- 13.12.2022	
8.	Передзахист кваліфікаційної роботи, Отримання рецензії	14.12.2023- 17.12.2023	
10.	Підготовка матеріалів для передачі секретарю ДЕК (ПЗ, ГМ, CD-R з електронними копіями ПЗ, ГМ, презентації, відгук керівника, рецензія, довідка про успішність, 2 папки, 2 конверти).	18.12.2023- 24.12.2023	
11.	Захист дипломної роботи перед ЕК	25.12.2023р.- 31.12.2023р.	

Дата видачі завдання: 02.10.2023 р.

Керівник дипломної роботи: к.т.н. доцент Олексій ГОРСЬКИЙ

Завдання прийняв до виконання: В’ячеслав БЕЗДІТНИЙ

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Методика проектування сервіс-орієнтованих застосунків, використовуючи ігровий рушій Unity3D»: 76 сторінок, 22 рисунки, 15 використаних джерела, 1 додаток.

SERVICE LOCATOR, DEPENDENCY INJECTION, АРХІТЕКТУРА ЗАСТОСУНКІВ В UNITY3D, STATE MASHINE, СТАНИ ЖИТТЄВОГО ЦИКЛУ ІГРОВОГО ЗАСТОСУНКУ.

Об'єктом дослідження є процес проектування сервіс-орієнтованих застосунків за допомогою ігрового рушія Unity3D.

Предметом дослідження є фундаментальні методи та засоби та розробки сервіс-орієнтованих застосунків.

Метою роботи є розробка методики проектування сервіс орієнтованих застосунків, яка дозволить створити гнучку та масштабовану архітектуру застосунку.

Використані методи дослідження:

1. Метод аналізу: виявлення та аналіз існуючих недоліків та проблем, пов'язаних із застосуванням патерну "Service Locator"; ідентифікація ситуацій, де цей підхід може бути неефективним, і аналіз витрат на управління сервісами.
2. Метод порівняння: порівняння використання патерну "Service Locator" з іншими популярними патернами проектування Unity3D; оцінка переваг та недоліків патерну порівняно із стандартними рішеннями або іншими архітектурними підходами.
3. Метод синтезу: узагальнення отриманих даних та знань для створення рекомендацій щодо оптимального використання патерну "Service Locator" у розробці сервіс-орієнтованих застосунків; формулювання принципів і кращих практик для використання патерну.
4. Практичне застосування: створення сервіс-орієнтованого застосунку з патерном "Service Locator", мінімізуючи недоліки цього патерну;

моделювання різних сценаріїв проектування прототипу для оцінки його здатності до розширення та гнучкості.

Практичне значення та рекомендації застосування отриманих результатів:

- результати дослідження можуть бути використані як розробниками-початківцями, так і досвідченими фахівцями з розробки ігор та застосунків за допомогою ігрового рушія Unity3D, які прагнуть оптимізувати свої процеси розробки та підвищити якість своїх продуктів;
- розробка методики, що забезпечує гнучкість та масштабованість архітектури, що може значно підвищити якість кінцевого продукту, та дозволяє легше адаптуватися до змін у вимогах та технологіях, додавати функціонал без переписування значної частини коду;
- застосування оптимізованих підходів і кращих практик може допомогти уникнути помилок у проектуванні та реалізації, які часто виникають при розробці складних систем;
- рекомендації та принципи, розроблені в ході дослідження, можуть бути корисними не тільки для розробників, які використовують Unity3D, але й у ширшому контексті розробки сервіс-орієнтованих застосунків.

Враховуючи швидкий розвиток технологій і зростання популярності сервіс-орієнтованих архітектур, результати такого дослідження можуть мати значний вплив на галузь програмної інженерії та розробки ігор.

ABSTRACT

The explanatory note to on master’s degree graduation work "Approach of service-oriented applications design using Unity3D game engine": 76 pages, 22 figures, 15 references, 1 appendice.

SERVICE LOCATOR, DEPENDENCY INJECTION, APPLICATION ARCHITECTURE IN UNITY3D, STATE MASHINE, LIFE CYCLE STATES OF GAME APPLICATION.

The object of research is the process of designing service-oriented applications using the Unity3D game engine.

The subject of research is the fundamental methods and tools for developing service-oriented applications.

The purpose of the work is to develop a methodology for designing service-oriented applications that will allow creating a flexible and scalable application architecture.

Research methods that were used:

1. 1.Analysis method: identification and analysis of existing shortcomings and problems associated with the use of the "Service Locator" pattern; identification of situations where this approach may be ineffective and analysis of service management costs.
2. Comparison method: comparing the use of the Service Locator pattern with other popular Unity3D design patterns; evaluating the advantages and disadvantages of the pattern compared to standard solutions or other architectural approaches.
3. Synthesis method: generalization of the obtained data and knowledge to create recommendations for the optimal use of the Service Locator pattern in the development of service-oriented applications; formulation of principles and best practices for using the pattern.
4. Practical application: creation of a service-oriented application with the "Service Locator" pattern, minimizing the disadvantages of this pattern;

modeling of various prototype design scenarios to assess its ability to expand and flexibility.

Practical usage and recomendations of the results:

- The results of the study can be used by both novice and experienced developers of games and applications using the Unity3D game engine who seek to optimize their development processes and improve the quality of their products;
- Developing a methodology that ensures the flexibility and scalability of the architecture can significantly improve the quality of the final product. This makes it easier to adapt to changes in requirements and technologies, and add functionality without rewriting a significant portion of the code;
- application of optimized approaches and best practices can help avoid design and implementation errors that often occur when developing complex systems;
- the recommendations and principles developed in the course of the research can be useful not only for developers using Unity3D, but also in the broader context of developing service-oriented applications.

Given the rapid development of technologies and the growing popularity of service-oriented architectures, the results of this research can have a significant impact on the software engineering and game development industry.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 10
ВСТУП 11

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ СТВОРЕННЯ ДОДАТКІВ В UNITY 3D 15
1.1. Огляд ігрового рушія Unity3D 15
1.2. Огляд популярних патернів проектування, переваги та недоліки 18
1.2.1 Патерн «State» 18
1.2.2 Патерн «Event Bus» 20
1.2.3 Патерн «Command» 24
1.2.4 Патерн «Object pool» 26
1.2.5 Патерн «Observer» 28
1.2.6 Патерн «Visitor» 30
1.2.7 Патерн «Strategy» 32
1.2.8 Патерн «Decorator» 34
1.2.9 Патерн «Adapter» 36
1.2.10 Патерн «Facade» 39
1.3 Огляд патерну «Service Locator», сутність сервісів 42
Висновки 45

РОЗДІЛ 2 МЕТОДИКА ПРОЕКТУВАННЯ СЕРВІС-ОРІЄНТОВАНИХ ЗАСТОСУНКІВ ІЗ ВИКОРИСТАННЯМ ПАТЕРНУ «SERVICE LOCATOR» 46
2.1 Теоретичне підґрунтя для розробки методики 46
2.1.1 Теорія категорій 47
2.2 Опис методики 49
2.2.1 Керування станами, в яких знаходиться застосунок 50

2.2.2 Проектування структури сервіс-орієнтованого застосунку........................ 52
2.2.3 Роль і взаємодія сервісів та компонентів.. 55
2.2.4 Розробка класу Service Locator для управління сервісами 56
2.2.5 Впровадження Dependency Injection контейнера для керування залежностями 58
2.2.6 Використання патерну «Factory» для створення об'єктів, доповнюючи патерн «Service Locator» 60
Висновки 62

РОЗДІЛ 3 РЕАЛІЗАЦІЯ СЕРВІС-ОРІЄНТОВАНОГО ЗАСТОСУНКУ В UNITY 3D 63
3.1. Створення ігрового застосунку для тестування запропонованої методики..... 63
3.2. Реалізація класу Service Locator... 69
3.3. Розробка сервісів та їх взаємодія... 70
Висновки 72

ВИСНОВКИ... 73
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 75
ДОДАТОК А Акт впровадження результатів виконання роботи 77

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API – Application Programming Interface;

DI – Dependency Injection;

ECS – Entity Component System;

FPS – Frame Rate;

FSM – Finite-State Machine;

GUI – Graphical User Interface;

HUD – Heads up Display

IDE – Integrated Development Environment;

MMORPG – Massively Multiplayer Online Role-Playing Game;

MVC – Model-View-Controller;

MVVM – Model-View-ViewModel;

NPC – Non-Player Character;

SOA – Service-Oriented Architecture;

SOLID – S: Single Responsibility ;O: Open-Closed Principle; L: Liskov Substitution Principle; I: Interface Segregation Principle; D: Dependency Inversion Principle;

UI – User Interface;

UML – Unified Modeling Language;

ЦПУ – центральний процесор.

ВСТУП

Актуальність теми. Сучасна ігрова індустрія постійно розвивається, пропонуючи все більш складні та інноваційні рішення у створенні ігрових додатків. Одним із ключових аспектів ефективного розроблення ігор є вибір архітектури, яка забезпечує гнучкість, масштабованість та легкість у підтримці коду. У цьому контексті з'являється необхідність дослідження та аналізу патернів проектування, які можуть бути застосовані у середовищі Unity3D – одному з найпопулярніших ігрових рушіїв сучасності [1].

Проектування архітектури та використання патернів у Unity може становити певні виклики та проблеми, зокрема:

1. Вибір відповідного архітектурного патерну: Unity не накладає жорстких обмежень на архітектуру, що може призвести до труднощів у виборі оптимального патерну. Розповсюджені патерни, наприклад, MVC (Model-View-Controller), MVVM (Model-View-ViewModel) або ECS (Entity Component System), мають свої переваги та недоліки залежно від проекту [2].

2. Початкові архітектурні рішення можуть ускладнити масштабування проекту в майбутньому. Неправильний вибір може призвести до необхідності переписування коду при додаванні нових функцій.

3. У Unity легко створити сильно зв'язані системи, що ускладнює тестування та рефакторинг. Використання принципів SOLID та патернів проектування, таких як Dependency Injection, Service Locator може допомогти керувати цими залежностями.

4. Архітектура, яка ускладнює тестування, може значно збільшити час розробки. Автоматизоване тестування, таке як модульні тести, може бути важким для реалізації у Unity через залежності від ігрового рушія [3].

5. Деякі архітектурні патерни можуть негативно вплинути на продуктивність, особливо в проектах з великою кількістю об'єктів і компонентів. Оптимізація і правильний вибір патернів є ключовими для підтримки високої продуктивності.

6. Інтеграція з зовнішніми системами, такими як бази даних або веб-сервіси, може бути складною залежно від обраної архітектури. Важливо планувати ці аспекти на ранніх етапах розробки.

7. Різні розробники можуть мати різні підходи до архітектури, що може ускладнити співпрацю в команді. Важливо встановити чіткі стандарти і засади архітектури на початку проекту.

Для вирішення цих проблем важливо витрачати час на планування архітектури, регулярно переглядати і оновлювати її відповідно до змін у проекті, а також враховувати досвід та найкращі практики інших розробників.

На сьогоднішній день, існує велика кількість патернів та реалізацій їх комбінацій. Для того, щоб обрати найбільш підходящу архітектуру пропонується розглянути з якими викликами доводиться справлятися розробникам. Перший і найбільш очевидний – це отримання залежностей від іншого класу. Зазвичай дана проблема вирішується прямим лінкування в едіторі, використанням патерну Singleton, статичних полів чи івентів. Всі недоліки цих підходів стають помітними у процесі наповнення проекту об’єктами та логікою. Альтернативою вище вказаним методам є Dependency Injection (DI), що надає змогу передати посилання через механізм інєкції залежностей. Інший виклик це підтримання структури проекту, мінімізація проблем порядку ініціалізації. Для цього потрібно чітко контролювати життєвий цикл застосунку, а також ігрових об’єктів. Для визначення етапів життєвого циклу підходить патерн State, оскільки кожен етап це окремий стан застосунку, в який ми можемо зайти та можемо вийти. Крім залежностей перед розробниками часто постає питання як використовувати один і той самий функціонал у різних куточках програми? Якщо логіка глобальна для всіх користувачів і 100 користувачів звернуться з запитами до однієї служби одночасно, то результат у кращому випадку отримають лише декілька. Саме цю проблем покликаний вирішити Service Locator у поєднанні з патерном State для забезпечення послідовного виконання запитів в порядку, визначеному станом [1-3].

Тому особливу увагу в роботі приділено інтеграції патерну Service Locator, DI та State, які в сукупності формують потужний інструментарій для створення сервіс-

орієнтованих застосунків. Ця комбінація патернів дозволяє розробникам створювати більш модульний, тестований та гнучкий код, що є особливо важливим у великих та складних проектах.

У роботі будуть детально розглянуті кожен з цих патернів, їхня роль та взаємодія в контексті розробки ігор на Unity3D. Будуть висвітлені основні виклики та проблеми, з якими можуть зіткнутися розробники при інтеграції цих патернів, а також надані рекомендації щодо їх ефективного використання.

Метою даної кваліфікаційної роботи є розробка методики проектування сервіс орієнтованих застосунків, яка дозволить створити гнучку та масштабовану архітектуру застосунку.

Для досягнення поставленої мети необхідно вирішити наступні **завдання**:

1. Провести доменний аналіз предметної області проектування застосунків у Unity3D, визначити переваги та недоліки кожного з підходів.
2. Розробити методику проектування сервіс-орієнтованих застосунків використовуючи паттерн «Service Locator».
3. Реалізувати запропоновану методику при розробці застосунку.

Об'єкт дослідження – процес проектування сервіс-орієнтованих застосунків за допомогою ігрового рушія Unity3D.

Предмет дослідження – фундаментальні методи та засоби та розробки сервіс-орієнтованих застосунків.

Методи дослідження.

1.Метод аналізу. Виявлення та аналіз існуючих недоліків та проблем, пов'язаних із застосуванням патерну "Service Locator." Ідентифікація ситуацій, де цей підхід може бути неефективним, і аналіз витрат на управління сервісами.

2. Метод порівняння. Порівняння використання патерну "Service Locator" з іншими популярними патернами проектування Unity3D. Оцінка переваг та недоліків патерну порівняно із стандартними рішеннями або іншими архітектурними підходами.

3. Метод синтезу. Узагальнення отриманих даних та знань для створення рекомендацій щодо оптимального використання патерну "Service Locator" у

розробці сервіс-орієнтованих застосунків. Формулювання принципів і кращих практик для використання патерну.

4. Практичне застосування. Створення сервіс-орієнтованого застосунку з паттерном "Service Locator", мінімізуючи недоліки цього патерну. Моделювання різних сценаріїв проектування прототипу для оцінки його здатності до розширення та гнучкості.

Практичне значення отриманих результатів: результати дослідження можуть бути використані як розробниками-початківцями, так і досвідченими фахівцями з розробки ігор та застосунків, які прагнуть оптимізувати свої процеси розробки та підвищити якість своїх продуктів.

Апробація отриманих результатів. Результати роботи були впроваджені у реальному проекті з розробки ігрового застосунку, що підтверджено актом впровадження результатів роботи (див. Додаток А).

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ СТВОРЕННЯ ДОДАТКІВ В UNITY 3D

1.1. Огляд ігрового рушія Unity3D

Unity – це найпопулярніший ігровий рушій у світі (рис. 1.1) [4]. Він має багато функцій та достатньо універсальний, щоб створити практично будь-яку гру. Unity популярний серед як аматорських розробників, так і професійних студій завдяки крос-платформенності [5]. Його використовували для створення таких ігор, як Pokemon GO, Hearthstone, RimWorld, Cuphead та багатьох інших.

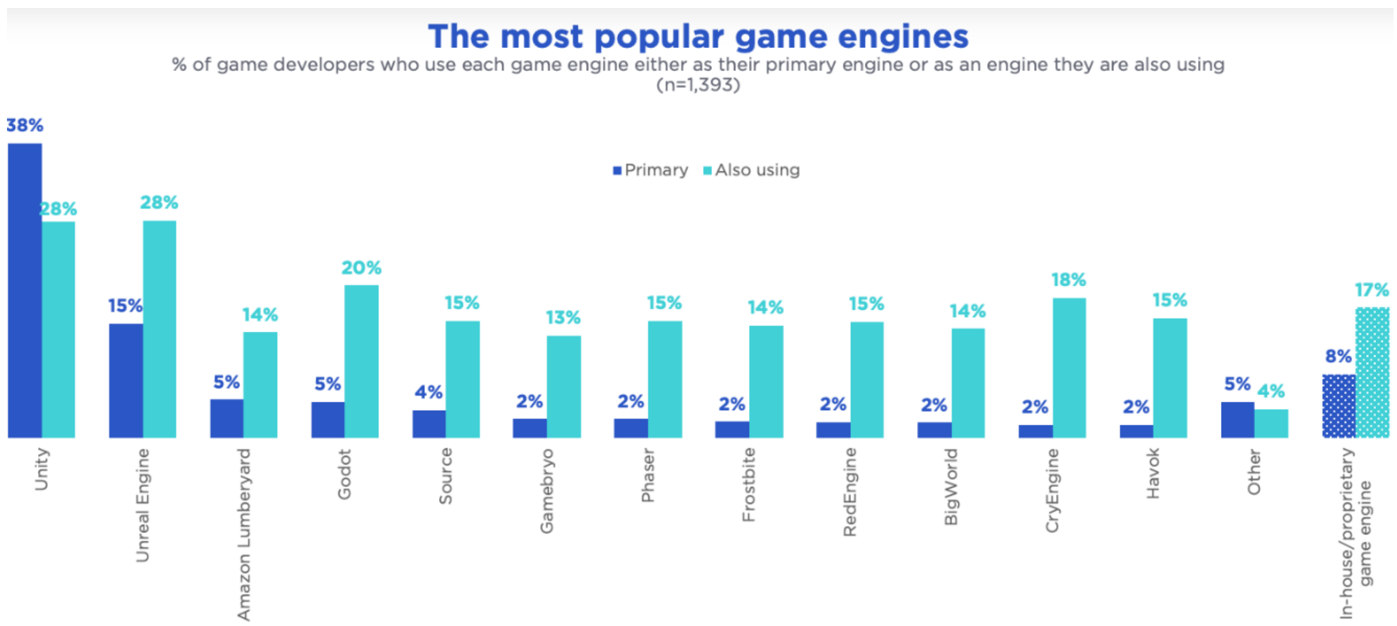


Рис. 1.1. Найбільш популярні у світі ігрові рушії

Хоча назва «Unity 3D» натякає на 3D, проте даний рушій також включає можливості для розробки 2D ігор. Сьогодні для написання коду можливо використовувати різні популярні IDE (Integrated Development Environment), що підтримують написання коду на C# (MS Visual Studio, Visual Studio Code, Rider, тощо). До того ж IDE інтегровані в Unity таким чином, що можуть аналізувати весь проект та рекомендувати розробниками потрібний API (Application Programming Interface) та навіть реалізацію методу, виходячи з його назви.

Для дизайнерів зручність полягає у тому, що рушій містить потужні інструменти для анімації, які спрощують створення власних 3D-послідовностей або розробку 2D-анімацій з нуля. У Unity майже все можна анімувати.

Оскільки Unity існує з 2005 року, він накопичив значну базу користувачів та неймовірну бібліотеку матеріалів. Unity пропонує не тільки відмінну документацію, але й відео та навчальні посібники, доступні в Інтернеті.

