УДК 004.02 DOI: 10.31673/2412-4338.2020.017987

Дібрівний О.А., Гребенюк В.В., Кравчук П.О., Сеньков О.В. *Державний університет телекомунікацій, Київ*
Кільменінов О.А. *Національний університет оборони України імені Івана Черняховського, Київ*

ВИКОРИСТАННЯ ПРИНЦИПІВ SOLID ПРИ РОЗРОБЦІ ВІДЕО ІГОР НА ОСНОВІ ІГРОВОГО ДВИГУНА UNITY

На сьогоднішній день світова індустрія відео ігор знаходиться з в своєму розквіті. З відкриттям різноманітних платформ для реалізації відео ігор, таких як Steam, Google Play, Apple Store а також безкоштовних ігрових двигунів таких як Unity та Unreal Engine поріг входу в гейм-дев для початківців та студентів помітно знизився, особливо якщо говорити про ринок мобільних ігор. З мінімальними витратами часу та зусиль, використовуючи всі потужності вищезгаданих ігрових двигунів, процес розробки повноцінної відео гри може бути зведеним до кількох місяців, а то й тижнів. Проте часто розробники початківці, а часто й програмісти з досвідом, в погоні за швидкою розробкою помітно ускладнюють собі процес подальшої підтримки та розвитку своїх ігрових проектів. Одним із способів уникнення такого розвитку подій є використання принципів SOLID під час проектування ігрового додатку.

В даній статті основну увагу приділяється огляду архітектури програмних продуктів на основі ігрового двигуна Unity та можливостям впровадження принципів SOLID для оптимізації архітектури, та полегшенню подальшої підтримки та модифікації таких додатків. Розглянуто , та відповідність основних компонент ігрового двигуна UNITY принципу єдиного обов'язку а також особливості побудови класів при керуванні даним принципом. Визначено основні особливості використання принципу відкритосіті/закритості та його вплив на процес подальшої модернізації програмних архітектур. Запропоновано архітектурні рішення для використання принципу підстановки Ліскова та принципу розділення інтерфейсу. Оглянуто основні вимоги до абстарцій при роботі з принципом інверсії залежностей.

Запропоновано загальний скелет для побудови ігрових архітектур з врахуванням програмного принципу SOLID на основі ігрового додатку в жанрі rpg, з реалізацією окремих ігрових механік. Алгоритмізовано побудову ігрової архітектури на основі принципу SOLID, що призведе до спрощення процесу модернізації та підтримки ігрових додатків розроблених на основі ігрового двигуна UNITY.

Ключові слова: *ООП C#, ООП Java, SOLID., UNITY, розробка ігор, принципи програмування, патерни програмування.*

Dibrivniy O.A., Grebenyuk V.V., Kravchuk P.O., Senkov O.V.
State University of Telecommunications, Kyiv
Kilmeninov O.A. *The National Defense University of Ukraine named Ivan Chernyakhovsky, Kyiv*

SOLID PRINCIPIES USAGE IN VIDEO GAME DEVELOPMENT BASED ON UNITY GAME ENGINE

Today, the global video gaming industry is in its prime. With the launch of a variety of video game platforms such as Steam, Google Play, Apple Store, and free game engines such as Unity and Unreal Engine, the threshold for gaming devs for beginners and students has dropped significantly, especially when it comes to the mobile gaming market. With minimal time and effort, using all the power of the above mentioned game engines, the process of developing a full-fledged video game can take up to several months or even weeks. However, novice developers, and often programmers with

© Дібрівний О.А., Гребенюк В.В., Кравчук П.О., Кільменінов О.А., Сеньков О.В. 2020

experience, in the pursuit of rapid development significantly complicate the process of further support and development of their game projects. One way of avoiding this development is to use the principles of SOLID when designing a game application.

This article focuses on reviewing the software architecture of the Unity game engine and the ability to implement SOLID principles to optimize the architecture, and to facilitate the continued support and modification of such applications. The correspondence of the main components of the UNITY game engine with the Single Responsibility Principle is considered, as well as the peculiarities of class building when managing this principle. The main features of the usage of the open/closed principle and its influence on the process of further modernization of software architectures are identified. Architectural solutions for the use of the Liskov substitution principle and the Interface Segregation Principle are proposed. The basic requirements for abstractions when working with the principle of dependency inversion are examined.

