Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Програмна система моделювання складної поведінки ігрового штучного
Назва теми
інтелекту

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРПЗ. 160114.01.05.ПЗ

Виконав студент 2 курсу, група ПЗм-20-1 _____ В. М. Кривий
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Т. В. Шестакевич
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук,доцент _____ О. М. Яшина
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

3 грудня 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
01 09 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Кривому Віталію Миколайовичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Програмна система моделювання складної поведінки ігрового штучного інтелекту

Керівник проєкту (роботи) Шестакевич Тетяна Валеріївна
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2021 р. № 102
2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2021 р.
3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Аналіз сучасних методів моделювання складної поведінки ігрового штучного інтелекту
2 Модель та методи розробки програмної системи моделювання складної поведінки ігрового штучного інтелекту
3 Архітектура програмної реалізації системи моделювання складної поведінки ігрового штучного інтелекту
4 Програмна реалізація системи моделювання складної поведінки ігрового штучного інтелекту
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

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

7. Дата видачі завдання «01» вересня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1. Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; визначення структури дипломної роботи	01.09 - 07.09.2021	
2. Аналіз предметної області, актуальних технологій, моделей та методів	08.09 - 25.09.2021	
3 Розробка моделей та методів вирішення завдання	26.09 - 10.10.2021	
4. Робота над науковими публікаціями	11.10 - 20.10.2021	
5. Проектування архітектури системи для вирішення задачі, розробка вимог.	11.10 - 26.10.2021	
6 Детальний опис реалізації системи та тестування	27.10 - 15.11.2021	
7 Оформлення пояснювальної записки згідно вимог чинних стандартів	16.11 - 30.11.2021	
8 Попередній захист дипломної роботи	17.11.2021	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12 - 04.12.2021	
10 Підготовка до захисту дипломної роботи	06.12 - 08.12.2021	

Студент _____ В. М. Кривий
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Т. В. Шестакевич
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Програмна система моделювання складної поведінки ігрового штучного інтелекту».

Автор роботи: Кривий Віталій Миколайович.

Керівник роботи: Шестакевич Тетяна Валеріївна.

Пояснювальна записка: 86 с., 23 рис., 5 табл., 6 дод., 18 джерел.

UNITY, AI, ШТУЧНИЙ ІНТЕЛЕКТ, NPC, ВУЗЛОВА МОДЕЛЬ

Об’єктом дослідження є процес моделювання складної поведінки ігрового штучного інтелекту.

Метою дослідження є вдосконалення методу моделювання складної поведінки ігрового штучного інтелекту на основі вузлового підходу.

У роботі використані наступні методи дослідження та апаратура:

– спостереження, експеримент, абстрагування, аналіз та синтез, формалізація;

– сучасні інструментальні засоби проектування та програмування;

– персональний комп’ютер.

У дипломній роботі досліджено існуючі методи моделювання поведінки ігрового штучного інтелекту, їх основні переваги та недоліки, та сформовано вимоги для їх вдосконалення.

У дослідженні удосконалено метод розробки алгоритмів поведінки ігрового штучного інтелекту з метою підвищення їх гнучкості. Дана мета була досягнута за рахунок впровадження вузлового моделювання алгоритму поведінки та використання зв’язних модулів. Обґрунтовано доцільність використання даного методу та його основні переваги. Результатом дослідження є покращений метод моделювання складної поведінки ігрового штучного інтелекту.

На основі проведеної роботи розроблено функціональну модель програмної системи для побудови та реалізації моделей поведінки штучного інтелекту, виконано

програмну реалізацію програмної системи. Для розробки програмної системи використано мову програмування C#, програмну платформу .NET Standart 2.0 та ігровий рушій Unity.

У підсумку проведено апробацію та тестування програмної системи, випробувано її ефективність та відповідність вимогам. Завдяки високому показнику гнучкості до змін алгоритму, а також можливості розробки візуального редактора моделей поведінки ігрового штучного інтелекту, рекомендується використовувати дану модель для реалізації даного роду задач при розробці ігрових рішень.

03. 12.21

ABSTRACT

Master’s thesis: «Program System for Complex Behavior Modeling of Game Artificial Intelligence».

Author: Kryvyi Vitalii Mykolayovych.

Head of work: Shestakevich Tatiana Valerievna.

Master’s thesis consists of: 86 p., 23 pc., 5 tb., 6 add., 18 srs.

UNITY, AI, ARTIFICIAL INTELLIGENCE, NPC, NODAL MODEL.

The object of the research is the process of complex behavior modeling of game artificial intelligence.

The purpose of the research is to improve the method of modeling the complex behavior of gaming artificial intelligence based on the nodal approach.

The following research methods and equipment are used in the work:

– observation, experiment, abstraction, analysis and synthesis, formalization;

– modern design and programming tools;

– personal computer.

The study examined the existing methods of modeling the behavior of game artificial intelligence, their main advantages and disadvantages are investigated, and the requirements for their improvement are formed.

The study improved the method of developing algorithms in order to achieve better flexibility when changing them, based on this improved method of modeling the behavior of game artificial intelligence by introducing nodal modeling of the algorithm of behavior and the use of connected modules. The expediency of using this method and its main advantages is substantiated. The result of the study is improved method for modeling the complex behavior of gaming artificial intelligence.

Based on the defined requirements, the functional model of software system for construction and realization of models of behavior of artificial intelligence is developed, the

software implementation of the software system. The C # programming language, the .NET Standart 2.0 software platform and the Unity game engine were used to develop the software system.

In the end, the software system was tested and tested, its efficiency and compliance with the requirements were tested. Due to the high rate of flexibility to change the algorithm, as well as the ability to develop a visual editor of behavioral models of artificial intelligence, it is recommended to use this model to implement this type of task in the development of game solutions.

03.12.21

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AI – Artificial Intelligence

ШІ – штучний інтелект

ООП – об'єктно-орієнтоване програмування

OCP – Open Closed Principle

DRY – Don't Repeat Yourself

SRP – Single Responsibility Principle

UML – Unified Modeling Language

API – Application Program Interface

NPC – Non-Playable Character

SOLID – Single-Responsibility, Open-Closed, Liskov-Substitution, Interface-Segregation, Dependency-Inversion

LSP – Liskov-Substition Principle

ВСТУП

Розвиток комп’ютерних систем сприяє активному розвитку і систем симуляцій поведінки реальних об’єктів на основі їх віртуальних моделей. На сьогоднішній день є багато галузей на ринку програмного забезпечення, які потребують постійного розвитку таких моделей, наприклад – ігрова індустрія. В першу чергу це стосується саме візуального сприйняття моделей, що досягається за рахунок покращення технологій графічного рендерингу. Але моделям не достатньо виглядати правдоподібно. Адже користувач очікує, що об’єкти реалістично симулюють поведінку свого аналогу з реального світу. Чимало таких об’єктів в об’єктивній або вигаданій реальності є розумними живими істотами та персонажами, кожен з яких володіє індивідуальністю в поведінці, реагує по своєму на зміни в навколишньому середовищі

Щоб розробляти нові проекти, які відповідатимуть очікуванням гравців, програмісти почали писати ще більше станів для кожного персонажа, створюючи привабливих ворогів, важливих персонажів-союзників, і це дало гравцям більше можливостей, які допомогли переосмислити різні жанри та створити нові. Звісно, це також стало можливим, оскільки технології постійно вдосконалювалися, дозволяючи розробникам впроваджувати ще більше штучного інтелекту у відеоіграх [13].

