УДК 004.421.72/74 http://doi.org/10.15407/pp2024.02-03.092

*В.М. Бездітний, О.В. Чебанюк*

# МЕТОДИКА ПРОЄКТУВАННЯ МУЛЬТИМЕДІЙНИХ ЗАСТОСУНКІВ ДЛЯ ІГРОВИХ РУШІЇВ З КОМПОНЕНТНО-ОРІЄНТОВАНИМ АРХІТЕКТУРНИМ СТИЛЕМ

У статті представлено методику проєктування мультимедійних застосунків для ігрових рушіїв з компонентно-орієнтованим архітектурним стилем. Робота акцентує увагу на важливості вибору відповідного архітектурного патерну, враховуючи особливості сучасних ігрових рушіїв, таких як Unity, Unreal Engine та Godot Engine. Зазначено ключові проблеми та виклики, пов'язані з проєктуванням гнучких та масштабованих архітектур.
Описано модель життєвого циклу сеансу взаємодії користувача із застосунком, яка включає три основні етапи: Bootstrap, GameLoop та Dispose. Показано важливість контролю перехідних процесів між станами життєвого циклу для забезпечення коректного функціонування застосунку. Для кращої організації взаємодії сервісів застосовано математичний апарат теорії категорій та множин, що дозволяє чітко визначити та управляти залежностями та відносинами між компонентами.
Описано переваги застосування теорії категорій для моделювання та управління потоками даних та залежностей у системі, зокрема, через функтори та монади. Ці підходи дозволяють створювати адаптивні та масштабовані системи, які легко підтримувати та розширювати.
Робота містить практичні рекомендації для проєктування архітектури ігрових застосунків, враховуючи специфіку компонентно-орієнтованого та сервіс-орієнтованого архітектурних стилів.
Обґрунтовано необхідність регулярного перегляду та оновлення архітектури проекту відповідно до змін у вимогах та умовах експлуатації. Запропоновано методику, яка є теоретичним підґрунтям для розробки гнучких архітектур мультимедійних застосунків. Вона враховує специфіку компонентно-орієнтованого та сервіс-орієнтованого архітектурних стилів, а також особливості функціонування ігрових рушіїв.
У роботі також розглянуто приклади практичного застосування запропонованих теоретичних підходів у реальних проєктах, що демонструє їх ефективність та застосовність у різних контекстах розробки ігрових застосунків. Ці приклади включають конкретні реалізації патернів, управління станами за допомогою State Machine та оптимізацію взаємодії між компонентами через Service Locator та Dependency Injection.
Ключові слова: ігровий рушій, патерни проектуванння, Unity, сервіс, теорія категорій.

*V. Bezditnyi, O. Chebanyuk*

# DESIGN METHODOLOGY OF MULTIMEDIA APPLICATIONS FOR GAMING ENGINES WITH A COMPONENT-ORIENTED ARCHITECTURAL STYLE

The article presents the methodology of designing multimedia applications for game engines with a component-oriented architectural style. The work emphasizes the importance of choosing the appropriate architectural pattern, taking into account the features of modern game engines such as Unity, Unreal Engine and Godot Engine. The key problems and challenges associated with the design of flexible and scalable architectures are indicated.
The life cycle model of the user interaction session with the application is described, which includes three main stages: Bootstrap, GameLoop and Dispose. The importance of controlling transient processes between life cycle states to ensure the correct functioning of the application is shown. For a better organization of the interaction of services, the mathematical apparatus of the theory of categories and sets is applied, which allows you to clearly define and manage dependencies and relationships between components.
The advantages of using category theory for modeling and managing data flows and dependencies in the system, in particular through functors and monads, are described. These approaches enable the creation of adaptive and scalable systems that are easy to maintain and extend.
The work contains practical recommendations for designing the architecture of game applications, taking into account the specifics of component-oriented and service-oriented architectural styles.
The need for regular review and updating of the project architecture in accordance with changes in requirements and operating conditions is substantiated. A methodology is proposed, which is a theoretical basis for the development of flexible architectures of multimedia applications. It takes into account the specifics of component-oriented and service-oriented architectural styles, as well as the peculiarities of the functioning of game engines.
Examples of the practical application of the proposed theoretical approaches in real projects are also considered in the work, which demonstrates their effectiveness and applicability in various contexts of game application development. These examples include concrete implementations of patterns, state management using the State Machine, and optimizing interaction between components through Service Locator and Dependency Injection.
Key words: game engine, design patterns, Unity, service, category theory.