A general skeleton for the construction of game architectures is proposed, taking into account the SOLID program principle based on a game application in the rpg genre, with the implementation of individual game mechanics. Also was suggested the game architecture, based on the SOLID principle, which will simplify the process of upgrading and supporting game applications developed on the basis of the UNITY game engine.

***Keywords**: OOP C#, OOP java, SOLID, UNITY, game development, programing principles, programing patterns.*

Дибривный О.А., Гребенюк В.В., Кравчук П.А., Сеньков О.В.

Государственный университет телекоммуникаций, Киев

Кильменинов А.А. *Национальный университет обороны Украины имени Ивана Черняховского, Киев*

ИСПОЛЬЗОВАНИЕ ПРИНЦИПОВ SOLID. ПРИ РАЗРАБОТКЕ ВИДЕО ИГР НА ОСНОВЕ ИГРОВОГО ДВИГАТЕЛЯ UNITY

На сегодняшний день мировая индустрия видео игр находится с в своем расцвете. С открытием различных платформ для реализации видео игр, таких как Steam, Google Play, Apple Store а также бесплатных игровых двигателей таких как Unity и Unreal Engine порог входа в гейм-дев для начинающих и студентов заметно снизился, особенно если говорить о рынке мобильных игр. С минимальными затратами времени и усилий, используя все мощности вышеупомянутых игровых двигателей, процесс разработки полноценной видео игры может быть сведен до нескольких месяцев, а то и недель. Однако часто разработчики начинающие, так же как и программисты с опытом, в погоне за быстрой разработкой заметно усложняют себе процесс дальнейшей поддержки и развития своих игровых проектов. Одним из способов избежать такого развития событий является использование принципов SOLID при проектировании игрового приложения.

В данной статье основное внимание уделяется обзору архитектуры программных продуктов на основе игрового движка Unity и возможностям внедрения принципов SOLID для оптимизации архитектуры, и облегчению дальнейшей поддержки и модификации таких приложений. Рассмотрены и соответствие основных компонент игрового движка UNITY принципа единого долга а также особенности построения классов при управлении данным принципом. Определены основные особенности использования принципа открытости/ закрытости и его влияние на процесс дальнейшей модернизации программных архитектур. Предложено архитектурные решения для использования принципа подстановки Лескова и

принципа разделений интерфейсов. Осмотрено основные требования к абстарцый при работе с принципом инверсии зависимостей.

Предложен общий скелет для построения игровых архитектур с учетом программного принципа SOLID на основе игрового приложения в жанре rpg, с реализацией отдельных игровых механик. Алгоритмизированной построение игровой архитектуры на основе принципа SOLID, что приведет к упрощению процесса модернизации и поддержки игровых приложений разработанных на основе игрового движка UNITY

Ключевые слова: *ООП C#, ООП Java, SOLID, UNITY, розробка игр, принципы программирования, паттерны программирования.*

Вступ

Методом проб і помилок, програмісти з плином часу прийшли до деяких правил, які успішно використовуються на практиці і завжди приводять до хороших результатів. Ці принципи були винесені в одну збірку, названу SOLID. Фактично це було зроблено для полегшення інтеграції нових програмістів на існуючі проекти а також полегшення процесу підтримки да модернізації таких проектів. В свою чергу це під час розвитку, ігрового двигуна Unity, навколо нього утворилася велику общину програмістів, підказками та навчальними матеріалами від яких, користуються студенти та люди які хочуть вивчити даний ігровий двигун і більшість таких навчальних матеріалів розглядає лише компонентний підхід до побудови ігрових додатків, без використання принципу SOLID чи будь-яких інших принципів та патернів програмування. В свою чергу це призводить до ускладнення подальшої модернізації та підтримки таких ігрових проектів а також неготовності програмістів працювати над великими проектами які знаходяться в розробці на протязі кількох років. В зв'язку з цим виникла ідея розглянути основні можливості використання принципів SOLID в середині ігрового двигуна Unity.

1. Загальні підходи до використання принципів SOLID в контексті створення ігрових додатків

Принцип SOLID являє собою абревіатуру для позначення об'єднання п'яти принципів програмування (рис.1.), призначений зробити програмний дизайн продуктів чистішим, більш гнучким та легшим в підтримці та модернізації. Ці принципи – це підмножина багатьох принципів та ідей, запропонованих Робертом К. Мартіном і хоча вони використовуються для будь-якого об'єктно орієнтовного дизайну, принципи SOLID можуть також формувати основну філософію для таких методологій, як гнучка або адаптивна розробка програмного забезпечення. Теорія, що стоїть в основі принципу SOLID була запропонована Робертом К. Мартіном в його статі «Design Principles and Design Patterns» в 2000 році [1].