Важливою частиною ігрового штучного інтелекту є те, що персонажі ШІ реагують на інші частини гри реалістичним шляхом [3]. Існує чимало хороших прикладів серед наявної на ринку продукції, де саме поведінка штучного інтелекту стала однією з візитних карток ігрового продукту. Наприклад, «Alien: Isolation» 2014 року від компанії Creative Assembly дає гравцю в супротивники дуже небезпечного і непередбачуваного антагоніста. Завдання гри – викликати постійну напругу через страх зустрічі з головним ворогом – Чужим. Штучний інтелект, який керує цим персонажем, поводить себе вкрай непередбачено і індивідуально кожен раз, коли в нього з’являється шанс себе проявити. Розробники реалізували концепцію «Психопатична випадковість»: Антагоніст завжди опиняється в потрібному місці в

потрібний час [10]. Це створює відчуття реальності і правдивості небезпеки, з якою зіткнувся гравець, що змушує його боятись ризикувати і робити зайві рухи, оскільки він не діє за чітко-визначеним шаблоном поведінки, який можна з часом зрозуміти і користуватись його слабкими місцями. Адже кожен зайвий рух може одного разу виявитись не таким вже й не помітним і завдати клопоту.

Ще один яскравий і більш свіжий приклад реалізації по-справжньому складного і реалістичного штучного інтелекту присутній в грі The Last Of Us 2 від компанії Naughty Dog. Кожному з присутніх персонажів в грі, як важливим сюжетним, так і неважливим, було надано індивідуальність в їх поведінці, або ж на неї впливають емоції під час переживання різних ситуацій в ігровому процесі. Наприклад, побачивши загиблих товаришів, штучний інтелект NPC фокусується на більш пильному стеженні за навколишньою територією [9]. Такий рівень створення поведінки штучного інтелекту вимагає серйозного опрацювання і моделювання для реакції на різні ситуації. Можна лише припустити, наскільки сильно дані аспекті відображаються на складності програмного коду і алгоритмів, але важливим є саме результат, адже сприймається така симуляція дуже правдоподібно.

Часто доволі складний штучний інтелект доводиться реалізовувати розробникам не тільки ігрових проектів з одно-користувацькою сюжетною складовою, але й багатокористувацькою. Адже часто в таких продуктах для гравця довгий час не вдається підібрати суперників або союзників в ігрових сесіях, через що їх доводиться іноді заміняти комп’ютерними аналогами під керуванням штучного інтелекту. Але для того, щоб він таким не здавався і не псував враження від онлайн-сесій, його доводиться робити доволі складним і реалістичним, щоб імітувати поведінку реального гравця. Часто для таких завдань використовується навіть машинне навчання і нейронні мережі, які з одного боку дозволяють імітувати поведінку живої людини на високому рівні, але при цьому є досить дорогими, складними в навчанні і вимагають великих ресурсів розробника.

Актуальність даного дослідження полягає в масовості застосування технологій штучного інтелекту в ігровій індустрії.

Метою дослідження є вдосконалення методу моделювання складної поведінки ігрового штучного інтелекту на основі вузлового підходу.

Завданням дослідження є:

– розробка концепції вузлової моделі, яка допоможе досягти гнучкості та зручності при розробці складного штучного інтелекту;

– визначення основних вимог до реалізації моделей на основі цієї концепції, та функцій, які вона надає;

– проектування програмної системи або інструментарію, які допоможуть проектувати такі моделі та впроваджувати їх в ігрові об’єкти;

– реалізація програмної системи;

– тестування та практичне застосування програмної системи;

– аналіз одержаних результатів та формування рекомендацій щодо їх подальшого застосування.

Об’єктом дослідження є процес моделювання складної поведінки ігрового штучного інтелекту.

Предметом дослідження є методи моделювання та реалізації ігрового штучного інтелекту.

Наукова новизна роботи:

– удосконалено метод моделювання та реалізації ігрового штучного інтелекту на основі вузлового підходу;

– розроблено архітектурні шаблони, які дозволять вирішити проблему повторюваності його коду, дадуть можливість як візуально, так і програмно моделювати його поведінку та легко її модифікувати.

1 ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області і виявлення наявних проблем та завдань

Більшість дво- та тривимірних відеоігор є прикладами того, що комп'ютерні вчені назвали б м'якими інтерактивними моделями комп'ютерного моделювання в реальному часі.

У більшості відеоігор деяка підмножина реального або уявного світу моделюється математично таким чином, що може керуватись комп'ютером. Модель є наближенням і спрощенням реальності (навіть якщо це уявна реальність), тому що недоцільно включати в модель кожне явище, деталізуючи її до рівня атомів або кварків. Таким чином, математична модель – це імітація реального чи уявного ігрового світу. Наближення та спрощення – два найпотужніших інструменту розробника ігор. При майстерному їх використанні, навіть значно спрощена модель іноді майже не відрізняється від реальності – і при цьому приносить набагато більше задоволення [1].

Таким чином, при розробці ігрових рішень розробник працює не просто над потоком передачі і обробки даних в різних формах. Його робота напряму пов'язана з моделюванням і симуляцією реального або вигаданого світу, відображенням деяких процесів та елементів цього світу, при цьому надаючи користувачу можливість взаємодіяти з ним і впливати на нього, часто в режимі реального часу. Якісна реалізація такої симуляції напряму впливає на позитивний користувацький досвід і занурення в ігровий процес, що і є однією з основних задач в розробці гри.

При розробці часто не потрібно ідеально повторювати реальний досвід, щоб створити хорошу гру. Потрібно лише передати суть цього досвіду [12]. Розробники ж можуть передати цей досвід багатьма способами. Основні проблеми і задачі при симуляції і моделюванні ігрового світу описані в Таблиці 1.1.

Таблиця 1.1 – Основні проблеми і задачі при симуляції і моделюванні ігрового світу.