© В.М. Бездітний, О. В. Чебанюк, 2024

## Вступ

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. Основою ігрового застосунку є ігровий рушій, який є середовищем для розробки та прототипування ігрових застосунків зі своїми особливостями та мовою програмування. Серед популярних ігрових рушіїв найчастіше виділяють Unity Engine, Unreal Engine, Godot Engine рис 1 [1].

## 1. Проблемні питання та виклики проєктування гнучких архітектур для сучасних ігрових рушіїв

Проєктування архітектури будь-якого застосунку може становити певні виклики та проблеми, зокрема:

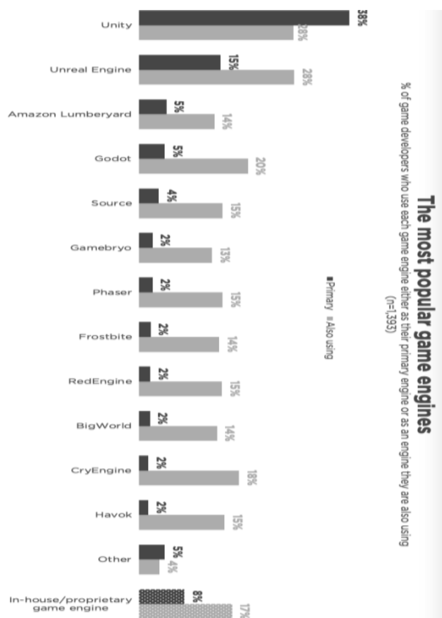


Рис. 1. Найбільш популярні у світі ігрові рушії[1]

1. Вибір відповідного архітектурного патерну: ігрові рушії не накладають 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 проєкту [2].

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

3. Користуючись доступним функціоналом ігрових рушіїв без спроєктованої архітектури та без використання кращих практик веде до створення сильно зв'язаних систем, що ускладнює їх тестування та рефакторинг. Використання принципів SOLID та патернів проєктування, 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 проєктах з великою кількістю об'єктів і компонентів. Оптимізація і правильний вибір патернів є ключовими для підтримки високої продуктивності.

6. Інтеграція із 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 проєкту.

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

На сьогоднішній день існує велика кількість патернів та реалізацій їхніх комбінацій. Для того, щоб обрати найбільш придатну архітектуру пропонується розг-

лянути виклики, з якими доводиться справлятися розробникам.

Перший – це отримання залежностей від іншого класу. Зазвичай це вирішується налаштуванням зв'язків між компонентами у середовищі розробки, використанням патерну Singleton, статичних полів чи івентів. Недоліки цього підходу стають помітними у процесі наповнення проєкту об'єктами та логікою. Альтернативою є Dependency Injection (DI), що дає змогу передати посилання га об'єкт через механізм ін'єкції залежностей.

Інший виклик це підтримання структури проєкту, мінімізація проблем порядку ініціалізації. Для цього потрібно чітко контролювати життєвий цикл застосунку, а також ігрових об'єктів. Для визначення етапів життєвого циклу відповідає 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 частинах проєкту? Якщо логіка глобальна для всіх користувачів і 100 користувачів звернуться із запитами до однієї служби одночасно, то результат у кращому випадку отримають лише декілька. Саме цю проблему вирішує патерн Service Locator у поєднанні з патерном State для забезпечення послідовного виконання запитів у порядку, визначеному станом [1-3].