SOLID завжди згадують в контексті об'єктно-орієнтованого програмування (ООП), оскільки саме в мовах, що базуються на цьому принципі, з'явилась зручна і безпечна підтримка динамічного поліморфізму. Фактично, в контексті ООП під SOLID розуміють саме динамічний поліморфізм.

В загальному випадку поліморфізм дає нам можливість використовувати один і той самий код для різних типів. Умовно поліморфізм можна розділити на динамічний і статичний:

Динамічний – визначення типу об'єктів з якими працює код відбувається під час роботи додатку. Сюди можна віднести абстрактні класи і інтерфейси.

Статичний – визначення типу об'єктів з якими працює код відбувається на етапі компіляції, коли з одного шаблонного коду генерується окремий код, відповідно до типу що використовується. Сюди можна віднести generics.

Рис.1. Принципи SOLID [1]

Крім звичних для ООП мов програмування, таких як Java, C#, Ruby, JavaScript, динамічний поліморфізм реалізований наприклад в Golang (за допомогою інтерфейсів), Clojure (за допомогою протоколів і мульти-методів).

Що стосується Unity то ігровий двигун являє собою програмну оболонку, в якій реалізовано велика кількість потрібних для розробки ігор механік, таких як:

- ігрова 2 і 3 фізика;
- 2 і 3 рендеринг;
- підтримка анімації;
- підтримка аудіо записів;
- підтрика мультиплеєра.

Для цього в ігровому двигуні було створено велика кількість бібліотек та класів наслідуючись від яких користувачі отримують доступ до вищезгаданих функцій. За рахунок цього створення ігрових додатків в середині Unity зводиться до виклику методів з базових класів двигуна. Також одним з підходів до програмування в середині Unity є так звана, компонентна архітектура, де кожен клас є компонентом, що додається на ігровий об’єкт в середині двигуна, що в свою чергу аналогічно до створення екземпляру класу і віддаляє роботу від принципів абстракції яким і є SOLID. Це в свою чергу призводить до ускладнення процесів підтримки та модернізації ігрових продуктів.

Насправді, розглядаючи Unity більш детально можна побачити, що сам ігровий двигун побудований з використанням принципів SOLID і розрахований на те, що б створені в ньому ігрові додатки керувалися відповідним принципом. Одним з доказів такого підходу є можливість перепису та зміни функціоналу основних методів Unity. В загальному більшість методів Unity реалізовані таким чином, що б спрацьовувати при відповідних обставинах виконуючи код реалізований в них користувачем, наприклад код написаний в методах:

- *Awake()* – виконується один раз після загрузки всіх ресурсів, та перед ініціалізацією сцени;
- *Start()* – виконується один після ініціалізації ігрової сцени;
- *Update()* – виконується в кожен кадр системи;
- *FixedUpdate()* – виконується через заданий проміжок часу.

Одною з передумов використання принципів ООП та SOLID в середині Unity – є можливість реалізувати ці методи з параметрами virtual та override що дозволяє змінювати реалізацію цих методів в кожному з дочірніх класів.

Також велика частина функціоналу ігрового двигуна Unity доступна за рахунок використання інтерфейсів, що викликаються при відповідних обставинах, дозволяючи працювати з вибірковими методами. По суті сам підхід ігровго двигуна Unity до використання фабричних методів, класів та інтерфейсів є реалізаціє принципу відкритості/закритості. Провівши розробку програмного продукту на базі ігрового двигуна Unity, можна розглянути наступні передумови та особливості використання принципів SOLID.

2. The Single Responsibility Principle(SRP) – принцип єдиного обов'язку

Згідно цьому принципу клас повинен мати тільки один обов'язок і як результат одну причину для змін. Що мається на увазі? Коли ми маємо клас який реалізує якийсь функціонал, зміни в ньому призведуть до змін в усіх похідних від нього класах та компонентах, які спираються на даний функціонал. В середині ж Unity, поза змінами самого коду, цілком можливо, що доведеться змінювати також префаби, асети і компоненти. Як результат чим більше ролей буде відігравати окремий клас, тим більше змін доводиться провести в процесі модернізації і тим більш складними ці зміни можуть виявитись.