Unity – це програмне забезпечення, яке дозволяє створювати відеоігри без необхідності знати основні технології розробки ігор, тому потенційні розробники ігор можуть зосередитися лише на механіці гри та використовувати високорівневий підхід до створення ігор, використовуючи мову програмування C#. Термін "високорівневий" у даному випадку означає, що, коли гра створюється з використанням ігрового рушія, немає необхідності вирішувати проблему того, яким чином програмне забезпечення буде відображати («промальовувати») гру або яким чином буде відбуватись взаємодія з графічною картою для оптимізації швидкості гри. Тому використання ігрового рушія надає наступні можливості та переваги:

- Прискорене розроблення. Ігрові рушії дозволяють зосередитися на механіці гри. Оскільки для загальних механік і функцій вже доступні вбудовані бібліотеки, їх не потрібно створювати з нуля, і програмісти можуть використовувати їх безпосередньо та зекономити час (наприклад, для інтерфейсу користувача або штучного інтелекту).
- Інтегроване середовище розробки (IDE). IDE допомагає створювати, компілювати та керувати вашим кодом і включає корисні інструменти, які полегшують розробку і налагодження.
- Графічний користувацький інтерфейс (GUI, Graphical user interface). Хоча деякі ігрові рушії базуються на бібліотеках, більшість загальних ігрових рушіїв дозволяють користувачам створювати об'єкти без зусиль та виконувати загальні завдання, такі як трансформація, текстурування та анімація за допомогою функцій перетягування та розміщення. Ще однією перевагою такого програмного забезпечення є можливість

розуміти та переглядати, як буде виглядати гра без необхідності компілювати код (наприклад, через сцени).

- <u>Кросплатформенність.</u> За допомогою одного ігрового рушія можна експортувати створену гру на декілька платформ за допомогою одного кліку (наприклад, для WebGl, TV, iOS або Android), не переписуючи весь код повторно [6].

На ринку присутні різноманітні ігрові рушії, проте Unity виявився одним з найкращих серед них. Його використовують розробники ігор протягом багатьох років і завдяки ньому було створено успішні 3D та 2D ігри. З прикладами таких проектів можна познайомитися на офіційному сайті за посиланням [7].

Виділимо основні переваги ігрового рушія Unity з-поміж інших:

- можливість створювати як 2D, так і 3D ігри, розробляти різні жанри ігор, включаючи ігри від першої особи (FPS, Frame Rate), масові мультиплеєрні онлайн-рольові ігри (MMORPG, Massively Multiplayer Online Role-Playing Game), казуальні ігри, пригодницькі ігри та багато інших;
- можливість створювати ігри високої якості зі зручним інтерфейсом;
- експорт ігор на широкий спектр платформ, включаючи Android, iOS, Universal Windows Platform, Mac, WebGL, PS4, PS5, tvOS;
- включає всі необхідні інструменти, спрощує застосування корисних технік для підвищення якості гри;
- в залежності від звички розробника дозволяє використовувати будь яке інтегроване середовище розробки, що підтримує мову C# – Rider, Visual Studio, Visual Studio Code. Інтеграція цих середовищ допомагає писати код швидше та має набір модулів для рефакторингу та відладки коду;
- вбудовані модулі штучного інтелекту (наприклад, навігація по navmesh), які можна використовувати навіть без попередніх знань про штучний інтелект, світло, вбудовані об'єкти та кінцеву машину станів, яку можна застосовувати до персонажів для налаштування їх поведінки.

1.2. Огляд популярних патернів проектування, переваги та недоліки

1.2.1 Патерн «State»

Патерн «State» має три основні складові у своїй структурі (рис. 1.2):

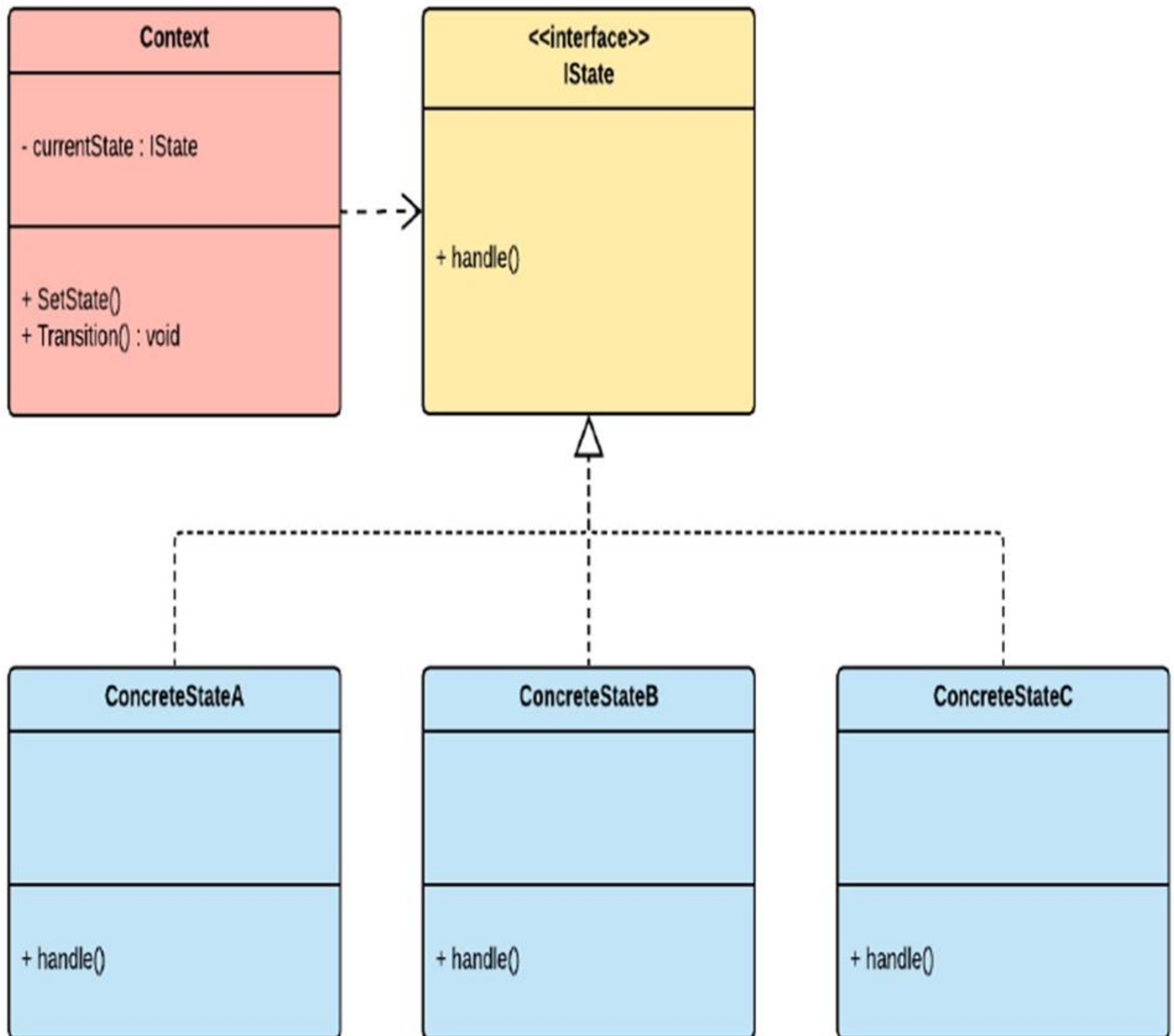


Рис. 1.2. UML-діаграма патерну «State»

Клас Context визначає інтерфейс, який дозволяє клієнту змінювати внутрішній стан об'єкта. Він також містить посилання на поточний стан.

Інтерфейс IState встановлює контракт на реалізацію для конкретних класів стану.

Конкретні класи ConcreteState реалізують інтерфейс IState та видають публічний метод handle(), який об'єкт Contex може викликати для ініціювання поведінки стану.

Щоб оновити стан об'єкта, клієнт може встановити очікуваний стан через об'єкт Context та запросити перехід до нового стану. Таким чином, Context завжди знає про поточний стан об'єкта, яким він керує. Однак, йому не потрібно знати кожен з існуючих конкретних класів стану. Можна додавати стільки класів стану, скільки необхідно, не змінюючи жодного рядка коду у класі Context.

Переваги використання патерну «State»:

- Інкапсуляція: патерн дає можливість реалізувати поведінку об'єкта зі станами у вигляді набору компонентів, які можуть бути динамічно призначені об'єкту при зміні станів.
- Підтримка: можливість легко впроваджувати нові стани, не змінюючи довгі умовні оператори або перенавантажені класи.

Однак, патерн «State» має свої обмеження, коли ми використовуємо його для управління анімованим персонажем [8]:

- Змішування: в первинному вигляді патерн «State» не пропонує рішення для змішування анімацій, що може становити проблему для досягнення плавного візуального переходу між анімованими станами персонажа.
- Перехід: у зазначеній вище реалізації патерну є можливість легко перемикатися між станами, але немає можливості визначати відношення між ними. Тому, для визначення переходів між станами на основі відношень та умов, доведеться написати набагато більше коду, наприклад, щоб стан бездіяльності переходив у стан ходьби, а потім стан ходьби переходив у стан бігу. Процес переходу має відбуватися автоматично та плавно, в прямому та зворотньому напрямках, в залежності від тригера чи умови. Це може потребувати великої кількості часу та ресурсів під час кодування.

Однак, зазначені вище обмеження можна подолати, використовуючи систему анімації Unity та її власну машину станів. Стани анімації можна легко визначити та прикріпити анімаційні кліпи, скрипти до кожного з налаштованих станів.

Нижче наведено список патернів, які є пов'язаними або альтернативними до патерну «State»:

1. Патерн «Blackboard» («Дошка оголошень») / патерн «Behavior Trees» («Дерева поведінки)»: використовуються для релізації складних інтелектуальних поведінок на основі штучного інтелекту для персонажів NPC (Non-Player Character). Наприклад, якщо є необхідність реалізувати штучний інтелект з динамічними поведінками прийняття рішень, то доцільно використати вказані вище патерни, оскільки вони дозволяють реалізовувати поведінку за допомогою дерева подій.
2. Патерн «FSM» (Finite-State Machine, тобто «Кінцевий автомат»): відмінність між «FSM» та патерном «State» полягає в тому, що «State» стосується інкапсуляції поведінки об'єкта, залежної від стану, а «FSM» більш глибоко пов'язаний з переходами між кінцевими станами на основі конкретних вхідних тригерів. Таким чином, «FSM» часто вважається таким, що більше підходить для реалізації систем, подібних до автоматів.
3. Патерн «Memento»: схожий на патерн «State», але з додатковою функцією, яка надає об'єктам можливість повернутися до попереднього стану; може бути корисним при реалізації системи, що потребує можливості скасувати зміну, внесену в неї саму.

1.2.2 Патерн «Event Bus»

Патерн «Event Bus» діє як центральний вузол, що керує певним списком глобальних подій, на які об'єкти можуть підписуватися або які вони можуть публікувати (рис. 1.3). Це найпростіший шаблон, пов'язаний з управлінням подіями. Він спрощує процес призначення об'єкту ролі «Subscriber» (підписник) або ролі «Publisher» (видавець) до одного рядка коду, що може бути корисним, коли

необхідно швидко отримати результати. Але, як і більшість простих рішень, даний патерн має недоліки та обмеження.

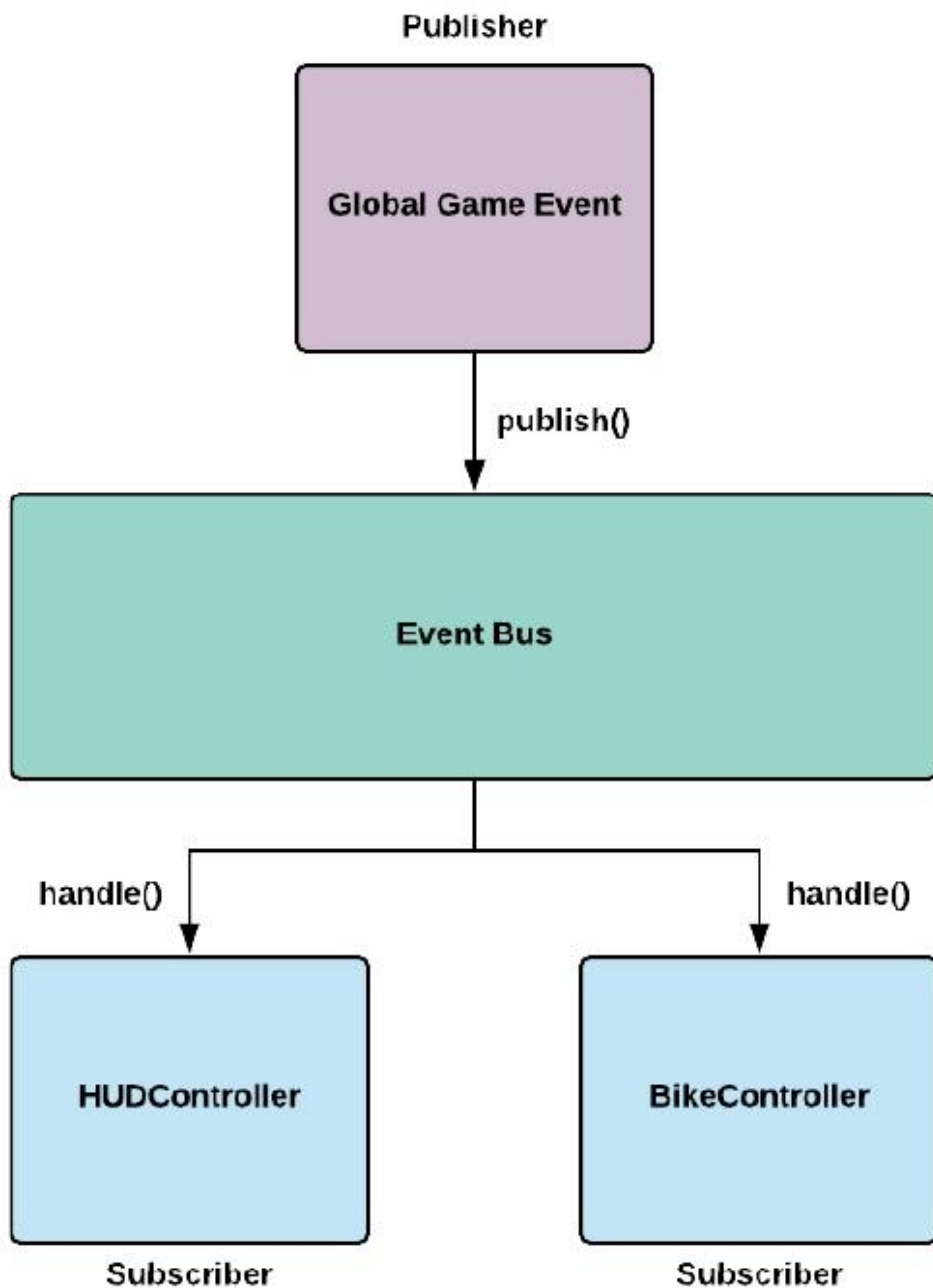


Рис. 1.3. UML-діаграма патерну «Event Bus»

Наведена вище (рис. 1.3) UML-діаграма патерну показує, що існують три основні компоненти:

- Видавці (publishers) – можуть публікувати певні типи подій, оголошені через шину подій, для підписників.
- Шина подій – цей об'єкт відповідає за координацію передачі подій між видавцями та підписниками.
- Підписники (subscribers) – дані об'єкти реєструють себе як підписників певних подій через шину подій.

Коли об'єкт «Publisher» викликає подію, він відправляє сигнал, який інші об'єкти «Subscribers» можуть отримати. Сигнал передається у формі повідомлення, яке може вказувати на те, що відбувається певна дія. В системі подій об'єкт транслює подію. Повідомлення отримують лише ті об'єкти, які підписані на неї, і вони обирають, яким чином її обробити. Тобто, така система події нагадує раптовий «сплеск» у спектрі радіосигналу, який виявляють лише ті приймачі, чиї антени налаштовані на певну частоту.

Патерн «Event Bus» є наближеним до системи обміну повідомленнями та патернів типу «Publish-Subscribe». Назва патерну «Event Bus» говорить про з'єднання між компонентами. Де компонентами будуть об'єкти, які можуть виступати у ролі «Subscriber» або у ролі «Publisher».

Отже, патерн «Event Bus» – це спосіб з'єднання об'єктів через події, використовуючи модель «Publish-Subscribe». Подібну модель можна реалізувати за допомогою шаблону «Observer» і нативних подій C#. Однак, ці альтернативи мають деякі недоліки. Наприклад, у типовій реалізації шаблону «Observer» може виникнути тісний зв'язок, оскільки «Observers» (слухачі) та «Publishers» (видавці) можуть стати залежними та знати однин про одного.

Але патерн «Event Bus» у тому вигляді, у якому він реалізовується в Unity, абстрагує і спрощує відносини між ролями «Subscriber» та «Publisher» таким чином, що вони повністю не знають один про одного. Ще однією перевагою патерну є те, що процес призначення ролі «Subscriber» та/або «Publisher» зменшується та спрощується до одного рядка коду.

Тобто патерн «Event Bus» є цінним шаблоном для дослідження та використання у реальних задачах. «Event Bus» виступає посередником між видавцями та підписниками.

До переваг патерну «Event Bus» варто віднести:

- Роз'єднаність: використання системи подій дозволяє роз'єднувати об'єкти, тобто об'єкти можуть бути пов'язані через події, а не напряму посилаючись один на одного, що зменшує ступінь їх зв'язності.
- Простота: шина подій спрощує механізм публікації або підписки, абстрагуючи його на подію від її клієнтів.

Серед недоліків патерну «Event Bus» варто відзначити наступні:

- Продуктивність: в основі будь-якої системи подій є механізм низького рівня, який керує обміном повідомлень між об'єктами, за рахунок чого вібувається невелика втрата продуктивності при використанні системи подій, але, залежно від цільової платформи, вона може бути мінімальною.
- Глобальність: завжди існує ризик при використанні глобально доступних змінних та станів, оскільки вони можуть ускладнити відлагодження та модульне тестування, однак даний недолік є скоріше контекстуальним, а не абсолютний.

Нижче наведено короткий список основних патернів, які варто враховувати при реалізації системи або механізму подій, але їх існує набагато більше:

- «Observer» – добре відомий патерн, де об'єкт (суб'єкт) підтримує список об'єктів («спостерігачів») та повідомляє їх про зміни внутрішнього стану. Це патерн, який слід розглядати, коли є необхідність встановити відношення один-до-багатьох між групою сутностей.
- «Event Queue» – дозволяє зберігати події, генеровані «видавцями» у черзі та пересилати їх своїм «підписникам» у зручний час. Такий підхід роз'єднує тимчасові відносини між «видавцями» та «підписниками».
- «ScriptableObjects» – забезпечує можливість створити систему подій з ScriptableObjects у Unity. Основна перевага цього підходу полягає в

тому, що він спрощує створення нових користувацьких ігрових подій. Його часто використовують, якщо виникає потреба побудувати масштабовану та налаштовувану систему подій.

1.2.3 Патерн «Command»

Патерн «Command» – це поведінковий шаблон проектування, який перетворює запити в об'єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

Патерн «Command» дозволяє розділити об'єкт, який викликає операцію, від того, хто знає, як її виконати. Іншими словами, обробнику користувацького вводу не потрібно знати, яка точна дія має бути виконана, коли гравець натискає пробіл. Йому просто потрібно переконатися, що виконується правильна команда.

На рис. 1.4 наведено UML-діаграму, за допомогою якої можна виділити основні класи даного патерну:

- Invoker (викликач) –об'єкт, який знає, як виконати команду і може вести облік виконаних команд.
- Receiver (отримувач) – тип об'єкта, який може отримувати команди і виконувати їх.
- CommandBase (базова команда) – абстрактний клас, який повинен успадковувати деякий індивідуальний клас ConcreteCommand (певна конкретна команда), який також відкриває метод Execute(), що може бути викликаний для виконання певної команди за допомогою Invoker.