№ з/п	Назва проблеми	Опис проблеми
1	Візуальна симуляція	Необхідність займатись рендерингом ігрових об’єктів в тривимірному або двовимірному просторі з імітацією світла, тіней, тощо.
2	Фізична симуляція	Необхідність взаємодії об’єктів між собою за законами фізики з дотриманням їх фізичних властивостей та оболонок.
3	Симуляція руху та анімацій	Окрім фізичної поведінки, об’єкти повинні мати заготовлені патерни руху для себе або своїх окремих частин, які будуть відображати зміну їх положення, повороту і розміру в просторі, щоб імітувати такі ж з реального світу або вигаданого гейм-дизайнером
4	Взаємодія з ігровим світом	Потрібно реалізувати контроль ігровим процесом і взаємодією ігровим світом з боку гравця за допомогою наявних в нього пристроїв введення
5	Механіки ігрового процесу	Згідно з тими цілями, які стоять перед гравцем в грі, потрібно реалізувати в ігровому світі процесі, які можуть бути використані, щоб допомогти йому досягнути ці цілі, або ж навпаки – перешкоди, які будуть заважати йому їх досягти.
6	Інтерфейс гравця	Оскільки гравець не може і не повинен завжди відчувати, розуміти і пам’ятати інформацію, яку він отримує і використовує при дослідженні ігрового світу, взаємодії з ним і виконанні своїх завдань, то деяка інформація може бути показано графічно або у текстовому вигляді як інтерфейс на екрані.

Кінець таблиці 1.1

7	Штучний інтелект	Деякі ігрові об’єкти повинні мати власні цілі в грі, і приймати рішення, які впливатимуть на ігровий процес, іноді оцінюючи зміни, які відбуваються в ігровому світі самі по собі або ж в наслідок поведінки інших об’єктів.
8	Звукове супроводження	Деякі об’єкти ігрового світу можуть імітувати звуки. Також можуть бути присутні звукові сигнали яку гратимуть інформативну роль, і саундтреки, які доповнюють сприйняття віртуального світу з боку гравця.

Сьогодні вже існує безліч готових і доступних технологій, які дозволяють вбудувати більшу частину описаних вище задач симуляції ігрового світу. Наприклад, для задач рендерингу існують технології шейдерів та графічні інтерфейси, які також працюють на виділеному апаратному відео-процесорі, а за допомогою спеціальних форматів файлів та графічних редакторів моделювання будь-яких тривимірних об’єктів є достатньо тривіальною задачею.

Це ж стосується і фізичної симуляції поведінки об’єкта. Фізичні закони і моделі, які формувались і розвивались протягом всієї людської історії, були використані для реалізації фізичних симуляцій зіткнень, прикладення сил та імпульсів об’єктів. В той час як анімаційні елементи поведінки дуже легко реалізовуються по-кадровим методом і скелетними анімаціями.

Існуючий технологічний стек для створення симуляцій та моделей можна легко прослідкувати на прикладах графічних або ігрових рушіїв. З останніх найпопулярнішим насьогодні є Unity – безкоштовний кросс-платформний ігровий рушій [2].

В таблиці 1.2 приведено аналіз ігрового рушія Unity на предмет наявності технологій для симуляції ігрового світу в Таблиці 1.2.

Таблиця 1.2 – Технології Unity, що дозволяють вирішувати проблеми симуляції ігрового світу

№ з/п	Назва проблеми	Спосіб вирішення в Unity
1	Візуальна симуляція	Графічний функціонал рендерингу, що працює як надбудова для різних графічних інтерфейсів, таких як DirectX, OpenGL, Metal, тощо, і реалізований через компоненти для ігрових об’єктів. Також присутні системи частинок для спеціальних ефектів, бібліотека Post-Processing Stack для обробки і корекції кінцевого зображення, можливість вибирати різні конвеєри рендерингу зображень і писати свої власні шейдери (інструкції, які виконуються на відеочіпі пристрою з метою рендерингу об’єкту в просторі і його розфарбовування)
2	Фізична симуляція	Unity допомагає імітувати фізику у вашому проекті, щоб забезпечити правильне прискорення об’єктів та реакцію на зіткнення, силу тяжіння та різні інших сил. Unity пропонує різні реалізації фізичних механізмів, які можна використовувати відповідно до потреб вашого проекту: 3D, 2D, об'єктно-орієнтовані або орієнтовані на дані. [8]
3	Симуляція руху та анімацій	Система анімаційних станів Animator, яка дозволяє легко і різнобічно налаштовувати анімацію об’єктів.
4	Взаємодія з ігровим світом	Модуль Input, а також новий модуль InputSystem. Вони надають гнучку систему роботи з користувацьким введенням

Кінець таблиці 1.2.

5	Механіки ігрового процесу	C#-компілятор і система написання сценаріїв, яка дозволяє створювати свої компоненти для об’єктів ігрового світу в Unity і взаємодіяти з уже існуючими.
6	Інтерфейс гравця	Пакет інтерфейсних компонентів для створення текстів, зображень на екрані, тощо.
7	Штучний інтелект	Модуль пошуку шляху
8	Звукове супроводження	Аудіо-функціонал Unity включає повністю трьовимірний просторовий звук, змішування в реальному часі і мастеринг, ієрархії змішування, знімки, заготовлені ефекти і багато іншого [7].

На відміну від більшості інших перерахованих проблем, штучний інтелект не представлений великою кількістю запропонованих рушієм рішень, до таких можна віднести хіба що модуль, що вирішує стандартну алгоритмічну задачу пошуку шляху. Штучний інтелект в більш ширшому розумінні цього слова не представлений в рушії ніякими шаблонами чи базовими архітектурними одиницями, які можуть спростити роботу над ним. І це не є недоліком рушія, оскільки реалізація штучного інтелекту дуже залежить від механік ігрового процесу. А ці механіки здебільшого реалізовуються індивідуально для кожного проекту. Тому не тільки Unity, але й більшість усіх інших рушіїв надають небагато спеціального інструментарію для реалізації штучного інтелекту.

Але в той же час, в багатьох ігрових проектах це досить комплексна задача, принципи рішення якої часто повторюються за схожими шаблонами. Наприклад, якщо в поле зору потрапляє об’єкт-ворог, часто треба просто приймати рішення атакувати, незалежно від того чи об’єкт, який приймає рішення, є бойовий гелікоптер чи сторожова собака. Саме ж прийняття рішення може бути реалізоване математичним шляхом через просторові розрахунки відстані, кутів повороту, тощо і в багатьох іграх буде працювати аналогічно як для першого об’єкта, так і для другого.