Так наприклад розглянемо механіку керування ігровим персонажем, можна виділити основні вимоги до такого класу, який назвемо *PlayerController*:

–зчитування натиснення клавіши контролю;

–отримання та перевірка координати руху;

–рух до вказаної точки;

–перевірка можливих об'єктів для взаємодії;

–контроль взаємодії з об'єктом.

В результаті такої архітектури може виникнути наступна низка проблем: клавіша контролю буде залежати від платформи під яку розробляється додаток, як результат при переході від однієї платформи до іншої, буде виникати потреба переписувати частину яка відповідає зчитування вхідної інформації. Що ж до самого руху об'єкта то він не буде залежати від способу отримання напряму чи координати, а опиратиметься на саме їхнє значення то ж не потребуватиме змін. Так само від способу отримання координати руху не буде залежати й процес взаємодії. Цю ж саму логіку можна використати і в зворотному напрямку, оскільки спосіб отримання координат руху не залежить від реалізації руху, та взаємодії з об'єктами. Проаналізувавши це, було б доцільніше розділити перперший варіант класу на три кожен з яких буде справлятися з окремою роллю:

–*InputController* - зчитування натиснень клавіши контролю та отримання координати руху;

–*PlayerController* - перевірка отриманої координати, рух до вказаної точки;

–*PlayerInteraction* - взаємодія з іншими об'єктами.

Також важливо відмітити що компонентна структура ігрового двигуна Unity є одним з варіантів реалізації принципу SRP: де кожна окрема компонента відіграє тільки одну роль і зміни в ній, так само як і її заміна будуть впливати на мінімальну кількість ігрових модулів.

Ще одним способом реалізації принципу єдиного обов'язку в Unity є система вкладених префабів – згідно якої, комплексні об'єкти складаються з окремих частин зміна яких на рівні конструктора префабів буде призводити до зміни схожих префабів в усіх інших типах комплексних об'єктів та не потребуватиме окремих змін в самих комплексних об'єктах. Практичне використання цього принципу буде розглянуто в четвертій частині даної статті.

3. Open/Close Principle(OCP) – принцип відкритості/закритості

Згідно з цим принципом, розробник повинен мати змогу розширити поведінку класу без його зміни. Цей принцип є основним при бажанні розробити по справжньому гнучкий код, за рахунок створення об'єктів та компонент які можуть використовуватись кілька разів. Це досягається шляхом створення абстракцій, на противагу використання конкретних реалізацій[2]. Принцип можливий за рахунок дотримання наступних двох кроків:

1) Відкритість для розширення – замість змінення основного класу шляхом додавання нового коду, ми можемо змусити нащадків обраного класу поводитись по різному в залежності від конкретних обставин.

2) Закритий для модифікацій – клас не повинен змінювати свою основну поведінку, при використанні різними частинами коду. Для цього потрібно використовувати інтерфейси або абстрактні класи як базові типи для конкретних компонент та сутностей. Оскільки в Unity всі класи, які користувач хоче використовувати як компоненти(мати можливість додавати на ігрові об'єкти), мають наслідуватись від класу *MonoBehaviour*, правильним варіантом буде наслідувати абстракцію від цього класу.

Як було згадано вище, принципи роботи Unity є одним з найкращих прикладів реалізації цього принципу: ми маємо набір фабричних класів, які пов'язані з конкретними етапами роботи гри, основну поведінку яких ми не можемо змінити. Натомість в користувацьких класах ми можемо додавати поведінку яку буде опрацьовувати конкретний клас. Так само з компонентами, ми не можемо змінити глобальну поведінку фабричних компонент Unity проте ми можемо змінити особливості поведінки компоненти конкретного об'єкту(наприклад маса об'єкту в компоненті *Rigidbody*). Практичне використання цього принципу буде розглянуто в четвертій частині даної статті.

4. Liskov Substitution Principle (LSP) – принцип підстановки Ліскова

Суть цього принципу полягає в тому, що – класи в програмі можуть бути замінені їхніми нащадками без зміни коду програми. По суті цей принцип описує, що методи, що використовуються посилання на батьківський клас, повинні мати змогу використовувати нащадків цього класу, без знання про це[3].