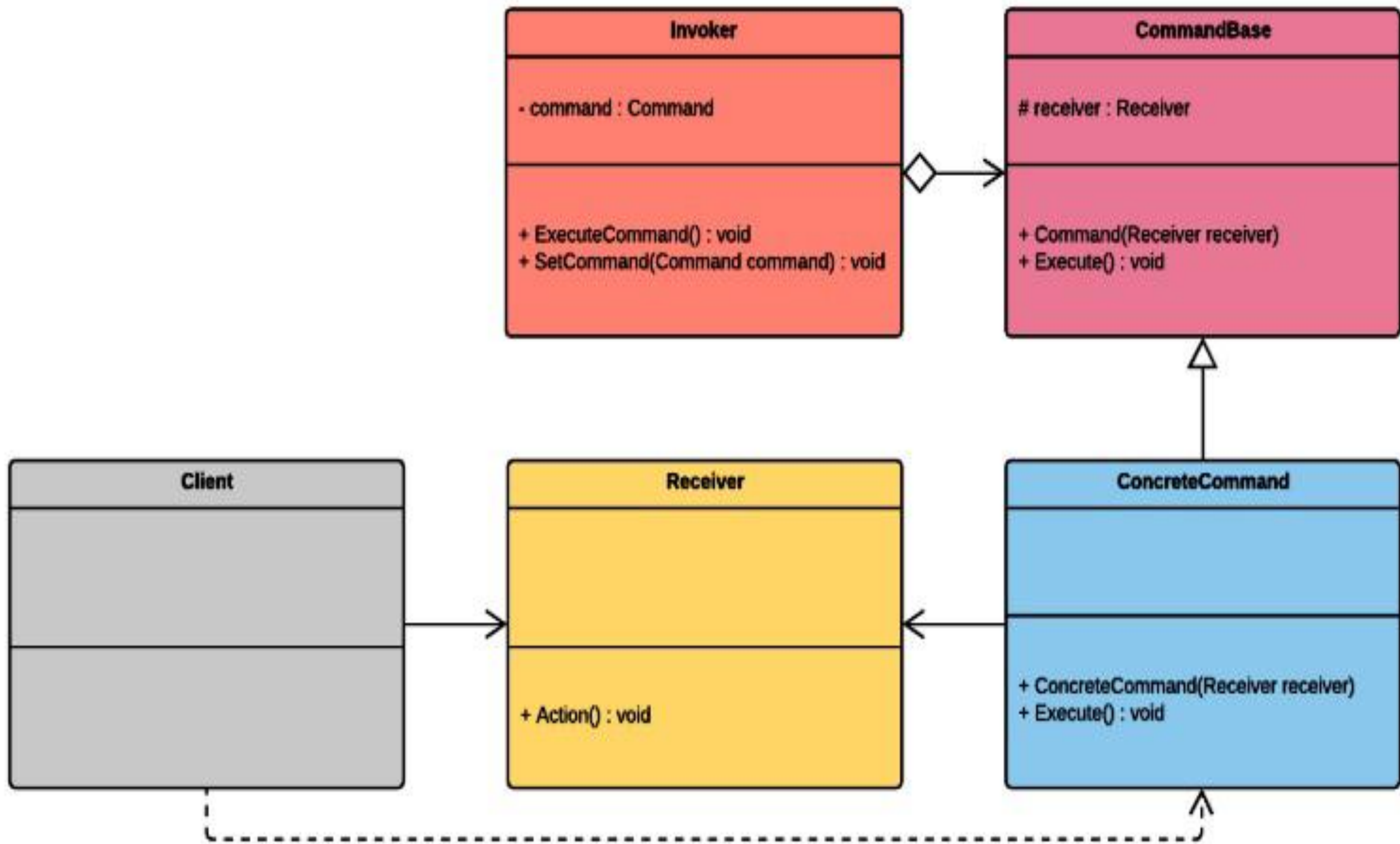


Рис.1.4. UML-діаграма патерну «Command»

Нижче наведено основні переваги патерну «Command»:

- Розділення: патерн дозволяє розділити об'єкт, який викликає операцію, від того, хто знає, як її виконати; даний рівень відокремленості дозволяє додати посередника, який зможе вести облік та послідовність операцій.
- Послідовність: патерн полегшує процес створення черги вводу користувача, що дозволяє реалізувати можливості скасування/повтору дій, макросів та черг команд.

Потенційним недоліком патерну «Command» є складність реалізації. Для реалізації даного патерну потрібно багато класів, оскільки кожна команда є класом сама по собі. Також потрібно мати відмінне розуміння принципів роботи патерну для підтримки коду, побудованого за його допомогою, що у більшості випадків не становить серйозної проблеми, але використання патерну без конкретної мети може ускладнити та абстрагувати кодову базу.

1.2.4 Патерн «Object pool»

Основна концепція патерну «Object pool» є простою — набір (ObjectPool) у вигляді контейнера зберігає колекцію ініційованих об'єктів у пам'яті. Клієнти можуть запитувати у набору (пулу) екземпляр об'єкта певного типу; якщо такий є в наявності, він буде вилучений з набору та переданий клієнту. Якщо в певний момент немає достатньої кількості об'єктів у наборі, нові будуть створені динамічно.

Об'єкти, що покидають пул, намагаються повернутися до нього, коли вони більше не використовуються клієнтом. Якщо в наборі більше немає місця, він знищуватиме екземпляри об'єктів, що намагаються повернутися. Таким чином, пул постійно поповнюється, може бути лише тимчасово спорожнений, але ніколи не переповнюється. Отже, використання його пам'яті є стабільним.

На рис. 1.5 наведено UML-діаграму патерну «Object pool», яка ілюструє обмін між клієнтом та пулом об'єктів:

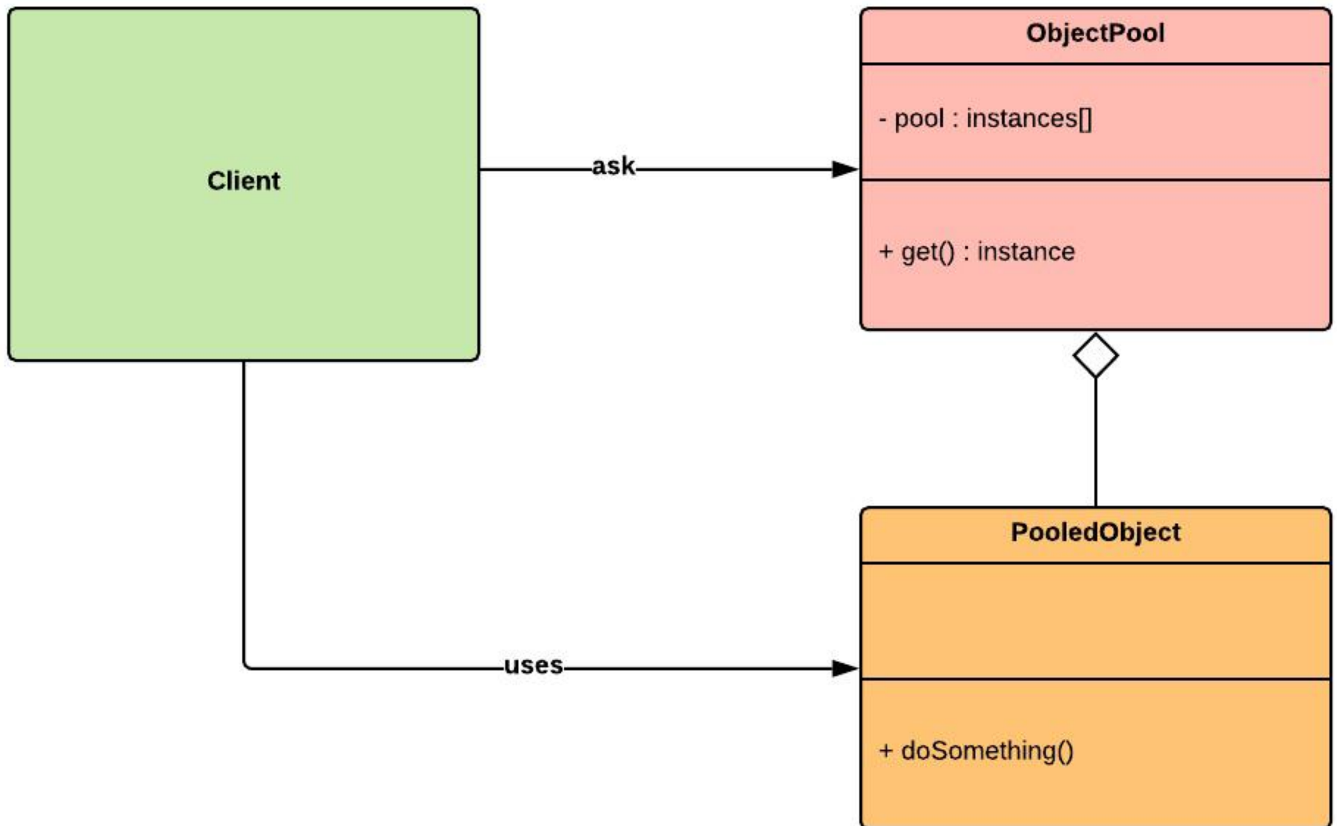


Рис. 1.5. UML діаграма патерну «Object pool»

З діаграми (рис. 1.5) видно, що пул об'єктів служить клієнту, надаючи йому доступ до пулу екземплярів об'єктів певного типу, наприклад, клієнтом може бути генератор, який запитує у пулу об'єктів екземпляри конкретного типу ворога.

Переваги патерну «Object pool»:

- Передбачуване використання пам'яті: за допомогою «Object pool» ми можемо виділити пам'ять у передбачуваний спосіб для утримання певної кількості екземплярів певного типу об'єкта.
- Підвищення продуктивності: маючи об'єкти, що вже ініційовані в пам'яті, можна уникнути витрат на ініціалізацію нових.

Нижче наведено потенційні недоліки патерну «Object pool»:

- Накладення на вже керовану пам'ять: сучасні керовані мови програмування, такі як C#, оптимально контролюють виділення пам'яті.
- Непередбачувані стани об'єктів: якщо патерн неправильно обробляється, об'єкти можуть бути повернуті до набору у своєму поточному стані, а не

в початковому. Це може становити проблему у випадку, коли об'єкт у пулі є таким, який можна пошкодити або знищити. Наприклад, ворожа сутність, яку щойно вбив гравець, при поверненні її до пулу без відновлення її здоров'я, з'явиться на сцені вже пошкодженою, коли «Object pool» знову поверне її для клієнта.

Проте патерн «Object pool» не варто використовувати, якщо, наприклад, є сутності, які потрібно згенерувати лише один раз на карті, зокрема такі, як, наприклад, «остаточний бос», додавання його до «Object pool» є марнуванням пам'яті, яка могла б бути використаною для чогось більш корисного.

Також варто вказати, що «Object pool» – це не кеш, хоча він має схожу мету – повторне використання об'єктів. Основна різниця полягає в тому, що «Object pool» має механізм, у якому сутності автоматично повертаються до набору після використання, і добре реалізований «Object pool» керує створенням та видаленням об'єктів в залежності від доступного розміру пулу.

У випадку, якщо є сутності, такі як, наприклад, кулі, частинки і ворожі персонажі, які часто генеруються та знищуються під час ігрового процесу, «Object pool» може зняти деяке навантаження, яке накладається на ЦПУ (центральний процесор), зменшуючи кількість звернень, пов’язаним зі створенням або знищенням об’єктів. Таким чином ЦПУ зможе зберегти обчислювальну потужність для більш критичних завдань.

1.2.5 Патерн «Observer»

Основна мета патерну «Observer» полягає в тому, щоб встановити взаємозв'язок типу «один з багатьма» між об'єктами, де один об’єкт діє як суб'єкт, а інші беруть на себе роль спостерігачів. Потім суб'єкт бере на себе «відповідальність» проінформувати спостерігачів у випадку, коли щось всередині нього змінюється і може зацікавити їх.

Такий процес дещо схожий на модель «Publisher-Subscriber», в якому об'єкти підписуються і очікують повідомлення про конкретні події. Ключова відмінність

полягає в тому, що суб'єкт і спостерігачі знають один про одного в моделі спостерігача, тому вони можуть бути пов’язані.

Розглянемо UML-діаграму типової реалізації шаблону «Observer» (рис. 1.6), щоб побачити, яким чином це буде реалізовано у коді:

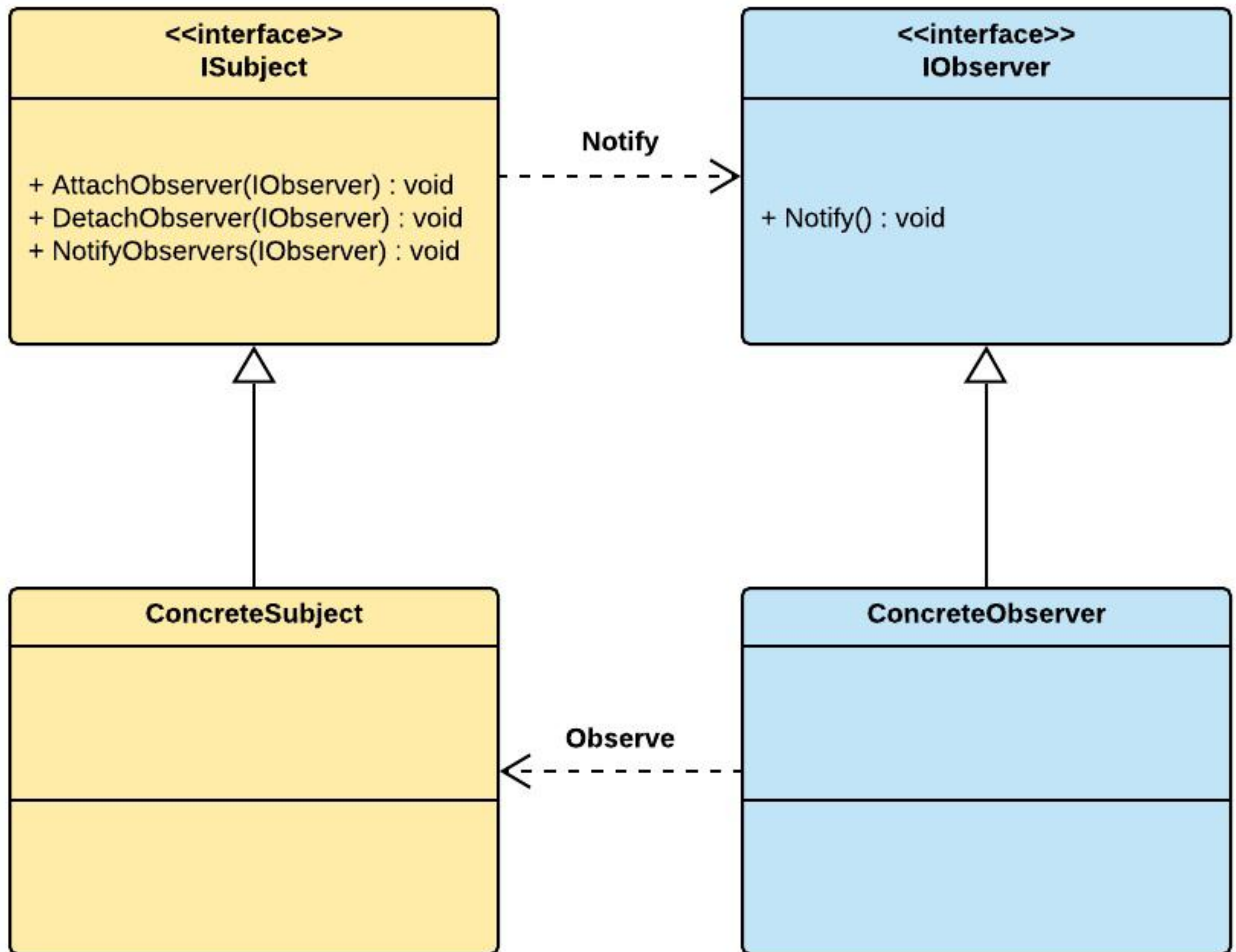


Рис 1.6. UML-діаграма патерну «Observer»

На рис. 1.6 показано, що ISubject (суб'єкт) і IObserver (спостерігач) мають відповідні інтерфейси, які вони реалізують, але найважливішим для аналізу є ISubject, який включає наступні методи:

- AttachObserver(): за допомогою цього методу можна додати об'єкт спостерігача до списку спостерігачів, яких слід сповістити.
- DetachObserver(): метод вилучає спостерігача зі списку спостерігачів.

- NotifyObservers(): метод сповіщає про всі об'єкти, які були додані до списку спостерігачів суб'єкта.

Об'єкт, який бере на себе роль спостерігача, повинен впроваджувати публічний метод Notify(), який буде використано суб'єктом, щоб сповістити його, коли він змінить стан.

Переваги патерну «Observer»:

- Спостерігач дозволяє суб'єкту додавати стільки об'єктів, скільки йому потрібно як спостерігачу, але він також може динамічно їх видаляти під час запуску.
- Основна перевага моделі спостерігача полягає в тому, що вона елегантно вирішує проблему впровадження системи обробки подій, в якій існує взаємозв'язок між об'єктами.

Недоліки патерну «Observer»:

- Патерн не гарантує дотримання послідовності порядку слідування, в якому спостерігачі отримують сповіщення. Отже, якщо два або більше об’єктів спостерігача мають спільні залежності і повинні працювати разом у певній послідовності, то модель спостерігача у своїй оригінальній формі не призначена для обробки цього типу контексту виконання.
- Патерн може спричинити витоки пам'яті, оскільки суб'єкт має посилання на своїх спостерігачів. Якщо програма буде виконана неправильно, а об'єкти спостерігача не будуть правильно відокремлені і вилучені, коли у них більше немає потреби, це може призвести до проблем зі збиранням «сміття», а деякі ресурси не будуть звільнені взагалі.

1.2.6 Патерн «Visitor»

Основна мета шаблону «Visitor» – об’єкт, до якого можна отримати доступ, дозволяє «Visitor» діяти над певним елементом його структури. Тобто даний процес дає можливість відвідуваному об'єкту отримувати нові функціональні можливості від відвідувачів без прямих змін. Таким чином, за допомогою шаблону «Visitor»

можна «проникнути» до структури об'єкта, працювати з його елементами і розширювати його функціональність, не змінюючи його.