Оскільки розробка такого функціоналу часто потребує спеціальних знань і великого обсягу роботу, яка індивідуальна для різних проектів, а бібліотечних інструментів для імітації таких рішень обмаль, то навіть існують фахівці, які спеціалізуються на розробці штучного інтелекту в іграх. Програмісти штучного інтелекту працюють тісно з програмістами фізики та геймплею. Вони підзвітні головному дизайнеру і працюють в співпраці з геймплейними програмістами, щоб розробляти дерева рішень і нейронні мережі для формування базису гри [14].

1.2 Порівняльний аналіз переваг та недоліків існуючих рішень

Задача прийняття рішень штучного інтелекту для ігрових об’єктів може бути реалізована багатьма способами різного ступеня складності. Як і в будь-яких інших задачах в програмуванні, завжди можна обійтись без використання додаткових сутностей і абстракцій, застосовуючи процедурне програмування. Розглянемо цей метод на прикладі.

Нехай потрібно розробити функцію, яка обчислює, чи повинен ігровий об’єкт реагувати на гравця, що знаходиться на певній дистанції. Для цього потрібно задавати на вході параметри позиції гравця, який перевіряється, позиції бота, а також відстані для активації, і будемо повертати ІСТИНА в випадку якщо дія повинна бути виконана. Приклад реалізації такого рішення псевдокодом:

```
Чи_Реагувати(Хгравця, Угравця, Zгравця, Хбота, Убота, Zбота, Дистанція_Регування)
Початок
     Дистанція_До_Гравця = Корінь_Квадратний((Хгравця- Хбота)^2+(Угравця-
     Убота)^2+(Zгравця- Zбота)^2)
     Якщо Дистанція_До_Гравця < Дистанція_Реагування Повернути ІСТИНА
     Інакше Повернути ХИБА.
Кінець
```

Судячи з приведеного псевдокоду, дана задача вирішується дуже просто і без великої кількості операцій чи додаткових сутностей, що могли б переобтяжити код проекту. Але часто очікувана поведінка об’єкта буває набагато складнішою, і

структура програмного коду стає громіздкою за рахунок з великої кількості розгалужень і обчислень, що викликає такі проблеми:

– Поведінка ігрового штучного інтелекту може бути задана великою кількістю факторів та перевірок. Це означає, що код даної функції повинен буде розростатись в залежності від кількості поданих факторів. І як це часто буває на практиці, функція може стати громіздкою, нечитабельною, та недоступною для вивчення на предмет наявності помилок і багів, що не є прикладом хорошого стилю в програмуванні. Навіть якщо покращити ситуацію може декомпозиція функції на окремі під-функції, дані функції будуть залишатись в межах свого простору імен або класу і таким чином збільшувати його розмір;

– елементи поведінки можуть постійно повторюватись в різних типах ігрових об’єктів, але при цьому в багатьох випадках – відрізнятись. Це означає, що навіть в функції після декомпозиції доведеться вдаватись до додаткових перевірок, яку саме операцію треба виконати або не виконати при обчисленнях, що так само розтягує програмний код в об’ємі і робить його схильним до виникнення помилок, які важко виявити через цей ж самий об’єм;

– штучний інтелект ігрових об’єктів часто знаходиться на етапі постійного доповнення, коли кожна його версія тестується, перевіряється на коректність поведінки, необхідність її ускладнення за рахунок реалізації додаткових факторів або навпаки, спрощення. Це означає, що в монолітному або частково-монолітному за структурою програмному будуть вноситись постійні зміни, доповнення, видалення частин коду або ж повернення їх назад, які іноді проводити дуже важко, особливо в умовах, коли функція була розроблена одним розробником, а потім доповнюється іншим. Це спричинятиме постійні помилки в логіці роботи, які можуть виникати по ланцюговій реакції через одне некоректне обчислення і потім дуже важко відслідковуватись.

Тому потрібно брати до уваги, що штучний інтелект ігрових об’єктів дуже часто виходить за межі можливостей його розробки рамках одного класу чи сутності, через дуже часті його корекції в циклі розробки, через відносну складність і наявність

великої кількості факторів і перевірок, що впливають на кінцевий результат роботи програмного коду. При цьому реалізувати один шаблон його поведінки для всіх типів об’єктів і в різних проектах іноді буває неможливим – різниця в логіці поведінок об’єкта-корабля, об’єкта-тварини чи об’єкта-солдата є наочною і здебільшого несумісною для розробки загального для всіх цих типів об’єктів рішення, яке навіть за умови наявності добре налагоджених налаштувань в скрипті може не враховувати окремі мінімальні забаганки замовника або дизайнера гри, що змусить вносити зміни в його роботу і зіштовхуватись з переліченими вище проблемами.

Отже, переваги прямого способу:

– просте рішення для простих задач, не обтяжене зайвими сутностями і структурними елементами;

– швидко реалізовується навіть недосвідченим розробником;

– легко досліджується в випадках простого штучного інтелекту, не потребує розуміння патернів та якоїсь складної структури класів, які просто не використовуються.

Недоліки прямого способу:

– з ростом кількості факторів і перевірок код все важче і важче читати і досліджувати, тим паче – вносити додаткові зміни;

– важко узагальнювати елементи поведінки, описані одним монолітним скриптом, для багатьох об’єктів, щоб не описувати ці елементи в різних скриптах ;

– через суцільність і нероздільність коду одні його частини при корекцію можуть змусити інші частини перестати працювати.

Перераховані недоліки змушують відмовитись від монолітного і максимально-простого рішення, вийшовши за рамки реалізації всієї поведінки всього в одній сутності. Однією із найважливіших задач в програмуванні будь-яких складних систем є розподіл програмного коду на структурні одиниці. Будь-який об’єкт світу може задавати свою поведінку як сукупність різних компонентів, які визначають стан об’єкта. При цьому існує клас, який керує цими компонентами, і обирає, який з них

має бути задіяний наразі. Самий такий підхід застосований в архітектурі ШІ з трьох типів сутностей:

– скрипти-агенти;
– скрипти поведінки;
– скрипти, що базуються на скриптах поведінки [15].

Unity є рушієм, що базується на компонентах [6]. Для рушіїв з компонентною архітектурою скриптів даний метод дуже легко реалізовується, оскільки це заохочується самим рушієм.

Дана архітектура дуже проста і дозволяє розбити елементи поведінки на окремі структурні одиниці, що дозволяє розглядати кожну з них відірвано від інших, і помилки в одній з них не створюватимуть помилок в іншій.

Наприклад, потрібно задати штучний інтелект поведінки об’єкту «Винищувач». На етапі аналізу вимог з’ясувалось, що він має такі вміння:

– атакувати ворога з допомогою ракет;
– втікати від ворога;
– шукати ціль;
– виконувати посадку з вимкненням двигуна;
– виконувати виліт з ввімкненням двигуна;
– бути в режимі простою.