Як приклад в середині Unity можна розглянути компоненти типу Collider – які відповідають за обробку фізичних границь об'єкту та взаємодій між об'єктами. В двигуні існують наступні типи колайдерів, в залежності від форми об'єкту:

–*BoxCollider;*

–*CapsuleCollider;*

–*SphereCollider;*

–*MeshCollider;*

–*WheelCollider.*

Всі ці компоненти наслідуються від класу Collider, в якому реалізовано основний функціонал і при роботі в користувацьких класах можуть бути оголошені як екземпляр класу Collider, як результат при заміні типу колайдеру на ігровому об'єкті, розробник не матиме потреби змінювати код класу.

На практиці ж розглянемо приклад реалізації ігрової взаємодії між гравцем та ігровими об'єктами. Припустимо, що в нас є 3 типи ігрових об'єктів і відповідних взаємодій, при натисненні на них гравцем на ігровій сцені:

– *Item*(спорядження) – підібрати спорядження в інвентар;

– *Enemy*(ворога) – атакувати ворога та нанести йому пошкодження;

– *NPC*(неігрові персонажі) – почати діалог.

В результаті можна запропонувати наступну архітектуру методу взаємодії:

```
void Interact(Target target)
{
    if (target.type == Enemy)
    {
        Atack();
    }
    else if (target.type == NPC)
    {
        StartDialogue();
    }
    else if (target.type == Item)
    {
        AddToInventory();
    }
 }
```

На перший погляд може не виникнути зауважень, але розглянемо це з боку додавання нового функціоналу: при додаванні нових типів ігрових об'єктів з якими можлива взаємодія, нам кожен раз доведеться змінювати клас що відповідає за внутрішньо ігрову взаємодію, таким чином порушуючи принцип відкритості/закритості. Крім того, така архітектура буде порушувати принцип єдиної ролі, оскільки всі варіанти взаємодії будуть описані в даному класі. Проблеми також будуть виникати при варіативності ігрових об'єктів всередині категорій Item. Enemy та NPC, що призведе до нагромадження коду за рахунок потреби опису кожної окремої взаємодії.

Натомість можна використати наступну архітектуру:

```
void Interact(Interectable target)
{
    target.Interact();
}
```

Архітектура для класу Interectable представлена на рис. 2.

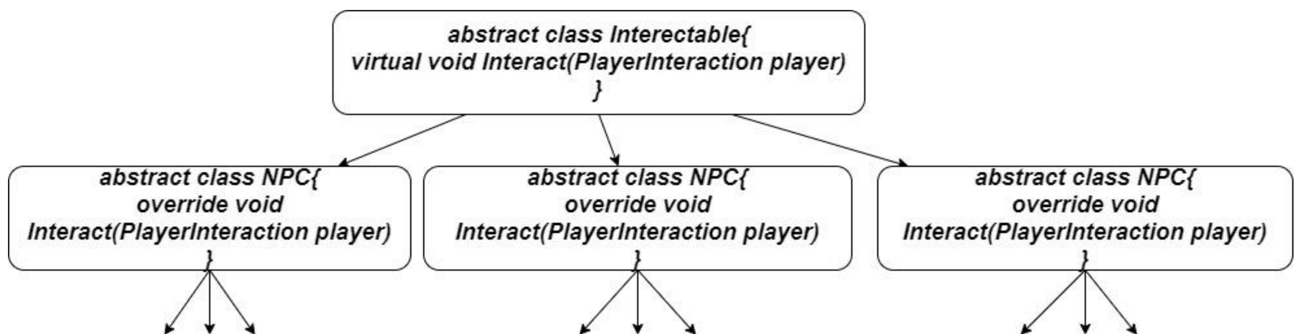

Рис. 2. Архітектура класів для реалізації взаємодії гравця з ігровими об'єктами

При цьому, отримання та перевірка цілі для взаємодії буде відбуватися відповідно в класах *InputController* та *PlayerController*. В класі ж *PlayerInteraction* ми можемо зберігати характеристики гравця (величина пошкоджень, посилання на інвентарь) і вже в конкретному класі, що наслідується від Interectable обробляти ці взаємодії. Також така архітектура вирішує

проблему варіативності в трьох вищеописаних категоріях, за рахунок чого ми можемо додавати нові особливості взаємодії без потреби переписувати класи з основним функціоналом.