Розглянемо UML-діаграму патерну «Visitor» (рис. 1.7):

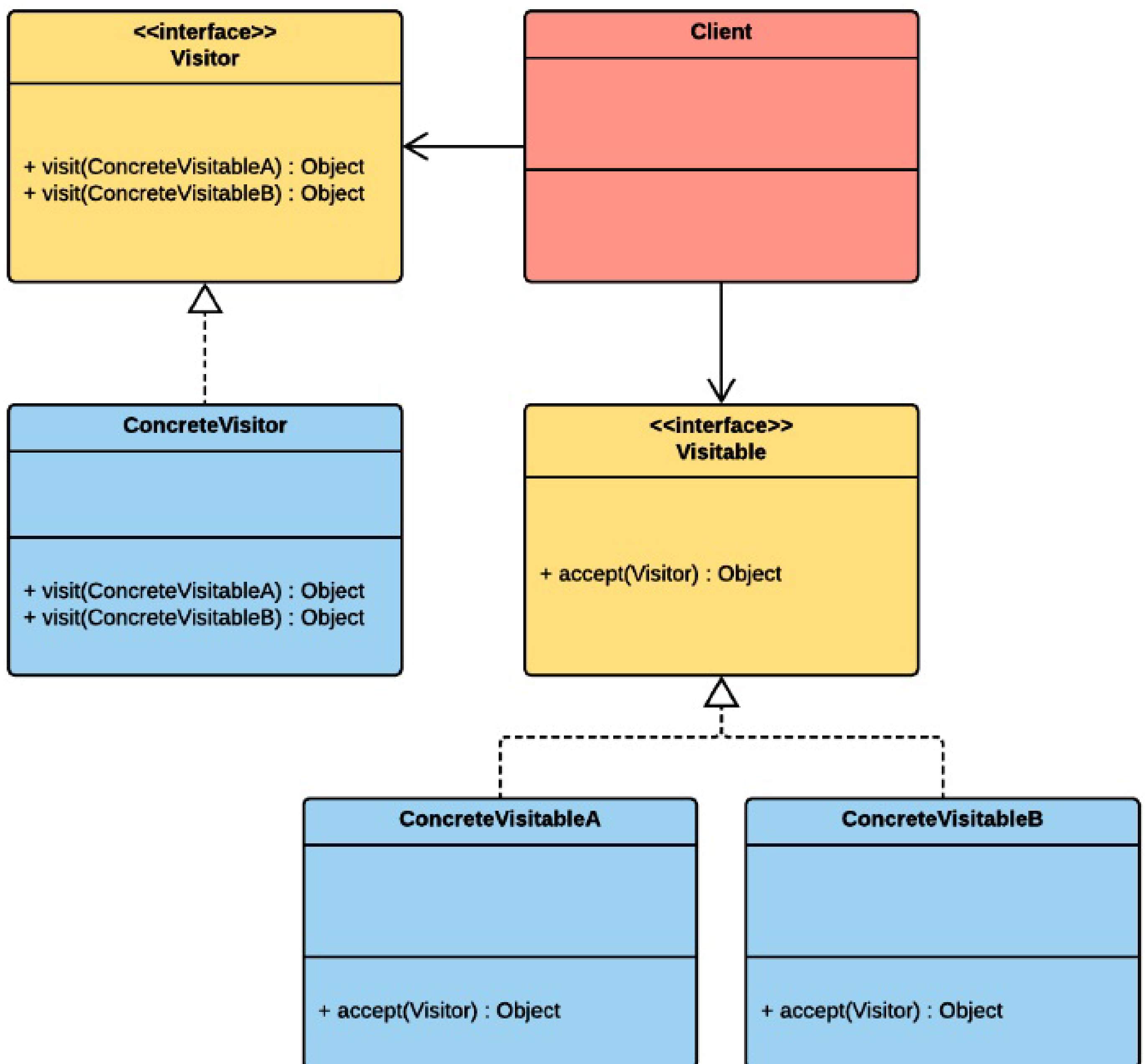


Рис 1.7. UML-діаграма патерну «Visitor»

Існують два ключові учасники цієї моделі (рис. 1.7):

- IVisitor – це інтерфейс, який повинен реалізувати клас, що бажає бути відвідувачем. Класу відвідувачів доведеться впроваджувати метод відвідувача за відвідуваним елементом.
- IVisitable – це інтерфейс, який повинні впроваджувати класи, що бажають стати відвідуваними. Він включає в себе метод accept(), який пропонує точку входу для об'єкта відвідувача, щоб прийти і відвідати.

Переваги патерну «Visitor»:

- Можливо додавати нові поведінки, які можуть працювати з об'єктами різних класів, не змінюючи їх безпосередньо. Цей підхід дотримується принципу об'єктно-орієнтованого програмування «Open/Closed», який стверджує, що суб'єкти повинні бути відкриті для розширення, але закриті для модифікації.
- Одиночна відповідальність: модель відвідувача може дотримуватися принципу «єдиної відповідальності» в тому сенсі, що можна мати об'єкт (візитний), який містить дані, а інший об' єкт (відвідувач) відповідає за введення конкретної поведінки.

Недоліки патерну «Visitor»:

- Доступність: відвідувачам може не вистачати необхідних прав для доступу до конкретних приватних полів і методів елементів, які вони відвідують, що породжує проблему викриття більшої кількості публічних об'єктів у класах.
- Складність кодової бази, яку деякі можуть вважати «заплутаною» за відсутності достатньої кількості знань щодо структури і тонкощів патерну.

1.2.7 Патерн «Strategy»

Основна мета патерну «Strategy» полягає в тому, щоб відкласти рішення про те, яку поведінку використовувати під час виконання. Це стало можливим завдяки тому, що модель «Strategy» дозволяє нам визначити набір поведінок, які закладені в

окремі класи, що називають стратегіями. Кожна стратегія є взаємозамінною і може бути присвоєна об'єкту контексту цілі для зміни його поведінки.

Візуалізуємо ключові елементи шаблону за допомогою UML-діаграми (рис. 1.8).

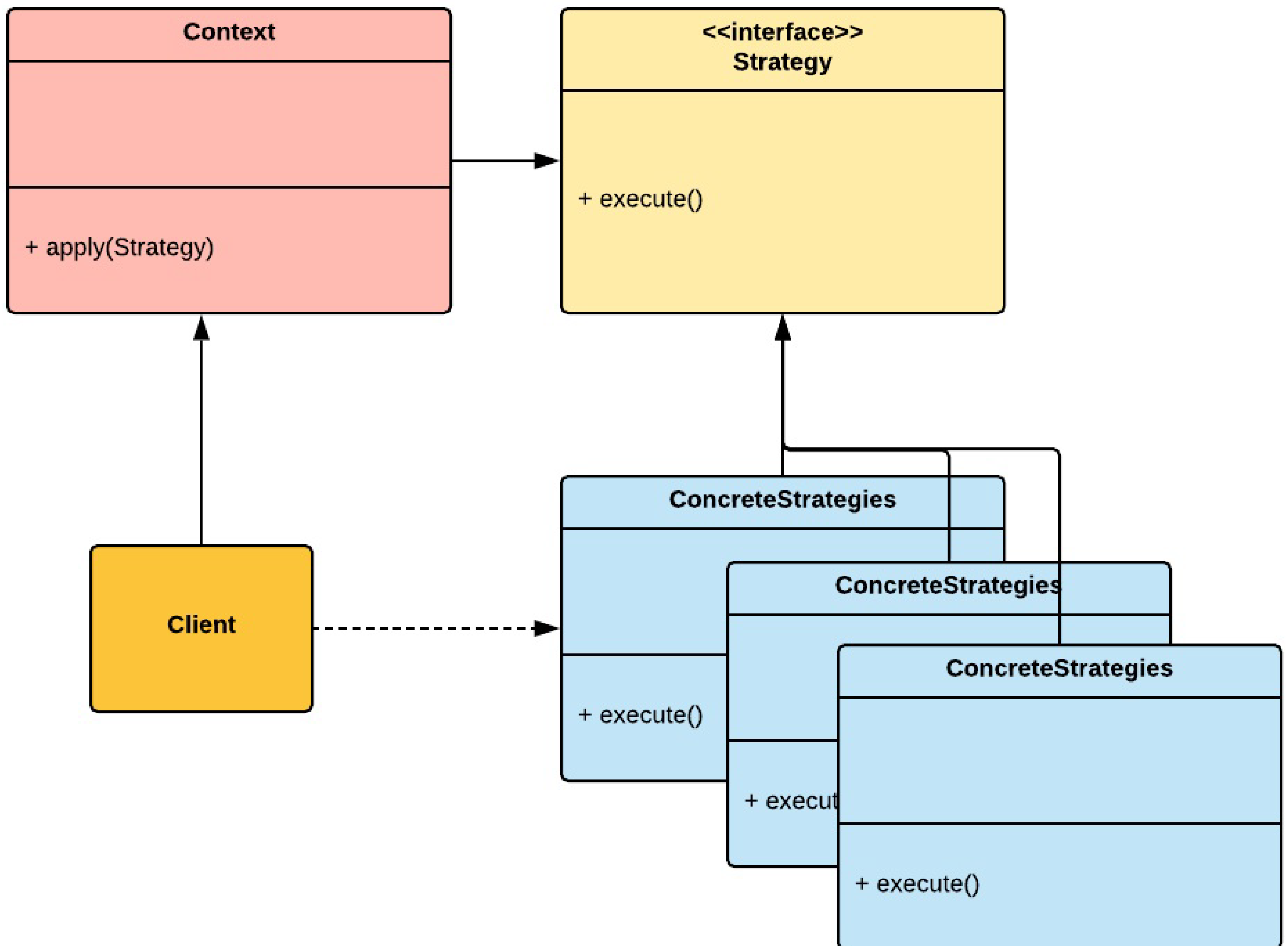


Рис 1.8. UML-діаграма патерну «Strategy»

Основні складові моделі (рис. 1.8):

- Клас Context – клас, який використовує різні конкретні класи Strategy і взаємодіє з ними через інтерфейс IStrategy.
- Інтерфейс IStrategy є спільним для всіх класів конкретних стратегій. Він розкриває метод, який клас Context може використовувати для виконання стратегії.

- Класи ConcreteStrategies, також відомі «Strategies», є конкретними реалізаціями варіантів алгоритмів поведінки, які можуть бути застосовані до об'єкта Context під час запуску.

Переваги патерну «Strategy»:

- Інкапсуляція: патерн вимагає, щоб варіації алгоритмів бути інкапсульованими в окремі класи, що допомагає уникнути використання довгих умовних тверджень, зберігаючи при цьому структуру коду.
- Час виконання: патерн реалізує механізм, який дозволяє змінювати алгоритми, які об'єкт використовує під час виконання; такий підхід робить об'єкти більш динамічними і відкритими для розширення.

Недоліки патерну «Strategy»:

- Клас Client повинен знати про окремі стратегії та варіації в алгоритмі, який ці стратегії реалізують, щоб знати, які з них вибрати; таким чином, Client відповідає за забезпечення того, щоб об'єкт поводився так, як очікується протягом його життєвого циклу.
- Заплутаність: оскільки патерн «Strategy» та патерн «State» дуже схожі за структурою, але мають різні призначення, може виникати плутанина при виборі того, який з них використовувати і в якому контексті.

1.2.8 Патерн «Decorator»

Патерн «Decorator» – це шаблон, який дозволяє додавати нову функціональність до існуючого об'єкта, не змінюючи його. Для цього створюється «клас-обгортка» над базовим класом. І за допомогою цього механізму можливо змінювати поведінку об’єкту.

Розглянемо UML-діаграму патерну «Decorator» (рис. 1.9):

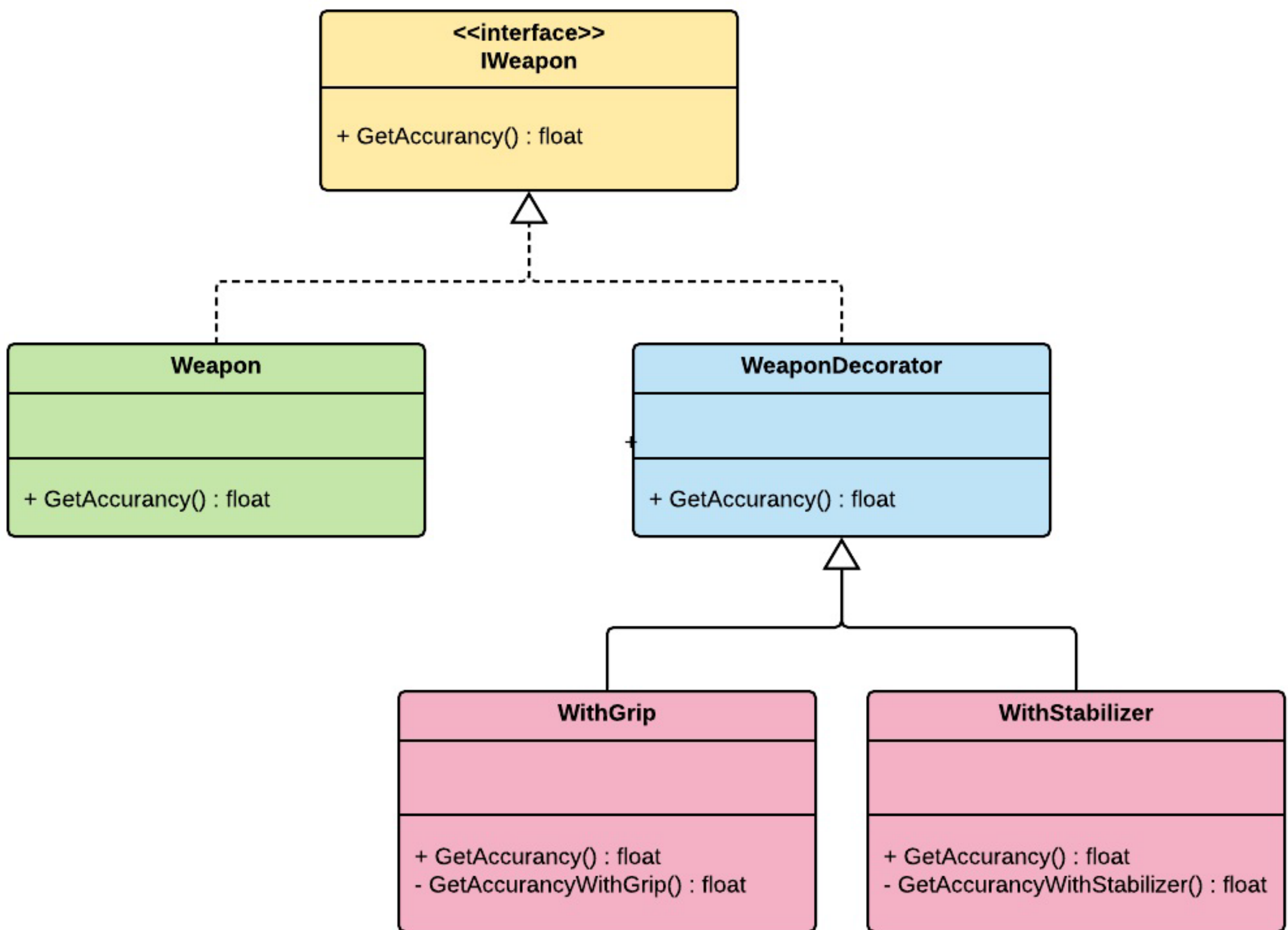


Рис 1.9. UML-діаграма патерну «Decorator»

Інтерфейс IWeapon визначає контракт реалізації, який буде підтримувати послідовний підпис методу між об'єктом і його декораторами.

WeaponDecorator обгортає цільовий об’єкт, а класи декораторів (WithGrip і WithStabilizer) доповнюють його, посилюючи або змінюючи його поведінку.

Підписи методу і загальна структура об'єкта не змінюються під час процесу, змінюються тільки його поведінка або значення властивостей. Таким чином, можна легко зняти зміни з об'єкта і повернути його до початкової форми.

Більшість прикладів даного патерну залежить від конструктора класу. Проте, рідні базові класи Unity API, такі як MonoBehaviour і ScriptableObject, не використовують концепцію конструктора для ініціалізації екземпляру об'єкта. Замість цього, у випадку MonoBehaviours, рушій забезпечує ініціалізацію класів, які

прикріплені до «GameObjects». Очікується, що будь-який код ініціалізації буде реалізовано у викликах Awake() або Start().

Переваги патерну «Decorator»:

- Альтернатива підкласуванню: наслідування – статичний процес, який, на відміну від «Decorator», не дозволяє розширювати поведінку існуючого об'єкта під час запуску. Можна замінити один екземпляр іншим з тим самим початковим класом, який має потрібну поведінку, тому модель «Decorator» є більш динамічною альтернативою підкласуванню, яка долає межі спадкування.
- Динаміка часу виконання: модель «Decorator» дозволяє додавати функціональність об'єкту під час запуску, прикріплюючи до нього декоратори. Зворотний варіант також можливий, що дає змогу відновити об'єкт до його початкової форми, видаливши його декоратори, якщо є така необхідність.

Недоліки патерну «Decorator»:

- Складність відносин: слідкувати за ланцюгом ініціалізації і відносинами між декораторами може стати дуже складним, якщо навколо об’єкта є декілька шарів декораторів.
- Складність коду: залежно від того, яким чином впроваджений патерн, він може додати складнощі кодової базі, для вирішення чого може знадобитися підтримка кількох невеликих класів «Decorator», що реалізується шляхом розділення великого класу на менші.

1.2.9 Патерн «Adapter»

Патерн «Adapter» має змогу адаптувати два несумісні інтерфейси. Тобто, як і адаптер в класичному розумінні, він не змінює те, що він регулює, а перетинає один інтерфейс з іншим. Цей підхід може бути корисним у випадку використання застарілого коду, який неможна переписати через його «крихкість», або у тому випадку, коли необхідно додати функції до бібліотеки третьої сторони, але немає мети змінювати її, щоб уникнути проблем під час оновлення.

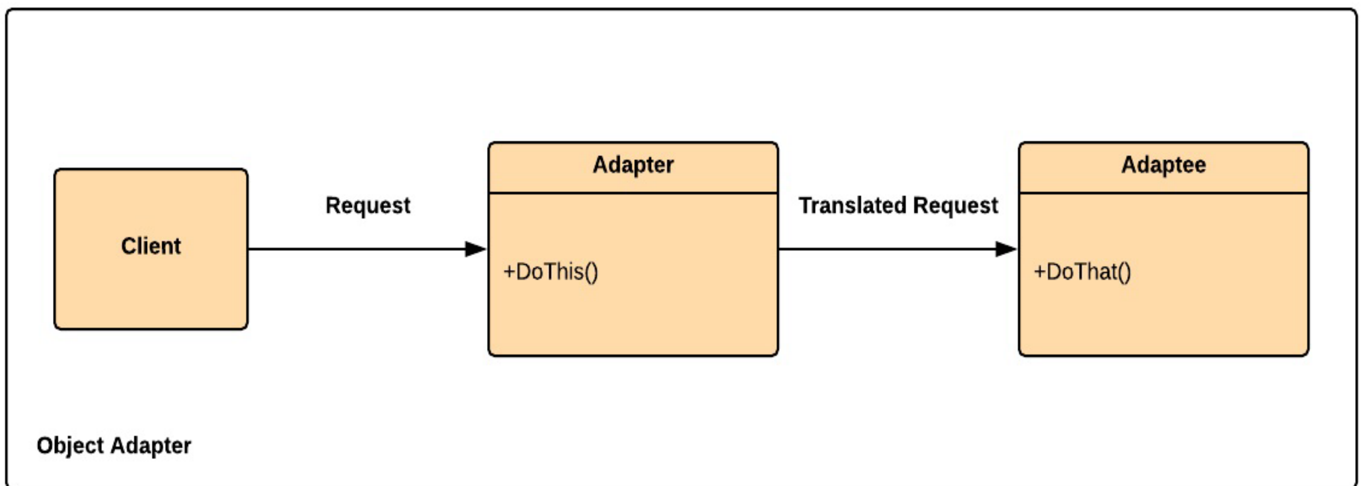
Нижче наведено короткий опис двох основних підходів до реалізації моделі «Adapter»:

- Адаптер об' єкта: шаблон використовує композицію об'єкта, а адаптер діє як обгортка навколо адаптованого об'єкта. Такий підхід може бути корисним у випадку, коли є певний клас, який не має потрібних методів, але який не можна змінити безпосередньо. Адаптер об'єкта приймає методи оригінального класу і адаптує їх до потрібних.
- Адаптер класу: шаблон використовує успадкування для адаптації інтерфейсу існуючого класу до інтерфейсу іншого класу, що може бути доцільним у випадку необхідності налаштування класу з метою можливості працювати з іншими класами, але не без змоги змінювати його інтерфейс безпосередньо.

Розглянемо UML-діаграму об’єкта і класу «Adapter» (рис. 1.10). В обох випадках Class Adapter розташовується між клієнтом і адаптованим суб'єктом (Adaptee). Але Adapter класу встановлює відносини з Adaptee через наслідування. На відміну від цього, Object Adapter об'єктів використовує композицію для обертання інстанції адаптера для адаптації.