Кожен з цих елементів є своєрідним режимом об’єкта, в якому може здійснюватися керування літаком. Отже, потрібен об’єкт-літак, який має свій інтерфейс дій як для гравця, який керуватиме ним, так і штучного інтелекту. Такий може бути набір дій в цьому інтерфейсі:

– задати напрямок штурвалу;
– випустити ракету в ціль;
– запустити двигун;
– вимкнути двигун;
– випустити теплові пастки.

Надалі, для керування цим літаком з боку комп’ютера, потрібні компоненти, які являють собою різні варіанти поведінки літака, і менеджер, який буде вибирати один з них в залежності від ситуації і активувати його його. Умовна діаграма класів такої архітектури зазначена на Рисунку 1.1.

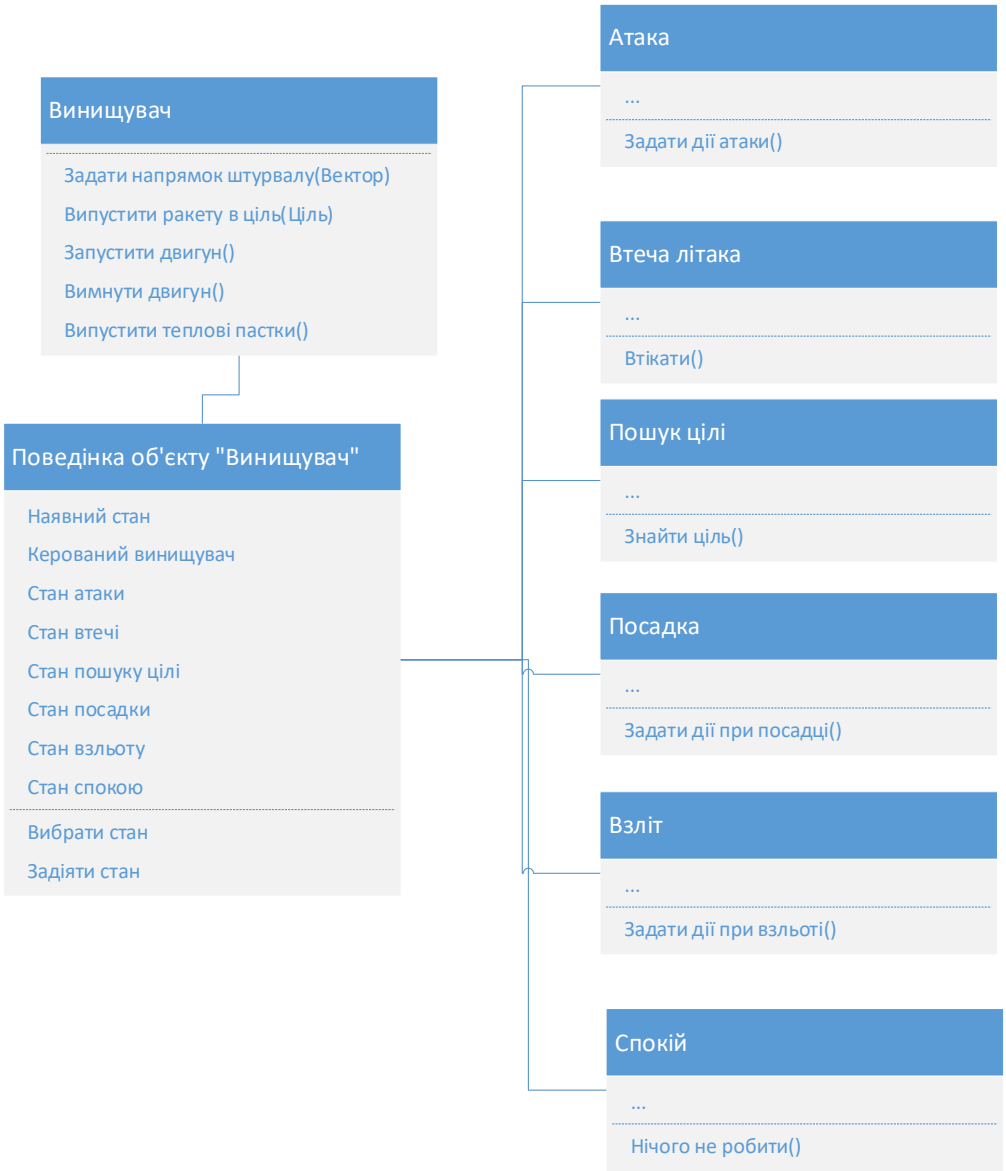


Рисунок 1.1 – Архітектура штучного інтелекту об’єкту «Винищувач» в методі розбиття на компоненти

Тут показано приклад реалізації патерну Фасад, що визначає високорівневий інтерфейс для простішого використання підсистем [5]. В даному випадку таким фасадом виступає сама сутність «Поведінка об’єкту винищувач», яка працює з більш

низькорівневими елементами. Дана архітектура може бути набагато більш децентралізована та вдосконалена за рахунок механізму наслідування, який дозволяє визначити загальну функціональність у батьківському класі, і яка може бути повторно використана, а можливо й змінена дочірніми класами [11]. При імплементації в дочірні реалізації кожен елемент поведінки буде сам перевіряти, чи доцільно йому бути задіяним, і задавати дії, які має виконувати цільовий об’єкт (Рисунок 1.2).