Розглянемо детальніше стани та опис ігрового циклу застосунку.

## 2. Технології підтримки функціонування ігрового застосунку

Говорячи про будь-який застосунок (додаток для навчання, гра тощо), життєвий цикл сеансу виглядає приблизно таким чином (рис. 2):

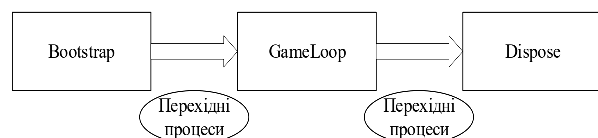


Рис. 2. Модель життєвого циклу сеансу взаємодії користувача із застосунком

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) містяться 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.

Для коректної роботи додатку нам необхідно 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.

## 3. Формальна модель ігрового процесу, представлена за допомогою апарату дискретної математики

З метою інтерпретації ігрового циклу в математичну модель можливо використати ***теорію категорій та теорію множин***. Розглянемо основні аналогії:

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 для 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 їхні 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 у комплексних проєктах.

## 4. Модель функціонування ігрового застосунку

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, тоді як перехід визначає рух з одного стану в інший. Дії, асоційовані зі станами або переходами, представляють логіку, яку потрібно виконувати у процесі входу, виходу або під час перебування в стані [4].

На рисунку 3 зображено послідовність запуску програмного застосунку (зокрема, гри).

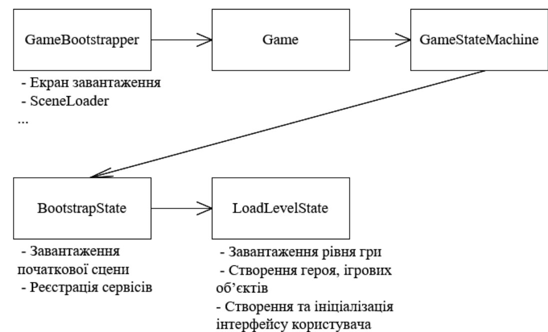


Рис. 3. Схема послідовності запуску застосунку

Перший крок у реалізації «State Machine» полягає у визначенні станів. Кожен стан буде представлений окремим скриптом (логікою), що наслідується від інтерфейсу IState, який декларує 2 основних методи: вхід до певного стану та вихід з нього.

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

Клас GameStateMachine може мати різні стани (наприклад: `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) в інший.

Одним із ключових аспектів теорії категорій є композиція морфізмів, що в 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.

Розглянемо наступну формулу (1), яка описує перехід з BootstrapState до GameLoopState через LoadLevelState:

LLS = BS --> LPS --> GLS (1)

Цю формулу можна інтерпретувати як композицію двох морфізмів:

- BootstrapState-> LoadProgressState. Цей морфізм відповідає переходу з BootstrapState до LoadProgressState.

- LoadProgressState->GameLoopState Цей морфізм відповідає переходу з LoadProgressState до GameLoopState.

Композиція цих двох морфізмів дає морфізм BootstrapState -> GameLoopState, який відповідає прямому переходу з BootstrapState до GameLoopState.

Даний приклад демонструє, як теорія категорій може використовуватися для описування та розуміння переходів між станами в системі станів гри.

Ця математична модель є спрощеним представленням системи станів гри. Реальна система може бути набагато складнішою, з більшою кількістю станів, переходів та взаємодій.

Теорія категорій надає потужний інструментарій для моделювання та аналізу систем станів, але для розуміння та застосування її концепцій потрібні певні знання абстрактної математики.

## 5. Проєктування архітектури ігрових застосунків, використовуючи теорію категорій

Проєктування 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 [5].

SOA визначає спосіб, як 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 чи 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, а це зменшує залежність між додатками.