5. Interface Segregation Principle(ISP) – принцип розділення інтерфейсів

Багато клієнтських конкретних інтерфейсів краще чим один загальний інтерфейс. Цей принцип рекомендує розбивати монолітні інтерфейси на маленькі частинки, в залежності від їхньої ролі [4].

Прикладом реалізації цього підходу в Unity є робота з переміщенням UI елементів, так в Unity існує цілий список інтерфейсів, в бібліотеці *UnityEngine.EventsSystem*, кожен з яких викликається на різних етапах руху та перетягування UI елементів наприклад: початок перетягування, процес перетягування, кінець перетягування.

Розглядаючи rpg з нашою архітектурою, можна припустити що в нас існуватиме три типи ворогів:

–Mage;

–Warrior;

–Rouge.

Також можна припустити, що кожен з них матиме змогу сатакувати гравця, тому можемо створити інтерфейс IAttaker – в якому буде метод – Atack(), але що тоді робити з унікальними атаками кожного з класів. Наприклад маг – повинен мати змогу накласти закляття, воїн атакувати щитом, а розбійник – кинути кинжал. Ми звичайно можемо додати методи ShieldSlam(), CastSpell(), TrowDagger() в інтерфейс IAttaker, але тоді вийде що в кожного ворога буде присутні два методи які ми маємо реалізувати і які для них не будуть актуальними. Натомість, за рахунок можливості реалізації класом кількох інтерфейсів одночасно ми можемо створити наступні інтерфейси з методами:

–*IAttaker - Atack();*

–*ISpellCaster - CastSpell();*

–*IShieldSlammer - ShieldSlam();*

–*IDaggerThrower - TrowDagger();*

Так само можна працювати і з полями які потрібні окремим сутностям.

6. Dependency Inversion Principle (DIP) – принцип інверсії залежностей

Останній принцип SOLID звучить наступним чином: опирайтеся на абстракції, а не на конкретику[5]. Принцип інверсії залежностей спирається на два простих правила:

1. Високо-рівневі модулі не повинні опиратися на низько-рівневі, обидва мають опиратися на абстракції.

2. Абстракції не повинні опиратися на деталі. Деталі мають опиратися на абстракції.

Щоб детальніше розглянути даний принцип, повернемось до попереднього прикладу. Ми визначили, що три типи ворогів можуть атакувати гравця, але потрібно зауважити, що особливості атаки кожного ворога будуть залежати від його типу, що ми реалізували за допомогою інтерфейсу. Що ж стосується гравця, то його атака буде оброблятся самим ворогом, але від чого тоді буде залежати рівень пошкоджень який може нанеси гравець? Зазвичай в іграх жанру rpg, атака гравця буде залежати від рівню та характеристик зброї. Так ми можемо створити наприклад клас Sword(), екземпляр якого передаватимемо гравцю і в залженості від нього – будуть формуватися вихідні пошкодження. Але що тоді робити, якщо ми захочемо додати сокиру, лук, спис? Тоді нам потрібен спосіб об’єднати всю зброю. Для цього ми можемо створити інтерфейс IWeapon, який буде реалізувати кожен з видів нашої

зброї. Таким чином для гравця не буде жодної різниці в тому яку зброю він використовує для атаки, що дозволить розрозробникам в подальшому з легкістю додавати нові види зброї.

Висновки

Як було показано в статті, принцип SOLID займає важливу роль в побудові якісної та гнучкої архітектури ігрових додатків. Беручи за основу приклади запропонованої архітектури, розробники ігрових додатків зможуть помітно спростити процес подальшої підтримки та модерації своїх додатків. Наведені приклади та алгоритми розробки архітектури можуть бути використані не тільки для розробки відео-ігор, а також як загальні рекомендації для побудови якісної архітектури програмних додатків.

References

1. Martin C. R. (2010) “*Design Principles and Design Patterns.”*, e-sourse – [access mode]: https://objectmentor.booked.net/
2. Martin C. Robert, Martin M.(2006) “*Agile Principles, Patterns, and Practices in C#.”* Prentice Hall.
3. Martin C. Robert. (2017) “*Clean Architecture: A Craftsman's Guide to Software Structure and Design*” Prentice Hall.
4. Baron D. (2019) “*Hands-On Game Development Patterns with Unity 2019: Create engaging games by using industry-standard design patterns with C#”* Packt Publishing.
5. Nystrom R. (2014) “*Game Programming Patterns”* Genever Benning, 2014.