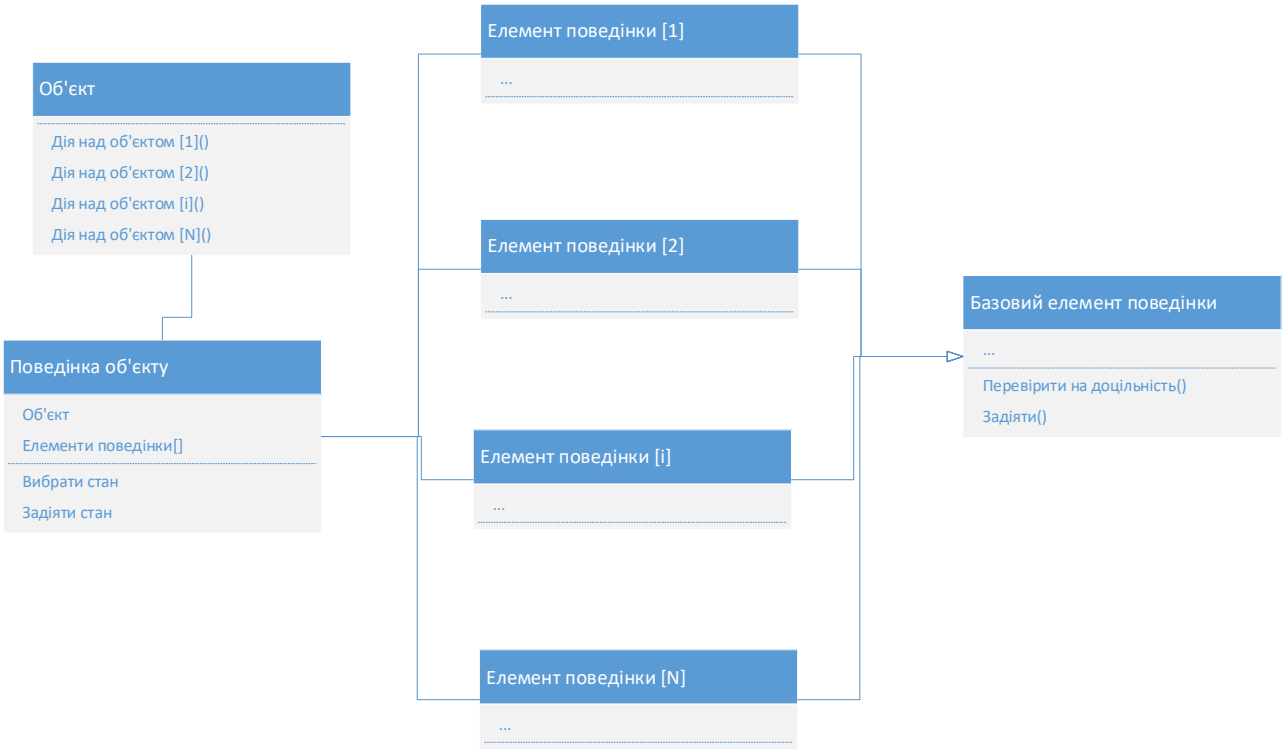


Рисунок 1.2 – Узагальнена структура класів в розподіленій архітектурі

Як уже раніше згадувалось, рушії з компонентною архітектурою доволі легко дають реалізовувати таку структуру. Приклад її налаштування в інспекторі об’єктів Unity зображений на Рисунку 1.3. Тут показано об’єкт, на який додані компоненти і оголошено посилання на ці згідно з вище-описаною діаграмою класів.

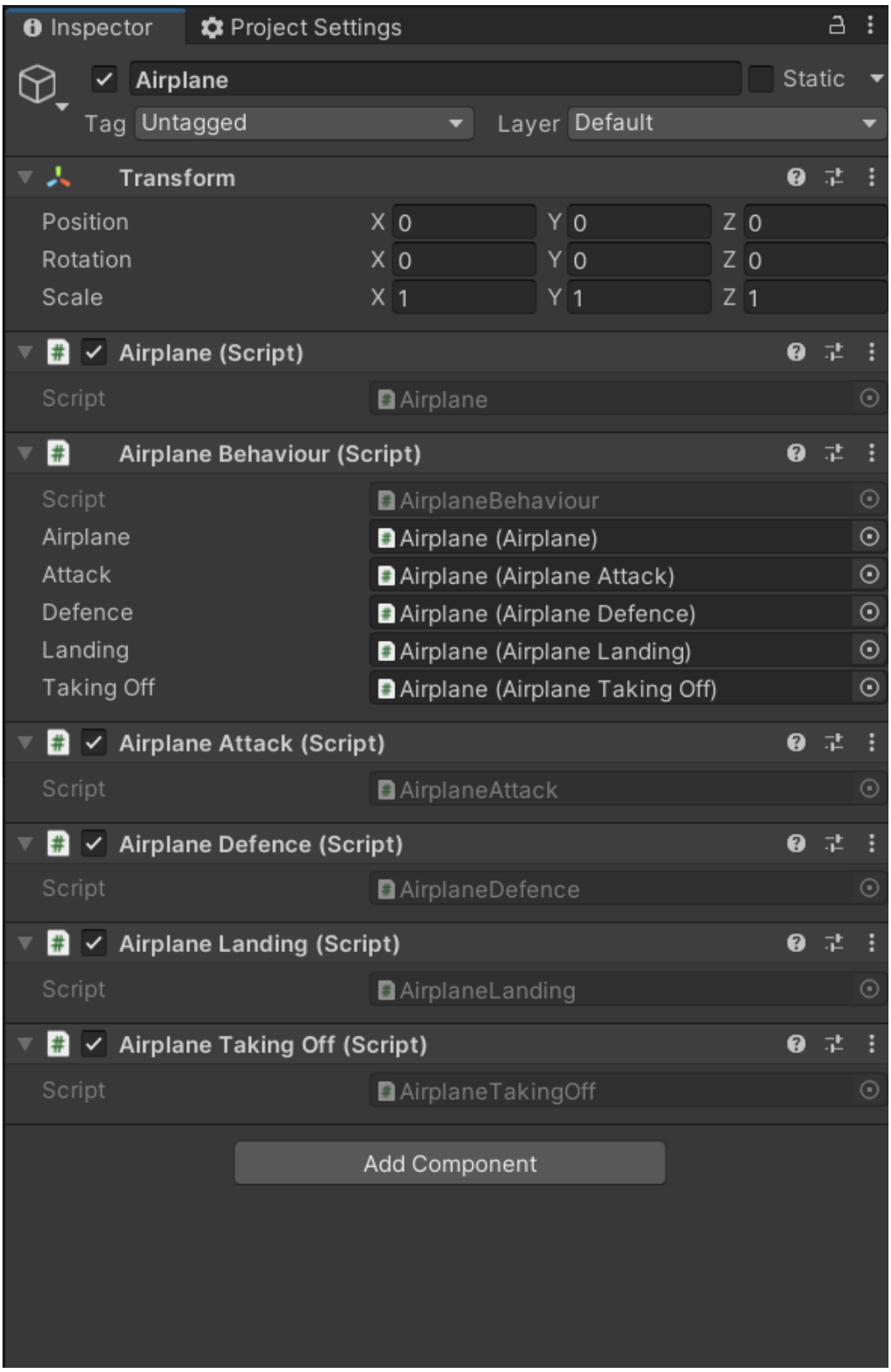


Рисунок 1.3 – Приклад налаштування об’єкту «Літак» під керуванням штучного інтелекту в Unity

Перевагами такого підходу є:

– наявність розподілу відповідальностей між сутностями в коді, через що код стає набагато простіше підтримувати і аналізувати на помилки, просто розглядаючи ті скрипти, з якими і пов’язані знайдені проблеми;

– легкість в доповненні штучного інтелекту через додавання нових сутностей-елементів поведінки;

– можливість налаштовувати як завгодно параметри кожної поведінки в кожній окремій сутності.