Нижче наведено ключові аспекти для проєктування SOA в Unity:

- Визначення сервісів. Функціональні елементи застосунку можуть бути відділені як самостійні сервіси. Це можуть бути, зокрема, 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.
- Перевірка залежностей та ін'єкція залежностей. Використання шаблону Dependency Injection допоможе керувати залежностями між різними сервісами та компонентами. Це може бути реалізовано через конструктори, сеттери або через спеціальні фреймворки (Zenject [6], VContainer [7]).
- Проєктування менеджеру сервісів. Для керування сервісами необхідно створити центральний менеджер сервісів (Service Locator), який буде відповідати за ініціалізацію, зберігання та доступ до різних сервісів у застосунку.
- Проєктування модульної та гнучкої архітектури. Кожен сервіс необхідно розробити так, щоб він був самодостатнім і міг функціонувати незалежно від інших частин системи. Це забезпечить високу гнучкість і спростить тестування та розвиток проєкту.
- Розгортання й оновлення сервісів. Способи розгортання та оновлення окремих сервісів повинні мати змогу оновлювати сервіси без зупинки всієї системи.
- Тестування. Кожен сервіс проєктується 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.

## Висновки

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

Описана інтерпретація дає змогу контрольовано масштабувати архітектуру проєкту та покращує розуміння виконання процесів та логіки у застосунку.

Запропонована методика є теоретичним підґрунтям для розробки гнучких архітектур мультимедійних застосунків. Вона враховує специфіку компонентно-орієнтованого та сервіс-орієнтованого архітектурних стилів, а також особливості функціонування ігрових рушіїв.

## Література


1. Kaushal Kishor, Rupa Rani, Atul Kumar Rai. 3D Application Development Using Unity Real Time Platform. Proceedings of Fourth Doctoral Symposium on Computational Intelligence. 2023. P. 665–675.
2. Using the Game Engine Unity Efficiently in Teaching: Development of a fully-automated webserver-based

build pipeline / P. Mosler et al. *eCAADe 2023: Digital Design Reconsidered*, Graz, Austria, 20–22 September 2023. 2023. URL: https://doi.org/10.52842/conf.ecaade.2023.2.883 (дата звернення: 05.04.2024).

3. Mobile Game Development Using Unity Engine. Methodologies and Intelligent Systems for Technology Enhanced Learning, Workshops - 13th International Conferenc. 2023. P. 129–138. URL: https://www.researchgate.net/publication/373475694_Mobile_Game_Development_Using_Unity_Engine (date of access: 02.04.2024).
4. Sufyan bin Uzayr. Mastering Unity: A Beginner's Guide (Mastering Computer Science) 1st Edition. CRC Press, 2022. 260 c.
5. What is service-oriented architecture (SOA)?. URL: https://www.ibm.com/topics/soa (дата звернення: 03.04.2024)..
6. GitHub - modesttree/Zenject: Dependency Injection Framework for Unity3D. *GitHub*. URL: https://github.com/modesttree/Zenject (дата звернення: 03.04.2024).
7. About | VContainer. About | VContainer. URL: https://vcontainer.hadashikick.jp/ (дата звернення: 02.04.2024)..

Одержано: 12.02.2024
Внутрішня рецензія отримана: 19.02.2024
Зовнішня рецензія отримана: 08.03.2024

***Про авторів***:

[1]*Бездітний В'ячеслав Миколайович*,
асистент кафедри інформаційної безпеки НН ФТІ.
https://orcid.org/0009-0002-7319-964X.

[2]*Чебанюк Олена Вікторівна*
дослідник
Instituto de Investigación en Inteligencia Artificial (IIIA) Consejo Superior de Investigaciones Científicas (CSIC).
https://orcid.org/0000-0002-9873-6010

***Місце роботи авторів:***

[1]НТУУ «КПІ ім. І. Сікорського»,
тел. +38-097-762-57-48
E-mail: vyachbezd@lll.kpi.ua

[2] Instituto de Investigación en Inteligencia Artificial (IIIA)
E-mail: elena.chebanyuk@iiia.csic.es