В обох випадках суб'єкт, який адаптується, не змінюється. Крім того, Client не має відомостей про те, що саме адаптується, але він знає, що має послідовний інтерфейс для спілкування з адаптованим об'єктом.

Композиція і успадкування є способами визначення відносин між об'єктами. І ця різниця у структурі відносин частково визначає різницю між об'єктом і класом Adapter.



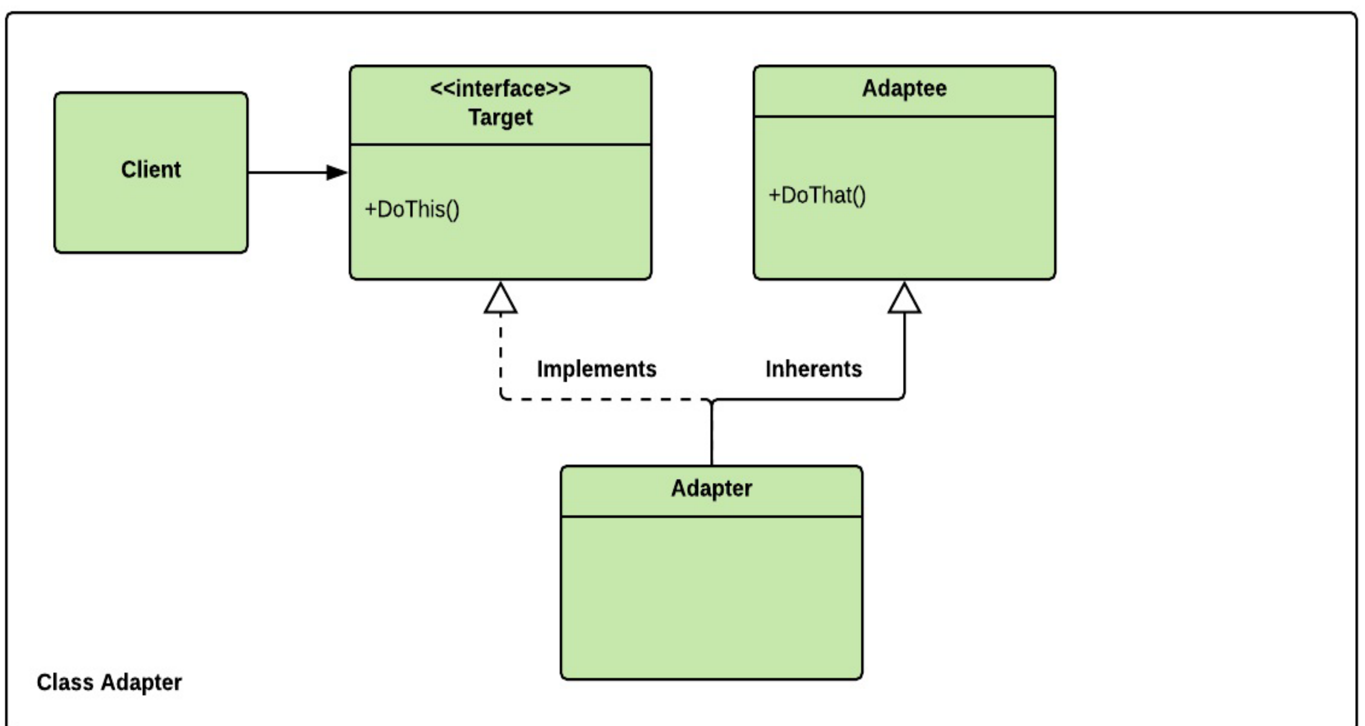


Рис 1.10. UML-діаграма патерну «Adapter»

Також патерн «Adapter» іноді можна плутати з патерном «Facade» (див. п. 1.2.10). Проте основна відмінність між ними полягає в тому, що модель «Facade» встановлює спрощений фронтальний інтерфейс до складної системи. Але «Adapter» може адаптувати несумісні системи, зберігаючи при цьому послідовний інтерфейс для клієнта.

Обидва шаблони пов'язані, оскільки вони є структурними шаблонами, але мають зовсім різні цілі.

Переваги патерну «Adapter»:

- Адаптація без модифікації: модель пропонує стандартний підхід до адаптації старого або стороннього коду без його модифікації.
- Повторна експлуатаційність і гнучкість: шаблон дозволяє продовжувати використання застарілого коду на нових системах з мінімальними змінами.

Недоліки патерну «Adapter»:

- Можливість використання застарілого коду з новими системами є економічно ефективною, але в довгостроковій перспективі це може стати проблемою, оскільки старий код може обмежувати можливості для оновлення, оскільки застарілий код в певний момент може стати несумісним із новими версіями Unity або бібліотеками третіх осіб.
- Можливе незначне погіршення продуктивності, оскільки у деяких випадках відбувається перенаправлення викликів між об’єктами, проте такий вплив зазвичай занадто малий, щоб стати великою проблемою.

1.2.10 Патерн «Facade»

Основна мета патерну «Facade» полягає в тому, щоб запропонувати спрощений фронтальний інтерфейс, який абстрагує складні внутрішні функціонування складної системи.

Цей підхід є корисним для розробки ігор, оскільки ігри складаються із складних взаємодій між різними системами. Наприклад, написаний таким чином код імітує поведінку та взаємодію основних компонентів двигуна транспортного засобу, а потім пропонує простий інтерфейс для взаємодії з цілою системою.

Назва патерну обумовлена тим, що прослідковується деяка аналогія з фасадом будівлі: певна зовнішня частина, що приховує складну внутрішню структуру. На відміну від будівельної архітектури, у розробці програмного забезпечення метою таким чином є спрощення, а не оздоблення та прикрашання.

Нижче наведено UML-діаграму патерну «Facade» (рис. 1.11), з якої видно, що патерн «Facade» зазвичай обмежується одним класом, який діє як спрощений інтерфейс до колекції взаємодіючих підсистем.

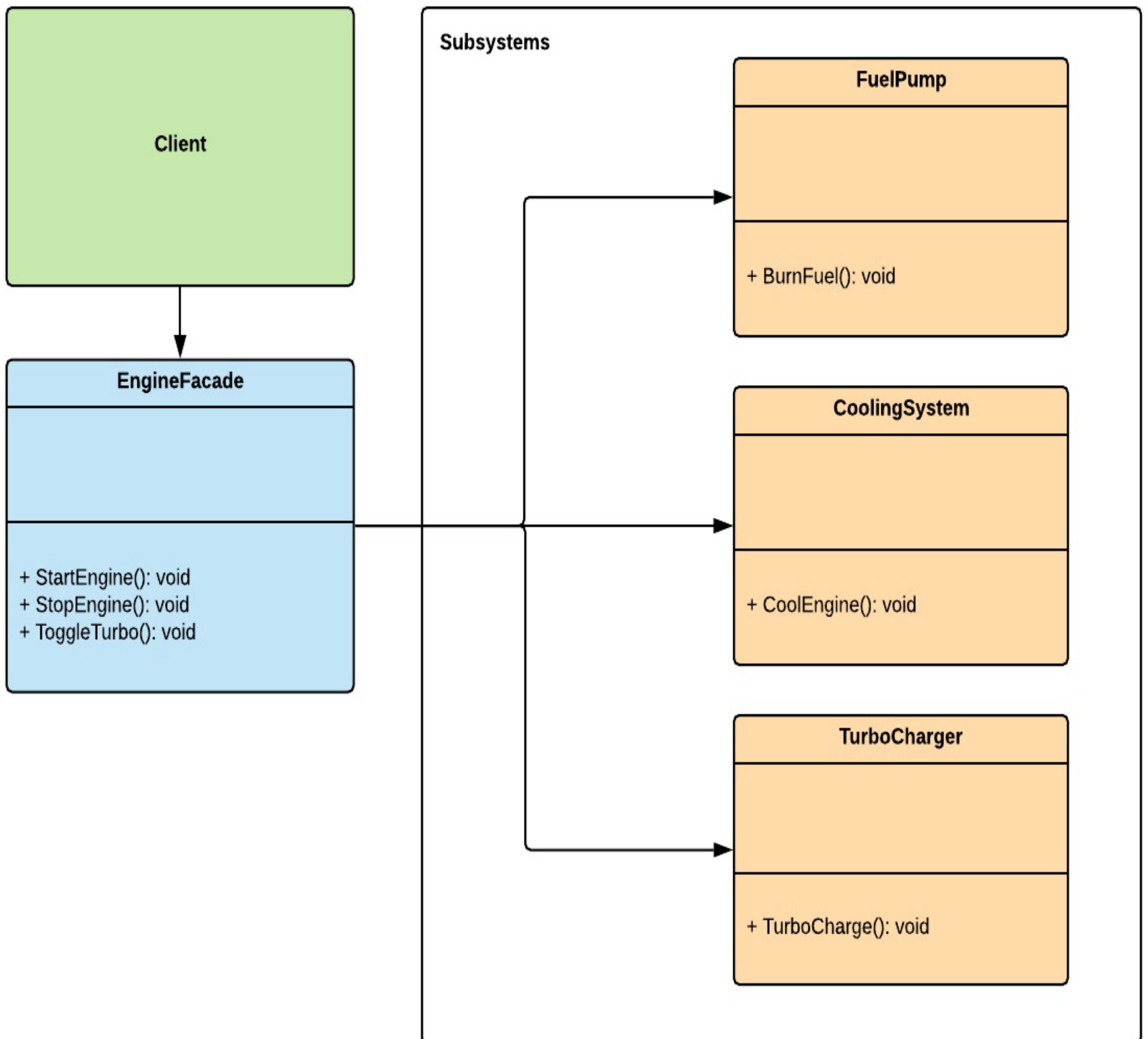


Рис 1.11. UML-діаграма патерну «Facade»

EngineFacade виступає в якості інтерфейсу для різних компонентів ігрового рушія, після чого клієнт не знає, що відбувається за сценами під час виклику StartEngine() на Engine Facade. Також клієнт не знає, з яких компонентів складається рушій і як до них дістатися. Клієнт знає лише про той функціонал, який він

потенційно може використовувати, не маючи потреби знати, яким чином він реалізований.

Переваги патерну «Facade»:

- Спрощений інтерфейс до складного коду: клас патерну «Facade» приховуватиме складність від клієнта, забезпечуючи спрощений інтерфейс для взаємодії із складною системою.
- Легке перезавантаження: простіше перетворювати код, який ізольований з використанням патерну, оскільки інтерфейс системи залишається узгодженим з клієнтом, поки його компоненти модифікуються за сценами.

Недоліки патерну «Facade»:

- Використання патерну з метою приховання недосконалостей коду за фронтальним інтерфейсом, що слугує своєрідною «обгорткою», дозволить вирішити проблему на певний час. Проте в довгостроковій перспективі такий підхід може стати суттєвою проблемою, через накопичення недосконалостей коду, що в подальшому стає складніше виправляти, наприклад, через відсутність достатньої кількості часу у розробника.
- Надмірне використання: глобально доступні класи менеджерів, які виступають в якості «фасадів» для основних систем, є популярними серед розробників Unity; вони часто реалізують їх, поєднуючи шаблони «Singleton» і «Facade». На жаль, така практика може призводити великої кількості класів менеджерів, що тісно пов’язані між собою, що ускладнює процеси дебагінгу, рефакторингу та юніт-тестування коду.

1.3 Огляд патерну «Service Locator», сутність сервісів

У Unity, класи MonoBehaviour не мають конструкторів, що означає, що кожен клас Monobehavior несе відповідальність за створення власних залежностей, які не завжди легко надаються через внутрішній механізм введення залежності Unity (перетягуванням і назначенням полів в Інспекторі).

При розробці проекту в Unity, розробникам доводиться вирішувати низку проблем. Однією з ключових проблем є відсутність вбудованої масштабованості. Якщо проект розширюється і клас посилається на інший клас з фіксованою реалізацією, порушується принцип інверсії залежності [9].

Крім того, якщо класи не тільки використовують свої залежності, а й намагаються створити ці залежності (які, у свою чергу, можуть мати свої власні залежності), це суперечить принципу єдиної відповідальності [10].

Патерн «Service Locator» по суті є постійним кешем, який зберігає посилання на інстанції класу і надає їх на запит. Простіше кажучи, він делегує завдання створення залежності іншому класу.

Основна ідея цієї моделі полягає у створенні централізованого реєстру ініціалізованих залежностей. Залежності – це компоненти, які пропонують конкретні сервіси (послуги), які можна розкрити за допомогою інтерфейсів, що мають назву "сервіс контракти". Наприклад, якщо клієнту потрібно викликати певну послугу, для нього немає необхідності знати, як її локалізувати і ініціалізувати, достатньо лише виконати запит патерну «Service Locator», що виконає всю роботу за контрактом.

Отже, метою патерну є локалізація послуг для клієнта за підтримки центрального реєстру об'єктів, які надають конкретні послуги.

Нижче наведено UML-діаграму патерну «Service Locator» (рис. 1.12):

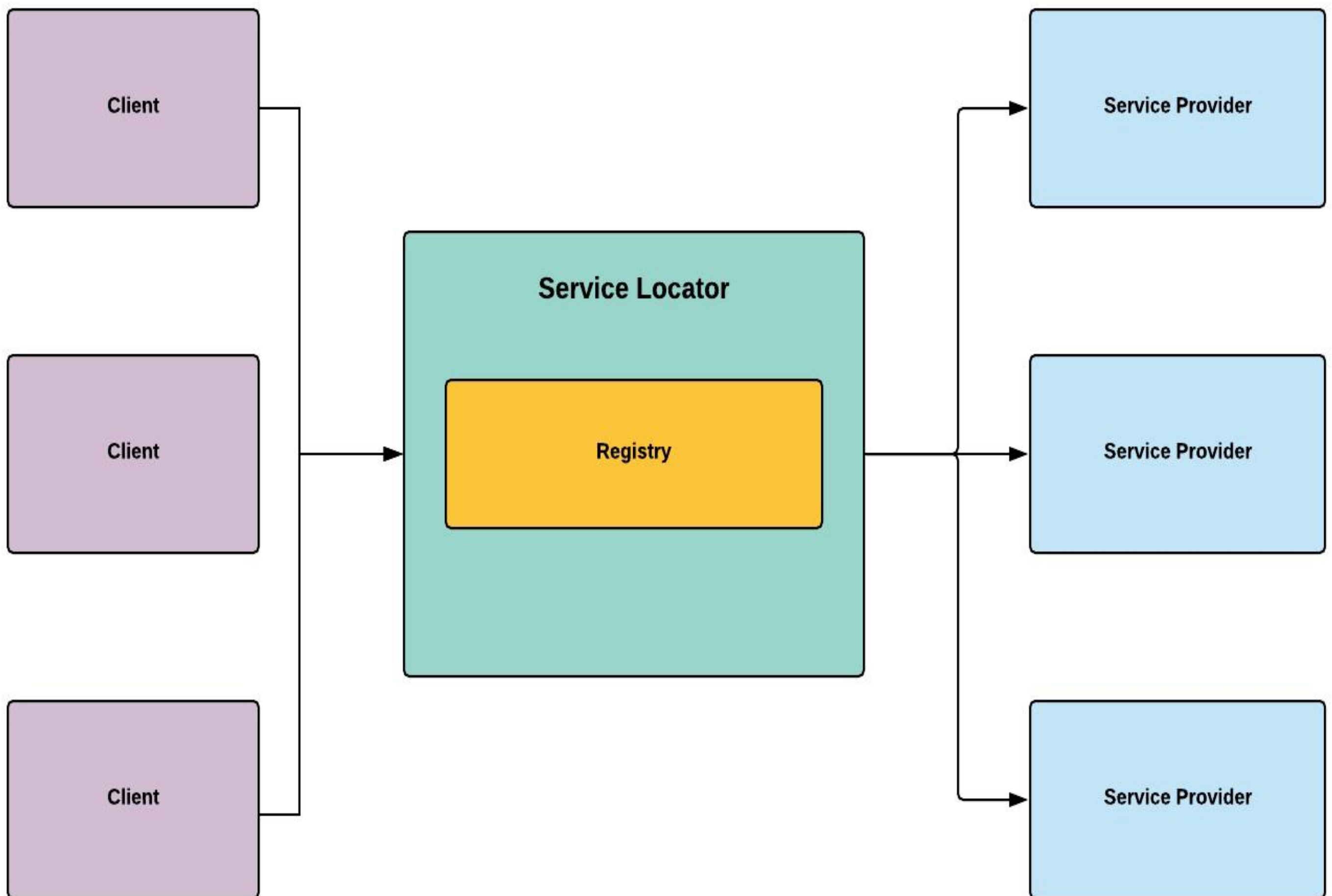


Рис 1.12. UML-діаграма патерну «Service Locator»

1. Service Locator — виступає посередником між клієнтом і реалізацією послуги. Він надає сервісні реалізації клієнту за запитом.

2. Client — користувач послуги, зазвичай клас, якому необхідний доступ до послуги.

3. Registry — зберігає посилання на сервіси, щоб запобігти створенню нових екземплярів сервісів.

4. Service — фактична реалізація сервісу, що дотримується сервісного інтерфейсу.

5. Service Scope (обсяг обслуговування) — опціональний компонент, який може бути надзвичайно корисним. Це клас, який об'єднує Service Locator і пропонує спосіб ініціалізації його екземпляру. Розширюючи цей клас, ми можемо створювати різні Service Locators для різних цілей, наприклад, один для релізу, а інший – юніт-тестування.

Модель «Service Locator» діє як проксі між клієнтами (Client) і постачальниками послуг (Service Provider) (рис. 1.12), і цей підхід роз'єднує їх до певної міри. Клієнту потрібно буде викликати шаблон «Service Locator» лише тоді, коли він має вирішувати залежність і потребує доступу до сервісу. Модель «Service Locator» має деяку аналогію з роботою офіціанта в ресторані: приймаючи замовлення від клієнтів і виступаючи посередником між різними послугами, які ресторан пропонує своїм клієнтам.

Потенційні переваги використання патерну «Service Locator»:

- Оптимізація часу запуску: модель може оптимізувати програму, динамічно виявляючи більш оптимізовані бібліотеки або компоненти для завершення роботи сервісу, в залежності від контексту часу виконання.
- Простота: модель є однєю з найпростіших серед моделей управління залежностями, які можна впровадити, і сумісна з використанням DI, що дає змогу швидко почати використовувати патерн у проекті.

Обмеження використання патерну «Service Locator»:

- Чорний бокс: реєстр шаблону «розмиває» залежності класу, тому під час його запуску можуть з'являтися деякі проблеми, пов’язані з відсутністю залежностей, а під час компіляції – ні, якщо залежності відсутні або неправильно зареєстровані.
- Глобально залежний: при надмірному використанні, модель може стати глобальною залежністю сама по собі, яка складно керується, що призведе до складностей із роз'єднанням основних компонентів.
- Патерн приховує залежності класів, тому немає іншого способу дізнатися, які залежності має клас, окрім як подивитись на його реалізацію. Саме тому «Service Locator» деякі розробники вважають антипатерном [11], подібно до «Singleton».

Для клієнтського класу, який використовує «Service Locator» для отримання залежностей, існування без «Service Locator» надалі стає неможливим, оскільки «Service Locator» сам по собі стає залежністю.

Саме тому, виникає необхідність у проектуванні системи, яка може легко налаштовуватись і автоматично обробляти такі інтерфейсні залежності. Тобто така система повинна вміти чітко визначити конкретні реалізації для існуючих залежностей та керувати ними.

Для вирішення даної проблеми може використовуватись Dependency Injection, що базується на принципах Inversion of Control (інверсії управління).

Inversion of Control (інверсія управління) – це абстрактний принцип, набір рекомендацій для написання слабо зв'язаного коду. Суть якого полягає в тому, що кожен компонент системи має бути якомога більш ізольованим від інших, не покладаючись у своїй роботі на деталі конкретної реалізації інших компонентів [12].