Недоліками такого підходу є:

– неможливість вирішити проблему прив’язки усіх скриптів лише до одного типу об’єктів, для інших треба знову реалізовувати цю ж архітектуру і скрипти, які в ній фігурують;

– прийняття рішення все ще здійснюється в рамках одного керуючого скрипта, який в випадках складних сценаріїв поведінки буде мати нагромадження коду перевірок і розрахунків.

1.3 Методологічні підходи до вирішення задачі за темою дослідження

При аналізі існуючих способів вирішення задач з реалізації поведінки штучного інтелекту ігрових об’єктів потрібно орієнтуватись на вирішення таких проблем:

– присутність в наявних моделях механізмів, які гарантують можливість повторного використання окремих частин функціоналу при розробці штучного інтелекту для різних об’єктів, якщо вони будо тісно пов’язані функціональною залежністю з об’єктами, для яких вони розроблені;

– позбавлення необхідності реалізації дуже занадто великих елементів поведінки в рамках одного модуля через декомпозицію;

– можливість створювати шаблони і наперед заготовлені скрипти для повторної використання в інших ігрових проектах, або ж суміжних частинах ігрового функціоналу та геймплею.

Можна зробити висновок про неможливість безпечного повторного використання окремих поведінкових класів і сутностей в інших ігрових об’єктах і тим паче – проектах. Через що доводиться майже в кожному з цих випадків створювати заново нові скрипти штучного інтелекту, дублюючи в них реалізацію механізмів, які вже були використані раніше, фактично знову і знову «придумуючи велосипед».

1.4 Висновки. Постановка задачі

Отже, в даному розділі було проаналізовано область розробки ігрових проектів. Досліджено та проаналізовано існуючі методи та засоби моделювання і реалізації складного штучного інтелекту в ігрових рішеннях, знайдено їх недоліки та переваги.

На основі висновків, підведених даному розділі, можна знайти концепції, які допоможуть позбутись існуючих недоліків та підсилити їх переваги, і на основі них спроектувати нове рішення спектру задач, пов’язаних з ШІ.

Перераховані раніше проблеми можуть бути вирішені покроково з розробкою спеціальних архітектурних патернів, які зможуть зробити сценарії поведінки і прийняття рішень штучного інтелекту настільки гнучкими і розподіленими, що дозволять багаторазово повторно використовувати їх для різних задач. Це може бути зроблено в наступні кроки:

– для вирішення проблеми функціональної залежності скриптів штучного інтелекту від об’єктів, якими вони керують, і таким чином неможливості використовувати їх на інших об’єктах, потрібно розробити механізм відділення тих модулів коду, які виконують основну частину роботи, пов’язану з ШІ, від самих об’єктів, які підлягають керуванню;

– ці модулі повинні вирішувати свої задачі, не знаючи нічого про об’єкти, якими вони керують, але при цьому мати механізм зв’язку з ними. Наприклад, це можна досягти через патерн Адаптер — структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом [4];

– вся система прийняття рішень і визначення поведінки штучного інтелекту повинна бути також розділена на окремі структурні блоки, які з’єднані одне з одним, передаючи дані, які одержали від попереднього блоку, обробляючи їх і передаючи наступному. Тобто, кожен фактор, перевірка або вплив на кінцеве рішення має задаватись окремою сутністю, яку можна буде так само повторно використовувати в інших моделях поведінки;

– кожен цей структурний блок може мати свій тип і реалізацію, але повинен бути сумісний з іншими типами блоків і одержувати від них інформацію. Це також полегшить їх повторне застосування;

– кожен такий структурний блок повинен дозволяти себе налаштовувати, щоб його властивості можна було змінити і впливати на результат їх роботи;

– потрібно мати можливість створювати варіанти однакових блоків з різними налаштуваннями, і задавати ці налаштування, а також зв’язки між блоками, без написання коду, який це робить, на приклад, через серіалізовані поля в Unity. Досягнення цієї мети означатиме, що за умови існування всіх необхідних типів блоків різні типи штучного інтелекту можна буде робити без участі програміста або без необхідності досліджувати код і вносити в нього корективи, а також це дозволить створювати різні варіанти одного і того ж штучного інтелекту з мінімальними відмінностями і тестувати їх, просто перемикаючи їх в вікні інспектора.

Таким чином буде одержана програмна система для моделювання штучного інтелекту, що відповідає таким вимогам:

– максимальна гнучкість моделі, можливість її підлаштовувати під суміжні або схожі за принципом задачі;

– можливість реалізовувати велику кількість факторів, перевірок, які будуть визначені і реалізовані в окремих блоках складної моделі;

– можливість використовувати повторно в інших проектах або для інших об’єктів як саму модель, так і окремі її елементи;

– можливість реалізувати велику кількість блоків різного типу, що майже позбавить необхідності в подальшому працювати з кодом;

– наявність механізму адаптації моделі під будь-який об’єкт, яким вона керуватиме, неприв’язаність моделі до конкретних реалізацій і механік;

Тому актуальність теми дослідження обґрунтована важливістю отримання такої системи моделювання поведінки штучного інтелекту, яка дозволить враховувати і налаштовувати велику кількість факторів та умов, повторно використовувати окремі елементи моделі та налаштовувати їх.

Об’єктом дослідження є процес моделювання складної поведінки ігрового штучного інтелекту.

Предметом дослідження є методи моделювання та реалізації ігрового штучного інтелекту.

Завданням дослідження є:

– розробка концепції вузлової моделі, яка допоможе досягти гнучкості та зручності при розробці складного штучного інтелекту;

– визначення основних вимог до реалізації моделей на основі цієї концепції, та функцій, які вона надає;

– проектування програмної системи або інструментарію, які допоможуть проектувати такі моделі та впроваджувати їх в ігрові об’єкти;

– реалізація програмної системи;

– тестування та практичне застосування програмної системи;

– аналіз одержаних результатів та формування рекомендацій щодо їх подальшого застосування.

2 МЕТОДИКА РОЗРОБКИ ПРОГРАМНОЇ СИСТЕМИ

2.1 Методологія дослідження

Розробку методу побудови програмної системи для моделювання складної поведінки штучного інтелекту буде проведено в наступні кроки:

1) аналіз можливості та доцільності вдосконалення існуючих методів моделювання штучного інтелекту. На даному етапі буде розглянуто існуючі архітектурні рішення, проаналізовано їх можливе вдосконалення, способи їх застосування в програмній системі та оцінено їх переваги та недоліки;

2) розробка покращеного методу створення алгоритмів роботи штучного методу;

3) аналіз результатів дослідження. Порівняння одержаних методів з існуючими на даний момент.

Результатом цих етапів буде метод, незалежний від архітектури, окремої реалізації конкретною мовою програмування чи бібліотекою.