Висновки

Було проведено доменний аналіз розробки застосунків за допомогою ігрового рушія Unity 3D, розглянуто основні можливості та переваги даного рушія. Проведено огляд популярних патернів проектування, зокрема патернів «State», «Event Bus», «Command», «Object pool», «Observer», «Visitor», «Strategy», «Decorator», «Adapter», «Facade», розкрито їх основні переваги та обмеження в контексті можливості та доцільності застосування в процесі розробки ігор на Unity.

Особливу увагу в роботі було приділено патерну «Service Locator», який був обраний для подальшого дослідження з огляду на його високу гнучкість та здатність до інтеграції в різноманітні архітектурні структури. Аналіз показав, що цей патерн може бути ефективно використаний у Unity для реалізації гнучких та масштабованих рішень, що дозволяє вирішувати складні завдання управління залежностями та компонентами гри (застосунку).

Вибір ефективних патернів і підходів до проектування архітектури може суттєво вплинути на успішність розробки, забезпечуючи не лише гнучкість та масштабованість проекту, а й оптимізацію робочого процесу та підвищення якості кінцевого продукту.

РОЗДІЛ 2
МЕТОДИКА ПРОЕКТУВАННЯ СЕРВІС-ОРІЄНТОВАНИХ ЗАСТОСУНКІВ ІЗ ВИКОРИСТАННЯМ ПАТЕРНУ «SERVICE LOCATOR»

2.1 Теоретичне підґрунтя для розробки методики

Для розробки методики проектування в Unity на основі патерну Service Locator + DI (Dependency Injection) + Factory + State Machine, можна використати теорію категорій та теорію множин як теоретичне підґрунтя. Ось як це можна реалізувати:

1. Об'єкти та морфізми. В теорії категорій, системи та їх взаємодії можна моделювати за допомогою об'єктів (компонентів, сервісів) та морфізмів (функцій, які описують взаємодії між цими об'єктами). У контексті Unity, це може бути застосовано до визначення взаємозв'язків між різними компонентами гри, такими як Service Locator, DI, Factory, та State Machine.
2. Функтори та монади. Функтори (структури, що відображають об'єкти та морфізми однієї категорії в іншу) і монади (тип функторів, що дозволяють послідовно об'єднувати операції) можуть бути використані для моделювання та управління потоками даних та залежностей у системі.
3. Множини та підмножини. Елементи гри (об'єкти, компоненти) можуть бути представлені як множини, а їх властивості та характеристики – як підмножини. Це дозволяє чітко визначити та управляти залежностями та відносинами між компонентами.
4. Операції над множинами. Операції, такі як об'єднання, перетин, різниця множин, можуть бути використані для опису взаємодій між компонентами. Наприклад, об'єднання множин може представляти інтеграцію різних сервісів в одну систему.

Застосування в Unity:

- Service Locator може бути представлений як централізована система, що відповідає за відстеження та надання доступу до різних сервісів (об'єктів), використовуючи принципи теорії категорій для управління залежностями.
- Dependency Injection може бути інтерпретовано через теорію множин, де залежності між об'єктами представляються як відносини між множинами.
- Factory Pattern використовується для створення об'єктів, що можна розглядати через призму теорії категорій, де фабрика – це функтор, який відображає параметри на об'єкти.
- State Machine можна моделювати, використовуючи теорію множин, де стани представляються як множини, а переходи – як відносини між ними.

Комбінуючи перераховані вище теоретичні підходи, можна розробити гнучку та масштабовану систему для Unity, яка оптимізує управління залежностями та потоками даних в комплексних проектах.

2.1.1 Теорія категорій

Категорії – це деякі математичні структури, окремими випадками яких є частково впорядковані множини, а також напівгрупи з одиницею (у тому числі групи). У багатьох випадках категорію зручно представляти як "узагальнену частково впорядковану множину", в інших випадках як "загальну напівгрупу з одиницею".

Визначення 2.1. Категорія K задається наступним набором даних:

1. Сукупністю об'єктів, які позначаються заголовними латинськими літерами *A, B, C*...
2. Сукупністю морфізмів, або стрілок, які позначаються рядковими латинськими літерами *f, g, h*...

3. Операціями *dom* і *cod*, які зіставляють кожній стрілці *f* деякі об'єкти *dom(f)* і *cod(f)*, називаються початком і кінцем *f*. Той факт, що *dom(f) = A* і *cod(f) = B*, наочно зображується так:

$$\mathrm{f}: \mathrm{A} \rightarrow \mathrm{B} \text{ або } A \xrightarrow{f} B, \quad (2.1)$$

де *f* – стрілка (або морфізм) з *A* у *B*.

4. Операцією композиції, яка по кожній парі стрілок *f* і *g*, розташованих у вигляді:

$$A \xrightarrow{f} B \xrightarrow{g} \mathrm{C}, \quad (2.2)$$

тобто, *cod(f) = dom(g)*, видає певну стрілку:

$$A \xrightarrow{g \circ f} \mathrm{C}, \quad (2.3)$$

що називається композицією *g* і *f*.

5. Операцією *id*, яка по кожному об'єкту *A* видає одиничну стрілку об'єкта *A*, яку також називають ідентичністю обсягу *A*, що є тотожним або однаковим морфізмом обсягу *A*:

$$A \xrightarrow{id_A} A, \quad (2.4)$$

При цьому повинні бути виконані наступні умови:

1. Асоціативність композиції. Для будь-якої трійки стрілок *f*, *g*, *h*, розташованих наступним чином:

$$A \xrightarrow{f} B \xrightarrow{g} \mathrm{C} \xrightarrow{h} D, \quad (2.5)$$

виконується рівність:

$$(h \circ g) \circ f = h \circ (g \circ f). \quad (2.6)$$

2. Властивості тотожності:

- Для будь-якої $f: A \to B$, виконується рівність $f \circ id_A = f$.
- Для будь-якої $f: A \to B$, виконується рівність $id_B \circ f = f$.

Функтори – правильні відображення однієї категорії в іншу. Прикладами функторів є монотонні відображення впорядкованих множин, а також гомоморфізми напівгруп з одиницею (у тому числі гомоформні групи).

Визначення 2.2. Функтор F з категорії K_1 в категорію K_2 – це пара відображень (позначаються однією літерою), така, що:

$$F: Ob(K_1) \to Ob(K_2), \quad (2.7)$$

$$F: Mor(K_1) \to Mor(K_2), \quad (2.8)$$

де (2.7) відображає об'єкти K_1 в об'єктів K_2, а (2.8) – морфізми K_1 у морфізмах K_2, причому зберігаються *dom*, *cod*, *id* і композиція.

Тобто, що для будь-яких f, g, A, B, C з категорії K_1 мають виконуватися наступні умови:

1. $f: A \to B$ веде до $\mathrm{F}(f): F(A) \to F(B)$.

2. $F(id_A) = id_{F(A)}$.

3. $A \xrightarrow{f} B$ і $B \xrightarrow{g} C$ означає $\mathrm{F}(g \circ f) = F(g) \circ F(f)$.

2.2 Опис методики

Говорячи про будь-який застосунок (додаток для навчання, гра тощо), життєвий цикл сеансу виглядає приблизно таким чином (рис. 2.1):

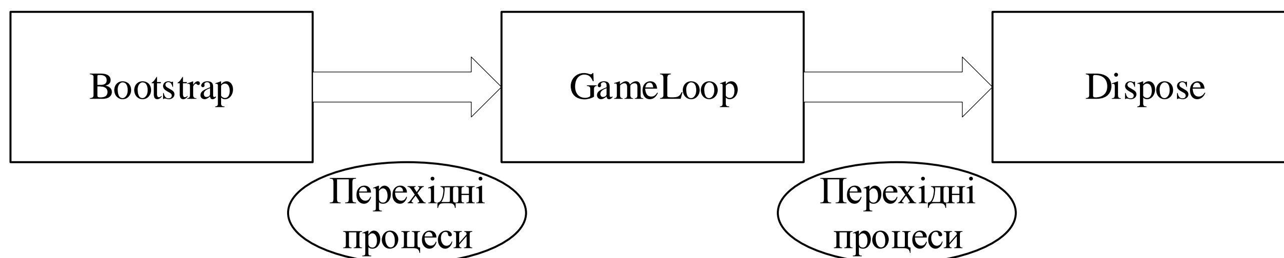


Рис. 2.1. Життєвий цикл сеансу взаємодії з додатком

Bootstrap – це вхідна точка, місце де ініціалізуються сервіси, залежності, завантажуються необхідні для запуску ресурси.

GameLoop – це безпосередньо ігровий процес, взаємодії з додатком, в якому відбувається вся ігрова або бізнес-логіка.

Dispose – це стан додатку перед виходом, вивантаження ресурсів, сервісів, тощо.

Найбільш критичні точки у цій схемі (рис. 2.1) знаходяться між станами життєвого циклу і можуть називатися «перехідними процесами». Під цим розуміється, наприклад, порядок ініціалізації сервісів, перевірки отримання залежностей, порядок вивантаження ресурсів з пам’яті. У разі залишення цих процесів без контролю, отримуємо невірне виконання програми або взагалі помилки, що блокують процес виконання.

Тому пропонується розробити методику, яка дозволить контролювати ці стани, мати зручний механізм переходу між ними, мати початкову точку входу, в якій контрольовано ініціалізуються необхідні залежності за допомогою сервісів. А доступ до самих сервісів реалізується за допомогою Dependency Injection.

2.2.1 Керування станами, в яких знаходиться застосунок

Стани та керування станами зазвичай реалізуються за допомогою патерну «Game State Machine».

«State Machine» – це математична модель, яка використовується для опису поведінки системи. Вона складається з набору станів, переходів та дій. Стан представляє конкретну поведінку або умову системи, у той час як перехід визначає рух з одного стану в інший. Дії, асоційовані зі станами або переходами,

представляють логіку, яку потрібно виконувати при вході, виході або під час перебування в стані [13].

На рисунку 2.2 зображено послідовність запуску програмного застосунку (зокрема, гри).

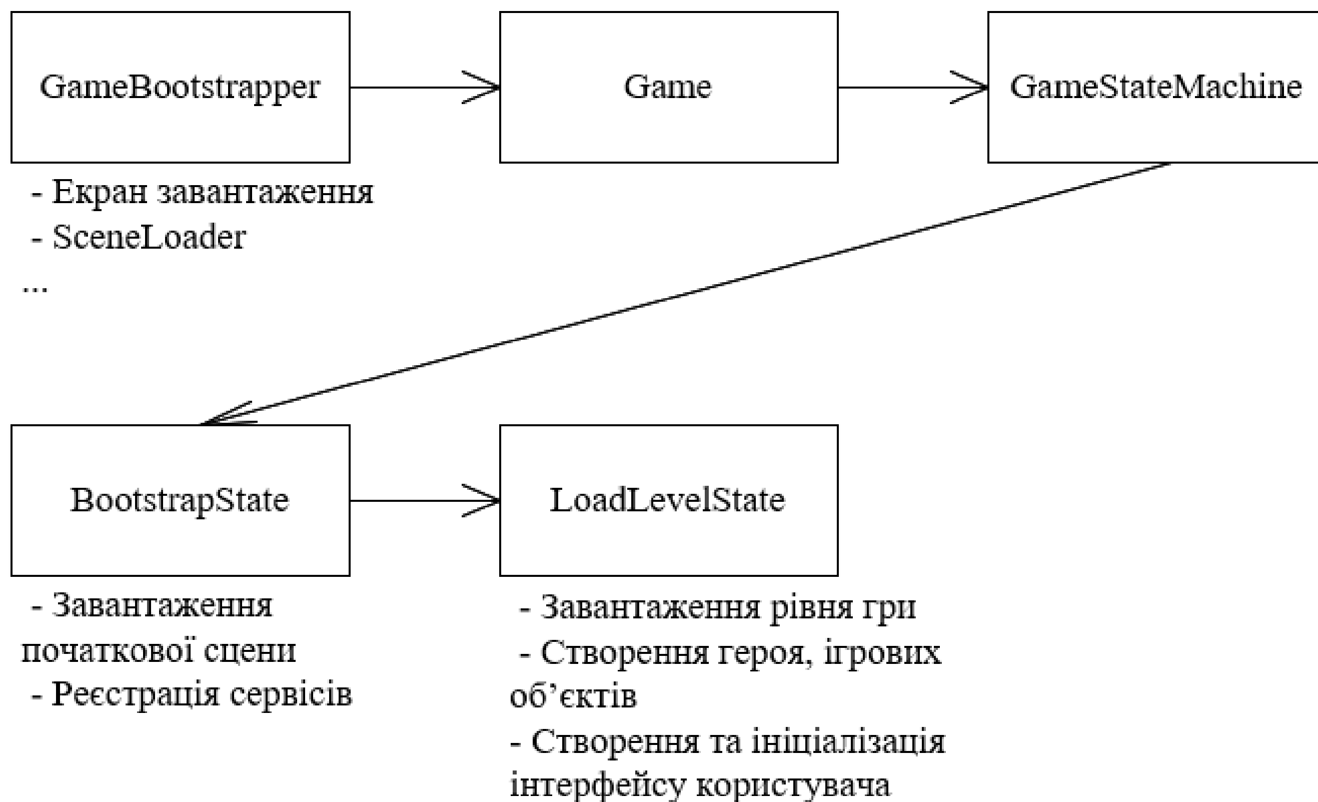


Рис. 2.2. Схема послідовності запуску застосунку

Перший крок у реалізації «State Machine» в Unity полягає у визначенні станів. Кожен стан буде представлений окремим скриптом.

Наступний крок – створення класу GameStateMachine для можливості керування переходами між станами та виконання відповідних дій.

Клас GameStateMachine може мати різні стани (\`BootstrapState\`, \`LoadLevelState\`, \`LoadProgressState\`, \`GameLoopState\`), які можна розглядати як об'єкти в категорії. Перехід між станами через методи Enter() та ChangeState() може бути аналогією морфізмів, які перетворюють один об'єкт (стан) у інший.

Одним з ключових аспектів теорії категорій є композиція морфізмів, що даному випадку може бути представлено як серія переходів між станами. Наприклад, перехід з \`BootstrapState\` до \`LoadLevelState\`, а потім до

`GameLoopState`. Композиція цих переходів (морфізмів) створює «шлях» крізь різні стани гри.

У теорії категорій функтори — це відображення між категоріями. Можна розглядати взаємодію між різними компонентами системи (наприклад, між `GameStateMachine` та конкретними станами) як функтор-подібні відносини, де поведінка одного компонента визначає поведінку іншого.

Система станів може бути розглянута як категорія, де стани є об'єктами, а переходи між ними — морфізмами. У рамках більшої системи (наприклад, всієї гри), така система станів може розглядатися як підкатегорія.

У теорії категорій кожен об'єкт має ідентичний морфізм, що відображає об'єкт сам у себе. Це може бути реалізовано як здатність стану залишатися незмінним, якщо не відбувається перехід до іншого стану.

2.2.2 Проектування структури сервіс-орієнтованого застосунку

Проектування структури сервіс-орієнтованого застосунку (SOA, Service-Oriented Architecture) в Unity вимагає розуміння як основних принципів SOA, так і специфіки Unity як ігрового рушія. Основна ідея SOA полягає у створенні модульних, незалежних сервісів, які можуть бути легко замінені, оновлені або модифіковані без впливу на інші частини системи [14].

SOA визначає спосіб зробити програмні компоненти придатними для багаторазового використання та взаємодії через сервісні інтерфейси. Сервіси використовують загальні стандарти інтерфейсу та архітектурний шаблон, щоб їх можна було швидко інтегрувати в нові додатки. Це знімає завдання з розробника програми, який раніше переробляв або дублював існуючу функціональність або повинен був знати, як з'єднати або забезпечити сумісність з існуючими функціями.

Кожна послуга в SOA містить код і дані, необхідні для виконання повної, дискретної бізнес-функції (наприклад, перевірка кредитоспроможності клієнта, розрахунок щомісячного платежу за кредитом або обробка заявки на іпотеку). Інтерфейси сервісів забезпечують вільний зв'язок, тобто їх можна викликати, майже

не знаючи, як реалізована послуга, що знаходиться під ними, що зменшує залежність між додатками.

Нижче наведено ключові аспекти для проектування SOA в Unity:

- Визначення сервісів. Функціональні елементи застосунку можуть бути відділені як самостійні сервіси. Це можуть бути, наприклад, система збереження гри, управління звуком, мережеві взаємодії, управління рекламою тощо.
- Інтерфейси сервісів. Необхідно чітко визначити інтерфейси для кожного сервісу. Це дозволить заміняти реалізації сервісів без необхідності зміни коду, який використовує ці сервіси.
- Залежності та ін'єкція залежностей. Використання шаблону Dependency Injection) допоможе управляти залежностями між різними сервісами та компонентами. Це може бути реалізовано через конструктори, сеттери або через спеціальні фреймворки (Zenject [15], VContainer [16]).
- Менеджер сервісів. Для керування сервісами необхідно створити центральний менеджер сервісів (Service Locator), який буде відповідати за ініціалізацію, зберігання та доступ до різних сервісів у застосунку.
- Модульність та гнучкість. Кожен сервіс необхідно розробити так, щоб він був самодостатнім і міг функціонувати незалежно від інших частин системи. Це забезпечить високу гнучкість та спростить тестування та розвиток проекту.
- Розгортання та оновлення сервісів. Способи розгортання та оновлення окремих сервісів, повинні мати змогу оновлювати сервіси без зупинки всієї системи.
- Тестування. Кожен сервіс проектується з урахуванням можливості його тестування. Автоматизоване тестування є ключовим елементом для підтримки високої якості коду та стабільності системи.

У Unity SOA може бути реалізовано як за допомогою вбудованих засобів (наприклад, через компоненти MonoBehaviour), так і за допомогою зовнішніх бібліотек або фреймворків. Головне — зберігати фокус на чіткому визначенні ролей та відповідальностей кожного сервісу, а також на гнучкості та розширюваності архітектури.

2.2.3 Роль і взаємодія сервісів та компонентів

Роль і взаємодія сервісів та компонентів у сервіс-орієнтованій архітектурі в Unity або в будь-якій іншій платформі є ключовими для створення добре структурованих, масштабованих і легко підтримуваних додатків. Ось декілька основних аспектів цієї взаємодії:

1. Роль сервісів. Кожен сервіс у SOA відповідає за конкретну функціональність або бізнес-логіку. Наприклад, у контексті ігрової розробки, можуть бути окремі сервіси для управління ігровими раундами, збереженням стану гри, взаємодією з мережею, тощо. Сервіси розробляються таким чином, щоб бути максимально незалежними один від одного, що дозволяє легко їх замінювати або модифікувати.
2. Роль компонентів. Компоненти часто відповідають за взаємодію з користувачем (UI, user interface), представлення даних і контроль над ігровим процесом. У Unity, це можуть бути різні MonoBehaviour скрипти, які приєднані до ігрових об'єктів. Компоненти можуть взаємодіяти з сервісами для отримання або зміни даних, виконання певних дій тощо.
3. Взаємодія між сервісами та компонентами. Компоненти можуть звертатися до сервісів з запитами на виконання певних дій або отримання даних. У свою чергу, сервіси обробляють ці запити та повертають результати. Сервіси можуть генерувати події, на які підписуються компоненти. Наприклад, сервіс управління ігровим раундом може генерувати подію про його завершення, на яку підписані UI-компоненти для відображення результатів раунду.
4. Залежності та ін'єкція залежностей. Замість того, щоб компоненти самостійно створювали екземпляри сервісів, вони часто отримують ці екземпляри через ін'єкцію залежностей (DI). Це знижує зв'язаність між компонентами і сервісами та полегшує тестування та підтримку коду.
5. Менеджери та фабрики. Часто використовуються спеціальні менеджери або фабрики для створення та управління життєвим циклом сервісів.

2.2.4 Розробка класу Service Locator для управління сервісами

Розробка Service Locator в Unity – це спосіб управління та використання сервісів у грі або додатку. Service Locator діє як централізований реєстр, де всі сервіси реєструються та можуть бути отримані іншими частинами програми. Це дозволяє компонентам отримувати доступ до сервісів без необхідності відслідковувати їх конкретні реалізації.

Нижче наведено базовий план для створення Service Locator:

1. Визначення інтерфейсів для кожного сервісу, який планується використовувати. Наприклад, `IAudioService`, `ISaveLoadService` тощо.
2. Розробка конкретні класів, які реалізують ці інтерфейси.
3. Розробка класу ServiceLocator, який буде зберігати екземпляри всіх сервісів. Типовий Service Locator може використовувати словник або іншу колекцію для зберігання сервісів.
4. Реєстрація сервісів: щоб мати змогу використовувати сервіси, їх необхідно зареєструвати. Наприклад, RegisterService<T>(T service).
5. Отримання сервісів – для цього необхідно реалізувати метод `GetService<T>()` для отримання сервісів.
6. Ініціалізація та використання: ініціалізація `ServiceLocator` та реєстрація всіх необхідних сервісів виконується на початку виконання програми (BootstrapState).

Для отримання доступу до сервісів у різних частинах програми потрібно використовувати клас ServiceLocator.

Нижче наведено приклад простого класу Service Locator (рис.2.3).

```
1:  public class ServiceLocator
2:  {
3:      private static readonly Dictionary<Type, object> _services =
    new Dictionary<Type, object>();
4:
5:      public static void RegisterService<T>(T service)
6:      {
7:          _services[typeof(T)] = service;
8:      }
9:
10:     public static T GetService<T>()
11:     {
12:         return (T)_services[typeof(T)];
13:     }
14: }
15:
16: // Приклад використання
17: public interface IAudioService
18: {
19:     void PlaySound(string soundName);
20: }
21:
22: public class AudioService : IAudioService
23: {
24:     public void PlaySound(string soundName)
25:     {
26:         // Логіка відтворення звуку
27:     }
28: }
29:
30: // Ініціалізація
31: ServiceLocator.RegisterService<IAudioService>(new AudioService());
32:
33: // Використання
34: IAudioService audioService =
    ServiceLocator.GetService<IAudioService>();
  audioService.PlaySound("soundName");
```

Рис. 2.3. Лістинг коду, що є прикладом простого класу Service Locator

Такий підхід дозволяє відокремити конкретні реалізації сервісів від їх використання, забезпечуючи легкість управління залежностями та гнучкість у заміні або оновленні сервісів. Однак, важливо використовувати Service Locator з обережністю, оскільки надмірне його використання може призвести до труднощів у відстеженні залежностей і ускладнити тестування.

2.2.5 Впровадження Dependency Injection контейнера для керування залежностями

Впровадження контейнера для Dependency Injection (DI) в Unity або в будь-якій іншій програмній платформі є важливим кроком для керування залежностями у вашому застосунку. DI допомагає знизити зв'язаність між різними частинами програми, полегшує управління змінами та спрощує тестування.

Основні кроки для впровадження DI контейнера:

1. Вибір або створення DI-контейнера. Можна обрати існуючий DI фреймворк, такий як Zenject для Unity, або створити свій власний.
2. Визначення інтерфейсів та реалізацій сервісів. Визначення інтерфейсів для всіх сервісів, які будуть використовуватися в застосунку. Реалізація інтерфейсів в конкретних класах.
3. Реєстрація сервісів у DI-контейнері. Реєстрція інтерфейсів та їх реалізації в DI-контейнері. Це може включати в себе визначення області видимості для кожного сервісу (наприклад, singleton, transient).
4. Ін'єкція залежностей. DI-контейнер використовується для передачі залежностей в компоненти або класи, які вимагають цих сервісів.
5. Ініціалізація контейнера. Ініціалізація DI-контейнера відбувається на початку виконання програми (BootstrapState).

Нижче наведено приклад використання Zenject, популярного DI-фреймворка для Unity (рис.2.4).

```
35: // Інтерфейс та його реалізація
36: public interface IAudioService
37: {
38:     void PlaySound(string soundName);
39: }
40:
41: public class AudioService : IAudioService
42: {
43:     public void PlaySound(string soundName)
44:     {
45:         // Логіка відтворення звуку
46:     }
47: }
48:
49: // Використання Zenject для реєстрації сервісів
50: public class MyInstaller : MonoInstaller
51: {
52:     public override void InstallBindings()
53:     {
54:
Container.Bind<IAudioService>().To<AudioService>().AsSingle();
55:     }
56: }
57:
58: // Клас, який використовує IAudioService
59: public class SomeClass
60: {
61:     private IAudioService _audioService;
62:
63:     [Inject]
64:     public void Construct(IAudioService audioService)
65:     {
66:         _audioService = audioService;
67:     }
68:
69:     public void DoSomething()
70:     {
71:         _audioService.PlaySound("soundName");
72:     }
}
```

Рис. 2.4. Лістинг коду, що є прикладом використання DI-фреймворка для Unity Zenject

В цьому прикладі Zenject використовується для реєстрації та впровадження залежностей, що дає можливість зосередитися на бізнес-логіці застосунку, не турбуючись про управління залежностями та створення об'єктів.

2.2.6 Використання патерну «Factory» для створення об'єктів, доповнюючи патерн «Service Locator»

Використання патерну «Factory» разом з патерном «ServiceLocator» у Unity дозволяє створювати об'єкти більш гнучко і організовано. Патерн «Factory» відповідає за створення об'єктів, а «ServiceLocator» дозволяє отримувати доступ до різних сервісів, необхідних для цього створення. Це знижує зв'язаність між різними частинами системи та полегшує розширення та тестування коду.

Щоб поєднати ці два патерни, необхідно виконати наступні дії:

1. Створення фабрики. Для цього визначається інтерфейс `IFactory`, який описує методи для створення об'єктів. Оскільки нам потрібно створювати об'єкти різних типів та призначення, фабрик може бути декілька.
2. Реєстрація фабрик у Service Locator як сервісів.
3. Використання фабрики через Service Locator. Отримання потрібної фабрики реалізується через Service Locator і використовується для створення об'єктів.

Нижче наведено приклад імплементації фабрики (рис.2.5).

```
73: // Інтерфейс та його реалізація для фабрики
74: public interface IEnemyFactory
75: {
76:     Enemy CreateEnemy(string type);
77: }
78:
79: public class EnemyFactory : IEnemyFactory
80: {
81:     public Enemy CreateEnemy(string type)
82:     {
83:         // Логіка створення ворога
84:         return new Enemy(type);
85:     }
86: }
87:
88: // Додавання фабрики до Service Locator
89: ServiceLocator.RegisterService<IEnemyFactory>(new EnemyFactory());
90:
91: // Використання фабрики через Service Locator
92: public class EnemySpawner
93: {
94:     private IEnemyFactory _enemyFactory;
95:
96:     public EnemySpawner()
97:     {
98:         _enemyFactory =
    ServiceLocator.GetService<IEnemyFactory>();
99:     }
100:
101:     public void SpawnEnemy(string type)
102:     {
103:         Enemy enemy = _enemyFactory.CreateEnemy(type);
104:         // Додаткова логіка
105:     }
106: }
```

Рис. 2.5. Лістинг коду імплементації фабрики

У прикладі (рис. 2.5) EnemyFactory відповідає за створення об'єктів ворогів, і його можна легко отримати та використовувати через ServiceLocator.

Висновки

У ході дослідження було проаналізовано ключові положення теорії категорій, яка стала теоретичним підґрунтям для розробки методики проектування застосунку.

Було запропоновано інтерпретацію життєвого циклу застосунку, як набору станів програми, які переходять з одного в інший за допомогою «State Machine». Композицію цих станів можливо представити як композицію морфізмів, відображення одного стану в інший, що випливає з теорії категорій.

Аналіз принципів Service Oriented Architecture показав, що основна ідея SOA полягає у створенні модульних, незалежних сервісів, які можуть бути легко замінені, оновлені або модифіковані без впливу на інші частини систем. Тому було сформульовано ключові вимоги до проектування та створення сервісів.

Для керування сервісами було обрано патерн «Service Locator» та наведено приклад його реалізації.

Щоб мінімізувати недоліки патерну «Service Locator» було використано Dependency Injection для керуванням залежностями та патерн «Factory» з метою контролю над життєвим циклом об’єктів.

Розроблена методика проектування застосунків демонструє ефективність використання патерну «Service Locator» для створення гнучких та масштабованих сервіс-орієнтованих застосунків.

Представлені рекомендації щодо проектування структури застосунку, взаємодії сервісів та компонентів, та реалізації Service Locator забезпечують підвищення ефективності розробки.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ СЕРВІС-ОРІЄНТОВАНОГО ЗАСТОСУНКУ В UNITY 3D

3.1. Створення ігрового застосунку для тестування запропонованої методики

Entry point (точка входу) у контексті розробки застосунків або ігор — це місце, де програма або гра починає своє виконання. Тобто, це початкова точка, з якої починається виконання коду.

Entry point дозволяє розробникам визначити, що саме повинно статися, коли програма або гра запускається. Сюди може входити ініціалізація ресурсів, налаштування з'єднань з базами даних, конфігурація сервісів тощо. Крім цього, тут визначається, як буде розгортатися потік виконання програми. Він визначає, які компоненти будуть завантажені, в якому порядку, та як будуть оброблятися цикли обробки подій.

Entry point є місцем, де розробники можуть швидко знайти початок логіки програми, що є важливим для розуміння та підтримки коду. Також, може використовуватися для налаштування середовища виконання, наприклад, визначення режиму відлагодження чи виробничого режиму.

Для складних додатків з багатьма залежностями, правильна послідовність ініціалізації компонентів є критично важливою. Entry point дозволяє управляти цією послідовністю.

В експериментальному додатку вхідною точкою є GameBootstrapper, який запускає гру та переводить GameStateMachine у початковий стан BootstrapState. На рисунку 3.1. проілюстрована UML діаграма класу GameBootstrapper.

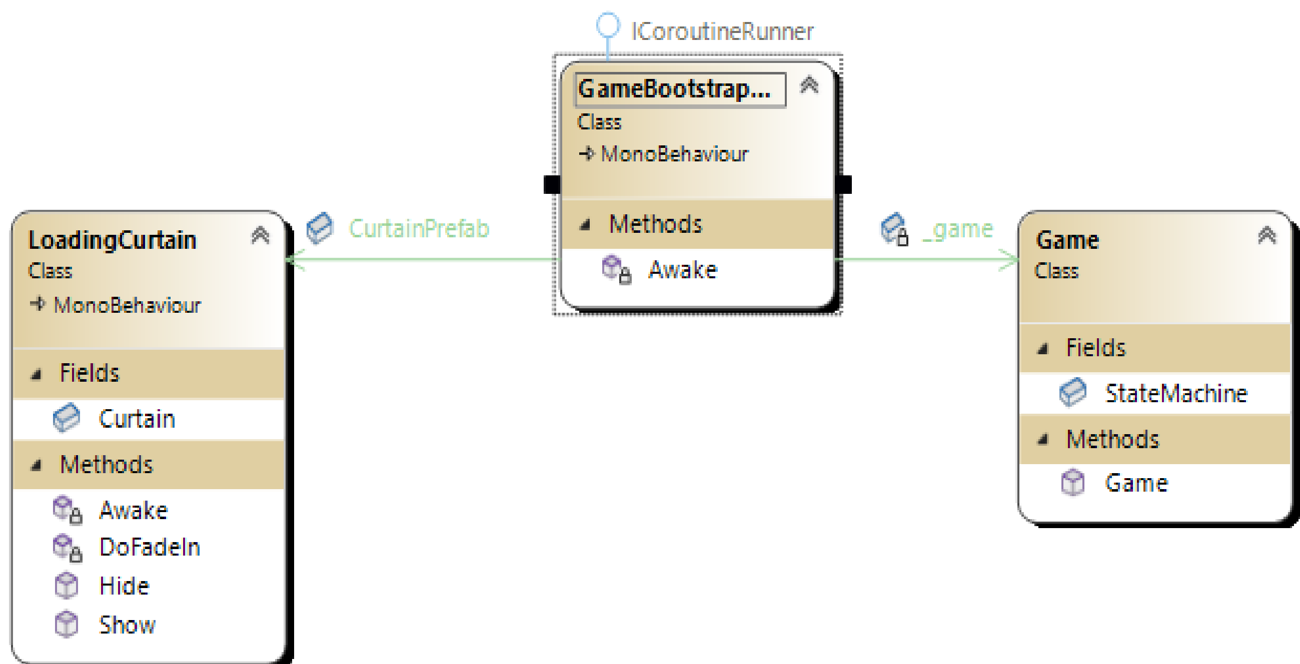


Рис 3.1. UML-діаграма класу GameBootstrapper

У BootstrapState (рис. 3.2) відбувається завантаження початкової сцени та реєстрація сервісів. Під час реєстрації кожному інтерфейсу сервісу закріпляється його реалізація, що додає додаткової гнучкості, оскільки далі під час виклику сервісу за контрактом ми отримуємо реалізацію, яку зареєстрували, не створюючи при цьому додаткових екземплярів сервісу.

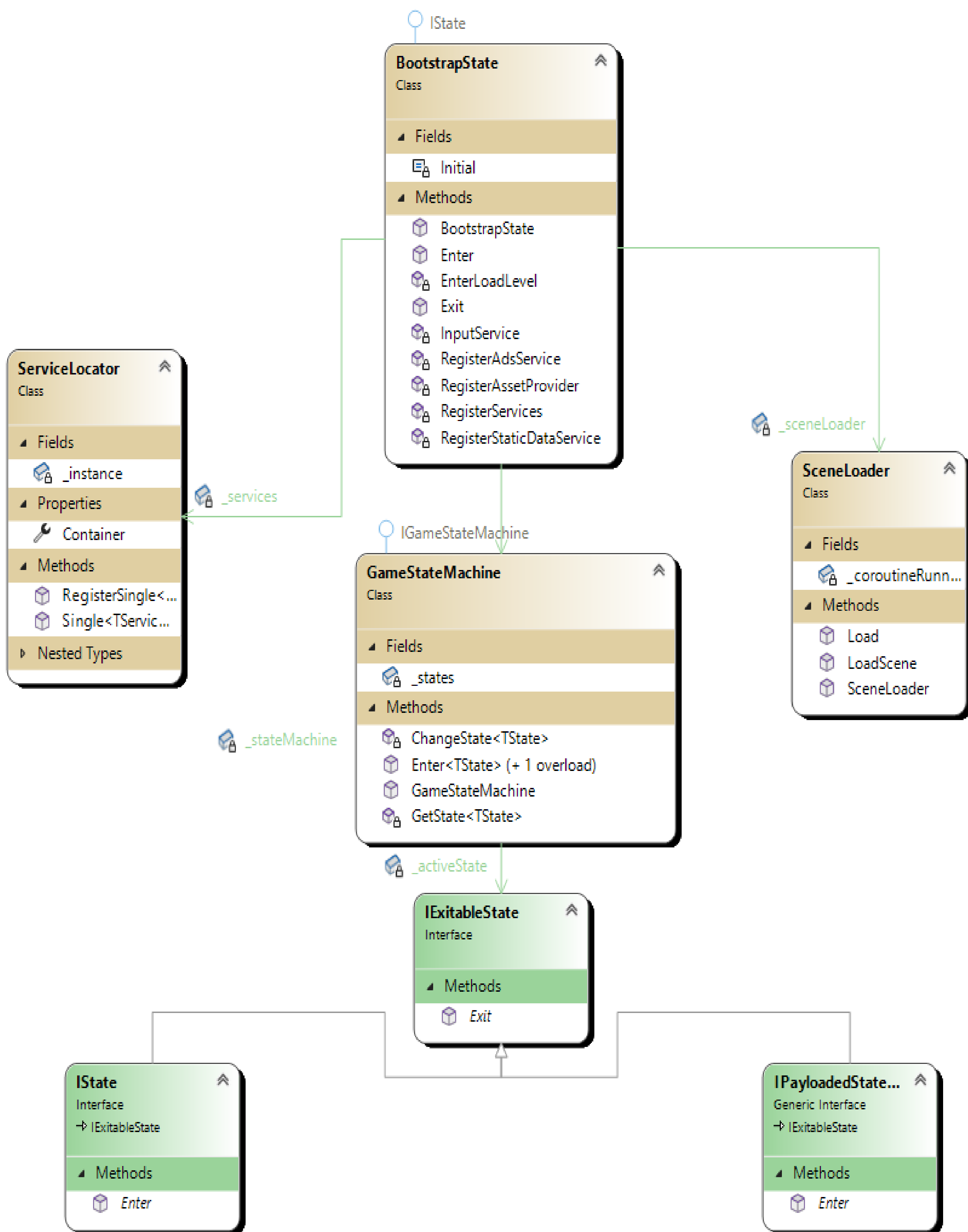


Рис 3.2. UML-діаграма класу BootstrapState

LoadLevelState (рис. 3.3) використовуючи сервіси ініціалізує головну сцену (рівень). За допомогою сервісу GameFactory відбувається створення персонажів, ворогів, heads up display (HUD), тригерів переходу на інший рівень. Реєструються об’єкти, що будуть зчитувати з прогресу або записувати в прогрес дані, які потрібно зберегти до прогресу.

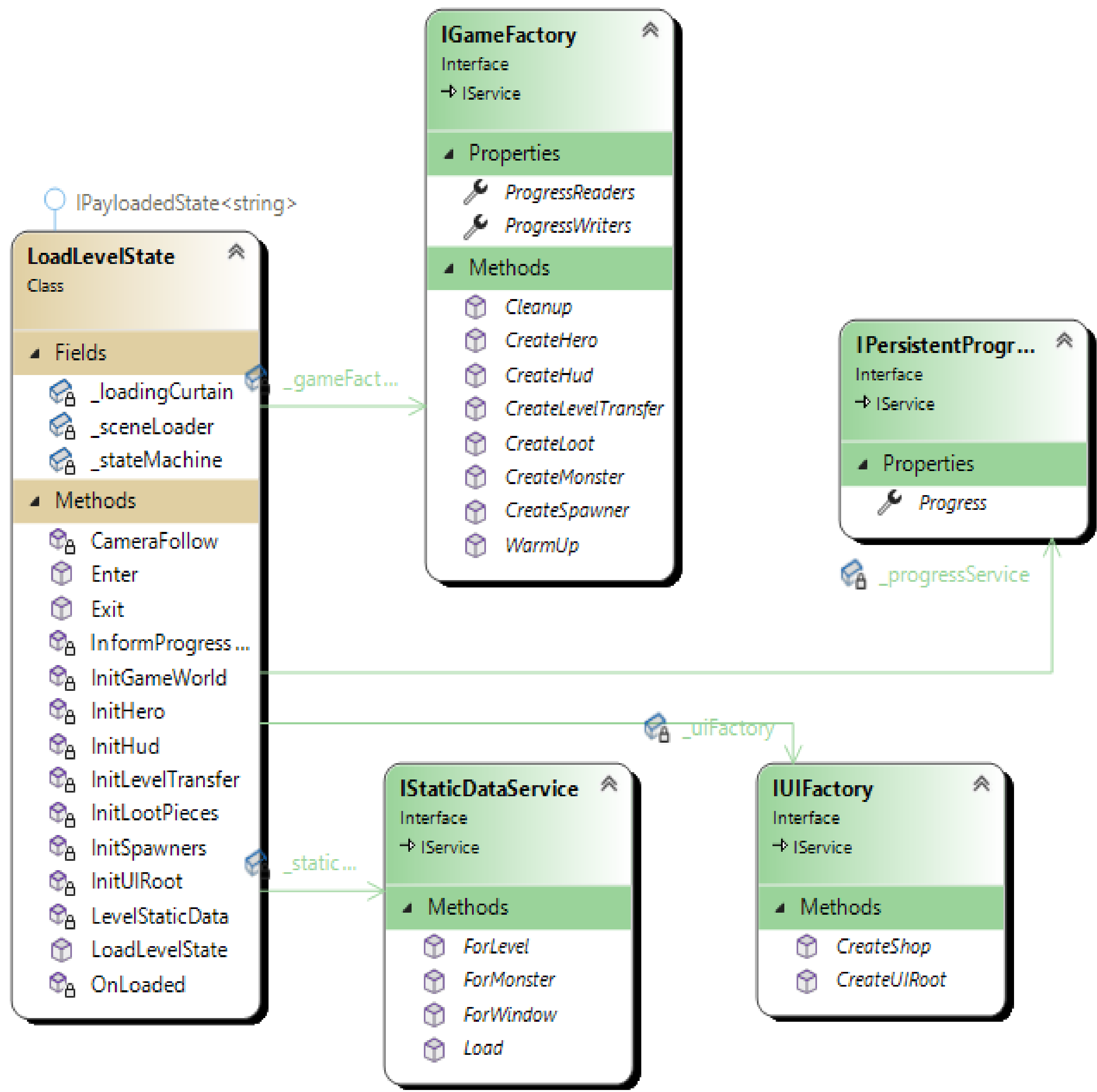


Рис 3.3. UML-діаграма класу LoadLevelState

Після ініціалізації всього необхідного StateMachine переходить до LoadProgressState (рис. 3.4). На цьому етапі завантажуються та застосовуються збережені дані про користувача, його позиція, кількість монет, а також статичні дані (баланс гри), тобто ті, що були створені ігровим дизайнером.

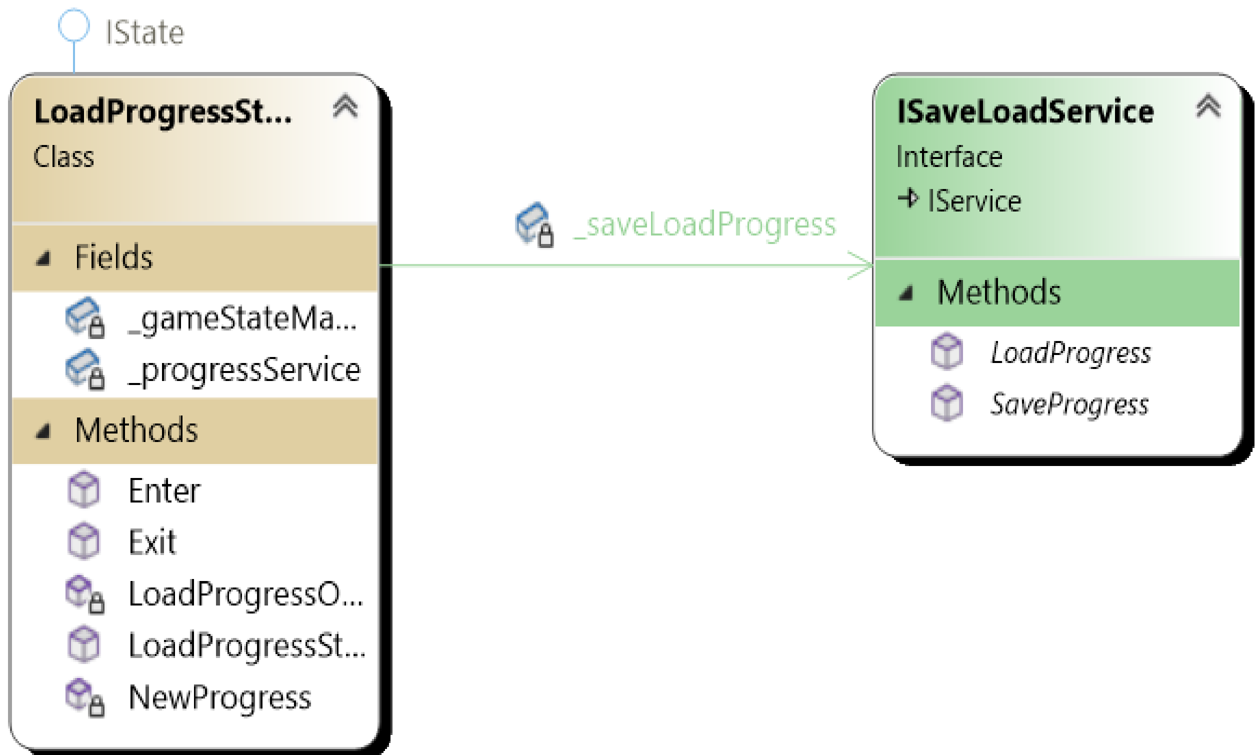


Рис 3.4. UML-діаграма класу LoadProgressState

Сама GameStateMachine реалізована наступним чином (рис. 3.5). В конструкторі ініціалізуються всі можливі стани за певним порядком, визначені методи для входу в новий стан а також метод зміни стану.

```
1:   public class GameStateMachine : IGameStateMachine
2:   {
3:     private Dictionary<Type, IExitableState> _states;
4:     private IExitableState _activeState;
5:
6:     public GameStateMachine(SceneLoader sceneLoader, LoadingCurtain
   loadingCurtain, ServiceLocator services)
7:     {
8:       _states = new Dictionary<Type, IExitableState>
9:       {
10:        [typeof(BootstrapState)] = new BootstrapState(this, sceneLoader,
   services),
11:        [typeof(LoadLevelState)] = new LoadLevelState(this, sceneLoader,
   loadingCurtain, services.Single<IGameFactory>(),
12:          services.Single<IPersistentProgressService>(),
   services.Single<IStaticDataService>(), services.Single<IUIFactory>()),
13:
14:        [typeof(LoadProgressState)] = new LoadProgressState(this,
   services.Single<IPersistentProgressService>(),
   services.Single<ISaveLoadService>()),
15:        [typeof(GameLoopState)] = new GameLoopState(this),
16:      };
17:    }
18:
19:    public void Enter<TState>() where TState : class, IState
20:    {
21:      IState state = ChangeState<TState>();
22:      state.Enter();
23:    }
24:
25:    public void Enter<TState, TPayload>(TPayload payload) where TState :
   class, IPayloadedState<TPayload>
26:    {
27:      TState state = ChangeState<TState>();
28:      state.Enter(payload);
29:    }
30:
31:    private TState ChangeState<TState>() where TState : class,
   IExitableState
32:    {
33:      _activeState?.Exit();
34:
35:      TState state = GetState<TState>();
36:      _activeState = state;
37:
38:      return state;
39:    }
40:
41:    private TState GetState<TState>() where TState : class,
   IExitableState =>
42:      _states[typeof(TState)] as TState;
43:  }
```

Рис. 3.5. Лістинг коду реалізації GameStateMachine

3.2. Реалізація класу Service Locator

Задля зручності ServiceLocator реалізовано через «Singleton», що дає змогу отримати екземпляр даного класу з будь-якого місця. Але варто мати на увазі, що екземпляр класу повинен бути лише один, щоб виключити дублювання виконання методів іншими екземплярами.

Нижче наведено лістинг коду реалізації класу ServiceLocator на прикладі експериментального застосунку (рис. 3.6).

```
107: public class ServiceLocator
108:   {
109:     private static ServiceLocator _instance;
110:     public static ServiceLocator Container => _instance ??
     (_instance = new ServiceLocator());
111:
112:     public void RegisterSingle<TService>(TService implementation)
     where TService : IService =>
113:       Implementation<TService>.ServiceInstance = implementation;
114:
115:     public TService Single<TService>() where TService : IService
     =>
116:       Implementation<TService>.ServiceInstance;
117:
118:     private class Implementation<TService> where TService :
     IService
119:     {
120:       public static TService ServiceInstance;
121:     }
122:   }
```

Рис. 3.6. Лістинг коду реалізації класу ServiceLocator

Клас ServiceLocator є центральним елементом паттерну. Його основна функція зберігати, інстанції сервісів.

Метод RegisterSingle<TService>() дозволяє реєструвати екземпляр сервісу в ServiceLocator. TService є загальним типом, що має реалізувати інтерфейс IService. Це дає можливість реєстрації різних типів сервісів.

Метод Single<TService>() повертає екземпляр зареєстрованого сервісу типу TService. Якщо такий сервіс був зареєстрований, він буде повернутий; інакше, результат буде `null` або викличе помилку залежно від реалізації.

Внутрішній клас Implementation<TService> використовується для зберігання інстанції сервісу. Це дозволяє уникнути прямого зберігання інстанцій у ServiceLocator, і замість цього вони зберігаються у статичній властивості ServiceInstance внутрішнього класу.

Даний код є прикладом реалізації паттерну «Service Locator», який часто використовується для управління залежностями у програмному забезпеченні. Його основна перевага полягає у здатності централізовано управляти сервісами та легкості доступу до них у різних частинах застосунку.

3.3. Розробка сервісів та їх взаємодія

Розглянемо сервіс для збереження поточного сеансу гри.

У першу чергу необхідно створити інтерфейс (рис. 3.7), що буде включати два методи: для збереження прогресу та для завантаження останнього збереженого прогресу.

Наведений вище код (рис. 3.7) реалізує сервіс збереження та завантаження прогресу в грі. Це досить типова задача для ігор, де необхідно зберегти стан гри між сеансами. Розглянемо його реалізацію більш детально.

Інтерфейс ISaveLoadService має два методи – SaveProgress() для збереження прогресу гри та LoadProgress() для завантаження останнього збереженого стану. Це дозволяє створити гнучку систему, яка може бути легко інтегрована з різними компонентами гри.

```
123: public interface ISaveLoadService : IService
124:   {
125:     void SaveProgress();
126:     PlayerProgress LoadProgress();
127:   }
128: public class SaveLoadService : ISaveLoadService
129:   {
130:     private const string ProgressKey = "Progress";
131:
132:     private readonly IPersistentProgressService _progressService;
133:     private readonly IGameFactory _gameFactory;
134:
135:     public SaveLoadService(IPersistentProgressService
     progressService, IGameFactory gameFactory)
136:     {
137:       _progressService = progressService;
138:       _gameFactory = gameFactory;
139:     }
140:
141:     public void SaveProgress()
142:     {
143:       foreach (ISavedProgress progressWriter in
     _gameFactory.ProgressWriters)
144:         progressWriter.UpdateProgress(_progressService.Progress);
145:
146:       PlayerPrefs.SetString(ProgressKey,
     _progressService.Progress.ToJson());
147:     }
148:
149:     public PlayerProgress LoadProgress()
150:     {
151:       return PlayerPrefs.GetString(ProgressKey)?
152:         .ToDeserialized<PlayerProgress>();
153:     }
154:   }
```

Рис. 3.7. Лістинг коду реалізації сервісу збереження прогресу гри

Клас SaveLoadService використовує дві залежності – IPersistentProgressService та IGameFactory. Це забезпечує розділення відповідальностей та гнучкість у розширенні.

Метод SaveProgress() проходить через всі об'єкти, які імплементують ISavedProgress, викликаючи метод UpdateProgress() для кожного з них, щоб оновити стан прогресу. Після цього зберігає прогрес у PlayerPrefs у форматі JSON.

Метод LoadProgress() завантажує збережені дані з PlayerPrefs та десеріалізує їх у об'єкт PlayerProgress.

PlayerPrefs є простим способом збереження даних між сеансами у Unity. Однак, варто зазначити, що він не є найбезпечнішим чи найефективнішим способом для збереження складних або об'ємних даних.

Код також передбачає, що клас PlayerProgress може бути серіалізований та десеріалізований у JSON. Це зручний спосіб зберігання структурованих даних, але вимагає, щоб клас PlayerProgress був правильно налаштований для цього.

Використання IGameFactory для отримання ProgressWriters додає рівень абстракції та гнучкості у систему, дозволяючи легко інтегрувати різні компоненти, які можуть вносити зміни у прогрес гри.

У даному сервісі відсутні жорсткі зв’зки з іншими компонентами, що дозволяє перевикористовувати даний функціонал.

Інші сервіси проектуються за схожим принципом.

Висновки

Було реалізовану запропоновану методику на експериментальному проекті. Запропонована архітектура дозволила чітко спроектувати життєвий цикл застосунку та мати контроль над порядком ініціалізації об’єктів. Розглядаючи новий функціонал як новий сервіс можливо масштабувати проект скільки потрібно, єдине що потрібно реєструвати сервіси у BootstrapState

Реалізація патерну «Service Locator» в Unity3D показала його практичну придатність та ефективність у контексті ігрового розроблення.

Аналіз використання патерну в розробленому застосунку підкреслює його важливість для забезпечення гнучкості та масштабованості проектів.

ВИСНОВКИ

Аналіз предметної області проектування застосунків за допомогою ігрового рушія Unity показав, що під час розробки та проектування виникають різні фактори, які в подальшому впливають на якість та масштабованість майбутнього проекту. Вибір підходу до проектування вносить свої корективи та впливає на час та якість продукту. А, отже, розробники постійно знаходяться у пошуку кращого підходу, комбінуючи напрацювання попередніх років.

Проведено огляд популярних патернів проектування, зокрема патернів «State», «Event Bus», «Command», «Object pool», «Observer», «Visitor», «Strategy», «Decorator», «Adapter», «Facade», розкрито їх основні переваги та обмеження в контексті можливості та доцільності застосування в процесі розробки ігор на Unity.

Проаналізовано ключові положення теорії категорій, що є теоретичним підґрунтям для розробки методики проектування майбутнього застосунку. Було запропоновано інтерпретацію життєвого циклу застосунку, як набору станів програми, що переходять з одного в інший за допомогою «State Machine».

Розглянуто принципи Service Oriented Architecture, що дало змогу сформулювати ключові вимоги до проектування та створення сервісів у рамках розроблюваної методики.

Вибір ефективних патернів і підходів до проектування архітектури може суттєво вплинути на успішність розробки, забезпечуючи не лише гнучкість та масштабованість проекту, а й оптимізацію робочого процесу та підвищення якості кінцевого продукту. З огляду на високу гнучкість та здатність до інтеграції в різноманітні архітектурні структури було обрано патерн патерну «Service Locator». Крім того, для реалізації застосунку було використано Dependency Injection для керуванням залежностями та патерн «Factory» з метою контролю над життєвим циклом об’єктів.

Розроблена методика проектування застосунків демонструє ефективність використання патерну «Service Locator» для створення гнучких та масштабованих сервіс-орієнтованих застосунків. Представлені рекомендації щодо проектування

структури застосунку, взаємодії сервісів та компонентів, та реалізації Service Locator забезпечують підвищення ефективності розробки.

Отже, результати дослідження показали, що використання розробленої в роботі методики може значно підвищити ефективність процесу розробки програмного забезпечення, що дає змогу краще адаптувати та масштабувати проекти, та може бути корисним як для розробників-початківців, а також досвідчених фахівців у сфері розробки ігор та програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 3D Application Development Using Unity Real Time Platform [Електронний ресурс] / K.Kaushal, Rupa Rani, Atul Kumar Rai, Varsha Sharma. – 2023. – Режим доступу до ресурсу: https://www.researchgate.net/publication/373987608_3D_Application_Development_Using_Unity_Real_Time_Platform.
2. Pascal Mosler. Using the Game Engine Unity Efficiently in Teaching - Development of a fully-automated webserver-based build pipeline [Електронний ресурс] / Pascal Mosler // Conference: Proceedings of the 41st Conference on Education and Research in Computer Aided Architectural Design in Europe (eCAADe 2023)At: Graz, Austria. – 2023. – Режим доступу до ресурсу: https://www.researchgate.net/publication/374256030_Using_the_Game_Engine_Unity_Efficiently_in_Teaching_-_Development_of_a_fully-automated_webserver-based_build_pipeline.
3. Mobile Game Development Using Unity Engine [Електронний ресурс] / Fatima Sapundzhi1, Anton Kitanov, Meglena Lazarova, Slavi Georgiev // Methodologies and Intelligent Systems for Technology Enhanced Learning, Workshops - 13th International Conference. – 2023. – Режим доступу до ресурсу: https://www.researchgate.net/publication/373475694_Mobile_Game_Development_Using_Unity_Engine.
4. Game Engines & their use in game development [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://www.slashdata.co/blog/did-you-know-that-60-of-game-developers-use-game-engines.
5. Mastering Unity A Beginner’s Guide – Boca Raton, Florida: CRC Press, 2022. – 280 с.
6. Patrick Felicia. Unity 5 From Zero to Proficiency (Foundations) / Patrick Felicia., 2015. – 162 с.

7. Made With Unity [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/made-with-unity.8. Game Development Patterns with Unity 2021 *Second Edition*

8. Bipin Joshi. Beginning SOLID Principles and Design Patterns for ASP.NET Developers / Bipin Joshi. – Thane, India, 2016. – 415 с.

9. Taha Mert Gokdemir. Design Patterns for Unity Developers: Service Locator [Електронний ресурс] / Taha Mert Gokdemir // Medium. – 2023. – Режим доступу до ресурсу: https://medium.com/@taha.m.gokdemir/design-patterns-for-unity-developers-service-locator-124cd4628c43#:~:text=The%20Service%20Locator%20is%20essentially,creating%20dependencies%20to%20another%20class.

10. Is ServiceLocator an anti-pattern? [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://medium.com/@iamprovidence/is-servicelocator-antipattern-7c54d6c1a3a1

11. Jahid Momin. Inversion of Control (IoC) Design principle [Електронний ресурс] / Jahid Momin // LinkedIn. – 2023. – Режим доступу до ресурсу: https://www.linkedin.com/pulse/inversion-control-ioc-design-principle-jahid-momin/.

12. Han Acıpayam M. Introduction to the Unity State Machine Pattern [Електронний ресурс] / Murat Han Acıpayam. – 2018. – Режим доступу до ресурсу: https://mracipayam.medium.com/introduction-to-the-unity-state-machine-pattern-ad3bce7d987c#:~:text=The%20Unity%20State%20Machine%20pattern%20is%20a%20powerful%20tool%20for,easily%20extend%20it%20as%20needed.

13. What is service-oriented architecture (SOA)? [Електронний ресурс] // IBM – Режим доступу до ресурсу: https://www.ibm.com/topics/soa.

14. Modesttree / Zenject [Електронний ресурс] // 2021 – Режим доступу до ресурсу: https://github.com/modesttree/Zenject.

15. VContainer [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://vcontainer.hadashikick.jp/.

ДОДАТОК А
АКТ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

ТОВАРИСТВО З ОБМЕЖЕНОЮ ВІДПОВІДАЛЬНІСТЮ
"ІНТЕКРОСС УКРАЇНА"

ідентифікаційний код 40618145

АКТ
про впровадження результатів магістерської роботи
Бездітного В'ячеслава Миколайовича

Цим актом підтверджується використання результатів кваліфікаційної роботи на проекті LeZoo.

Запропонована у роботі методика проектування застосунку на основі патерну Service Locator дозволяє створювати гнучку архітектуру, яку зручно масштабувати. А розбиття життєвого циклу застосунку на стани та використання State Machine для їх керування зменшує вірогідність помилкового відпрацювання програми, шляхом конролю над зміною програмою стану. Щоправда, для успішної інтеграції цієї архітектури у процес розробки необхідно встановити жорсткі вимоги до написання нового функціоналу для всієї команди розробників.

13.12.2023

місто Київ

Директор **С.П. Антонюк**

Україна, 04053, місто Київ, провулок Киянівський, 7-А
Телефон: +380 96 846 73 43
E-mail: finance@intecross.com