2.2 Способи вдосконалення існуючих методів

Найбільш практичним в масовому застосуванні на сьогоднішній день є метод Агент-Поведінка. Даний метод вимагає в розробника описувати логіку різних варіантів поведінок окремим функціоналом, який слабо пов’язаний з Агентом, що грає роль прийняття рішень і посередництва між штучним інтелектом та самою поведінкою.

Але метод не розглядає способи збереження дрібних повторюваних елементів в функціоналі для повторного використання, і не пропонує архітектурних рішень для безболісного і безпечного розширення або модифікації функціоналу. Щоб одержати вдосконалений варіант методу, або ж повністю замістити метод новою реалізацію, потрібно розглянути практичні аспекти роботи з методом. Це можна зробити через розв’язування типової прикладної задачі проектування штучного інтелекту на основі конкретних заданих вимог.

Нехай розробляється певний віртуальний світ, від якого вимагається симуляція військових дій з застосуванням великої кількості техніки з різними тактичними аспектами застосування. Моделювання поведінки відбувається на основі таких елементів:

– поведінка атаки;

– поведінка очікування;

– поведінка відступу.

Пропонуються наступні типи об’єктів під контролем штучного інтелекту:

– Гелікоптер;

– Танк;

– Літак;

Опис поведінки об’єкту Гелікоптер:

– поведінка атаки – оцінка співвідношення шансів перемоги/поразки, прорахунок дистанції до ворога, прорахунок напрямку до об’єкта, націлювання на об’єкт за трьома осями, рух до об’єкта в двох осях, вогонь на правильній дистанції та повороті;

– поведінка очікування – вибір випадкового вектору руху в напрямку до ворожої сторони в двох осях;

– поведінка відступу – оцінка співвідношення шансів перемоги/поразки, прорахунок дистанції до ворога, прорахунок протилежного від об’єкта напряму, рух від об’єкта в двох осях.

Опис поведінки об’єкту Танк:

– поведінка атаки – оцінка співвідношення шансів перемоги/поразки, прорахунок дистанції до ворога, прорахунок напрямку до об’єкта, націлювання башти на об’єкт за двома осями, націлювання гармати на об’єкт за однією віссю, рух до об’єкта в двох осях, вогонь на правильній дистанції та повороті башти і гармати;

– поведінка очікування – вибір випадкового вектору руху в напрямку до ворожої сторони в двох осях;

– поведінка відступу – оцінка співвідношення шансів перемоги/поразки, прорахунок дистанції до ворога, прорахунок протилежного від об’єкта напряму, рух від об’єкта в двох осях.

Опис поведінки об’єкту Літак:

– поведінка атаки – оцінка співвідношення шансів перемоги/поразки, прорахунок дистанції до ворога, прорахунок напрямку до об’єкта, націлювання на об’єкт за трьома осями, рух до об’єкта в трьох осях, вогонь на правильній дистанції та повороті;

– поведінка очікування – вибір випадкового вектору руху в напрямку до ворожої сторони в трьох осях;

– поведінка відступу – оцінка співвідношення шансів перемоги/поразки, прорахунок дистанції до ворога, прорахунок протилежного від об’єкта напряму, рух від об’єкта в трьох осях.

Для всіх описаних поведінок характерні дуже схожі між собою елементи, але іноді в них відрізняється послідовність застосування, іноді – вісі, в яких вони працюють. Використання моделей Агент-Поведінка допоможе створювати базові класи-каркаси для деяких елементів поведінок з шаблонною послідовністю виконання дій. Але система може дуже сильно масштабуватись або ж видозмінюватись. До існуючих моделей можуть бути додані нові типи техніки та бойових учасників, а всередині кожного типу техніка поведінка може видозмінюватись в залежності від конкретної техніки за її тактико-технічними характеристиками. Наприклад, серед об’єктів типу Танк можуть з’явитись класи – Легкий Танк, Середній Танк, Важкий Танк, тощо.

Для вирішення подібного роду проблем можливі використання наступних прийомів:

– декомпозиція – розбиття функціоналу на менші структурні одиниці;

– шаблонні методи – методи, що надають можливість працювати з типами даних шаблонно;

– абстрагування – створення абстрактних та інтерфейсних елементів функціоналу та їх реалізація і наслідування в конкретні сутності, при цьому взаємодія відбувається на рівні абстракцій.

Декомпозиція може бути використана для винесення однакового функціоналу в окремі модулі. Наприклад, можуть бути виділені наступні модулі:

– модуль вибору випадкового вектору руху;

– модуль оцінки співвідношення шансів поразки/перемоги;

– модуль націлювання об’єкта за двома осями;

– модуль націлювання об’єкта за однією віссю;

– модуль націлювання об’єкта за трьома осями;

– модуль прорахунку напряму до об’єкта;

– модуль прорахунку дистанції;

Завдяки шаблонним методам, можна добитись використання в модулях різних типів даних, які мають спільні операції. Наприклад, якщо при реалізації функціонал одержання позицій різних об’єктів буде досягатись однаковими діями, які підтримуються їх базовими класами, то модуль прорахунку дистанції між двома об’єктами взагалі не потребуватиме роботи з конкретним типом даних. Такі можливості можна спостерігати в движку Unity, який дозволяє будь-яким компонентам об’єктів одержати позиції цих об’єктів:

```
public float GetDistance<T1, T2>(T1 object1, T2 object2) where T1 : Component where
T2:Component
        {
            return Vector3.Distance(object1.transform.position,
object2.transform.position);
        }
```

На останньому прикладі завдяки передбаченню двох параметрів метода як наслідників класу Component, дозволено використовувати в методі будь-які об’єкти компоненти для прорахунку дистанції.

Механізм абстрагування дозволить об’єднати деякі елементи функціоналу в один абстрактний інтерфейс, що може унаслідуватись багато разів. Наприклад,

модулі націлювання об’єкта за однією, двома, трьома осями можуть мати спільний інтерфейс, але сама реалізація їх роботи буде різнитись. Якщо елементи поведінки також передбачаються як наслідувачі спільного базового класу або інтерфейсу, то в них може використовуватись абстрактний інтерфейс прицілювання. А вже в конкретних реалізаціях елементів поведінки на його місце підставлятиметься і конкретна реалізація прицілювання.

Завдяки застосуванню описаних в даному пункті ідей для вдосконалення методу Агент-Поведінка, метод одержав такі особливості:

– в елементів поведінки одного типу може бути своя базова реалізація, що може бути побудована на базових модулях;

– різні агенти можуть використовувати різні реалізації одного типу елементів поведінки;

– різні елементи поведінки можуть використовувати різні реалізації свої базових модулів.

Схематичне зображення концепції вдосконалення методу продемонстровано на Рисунку 2.1.

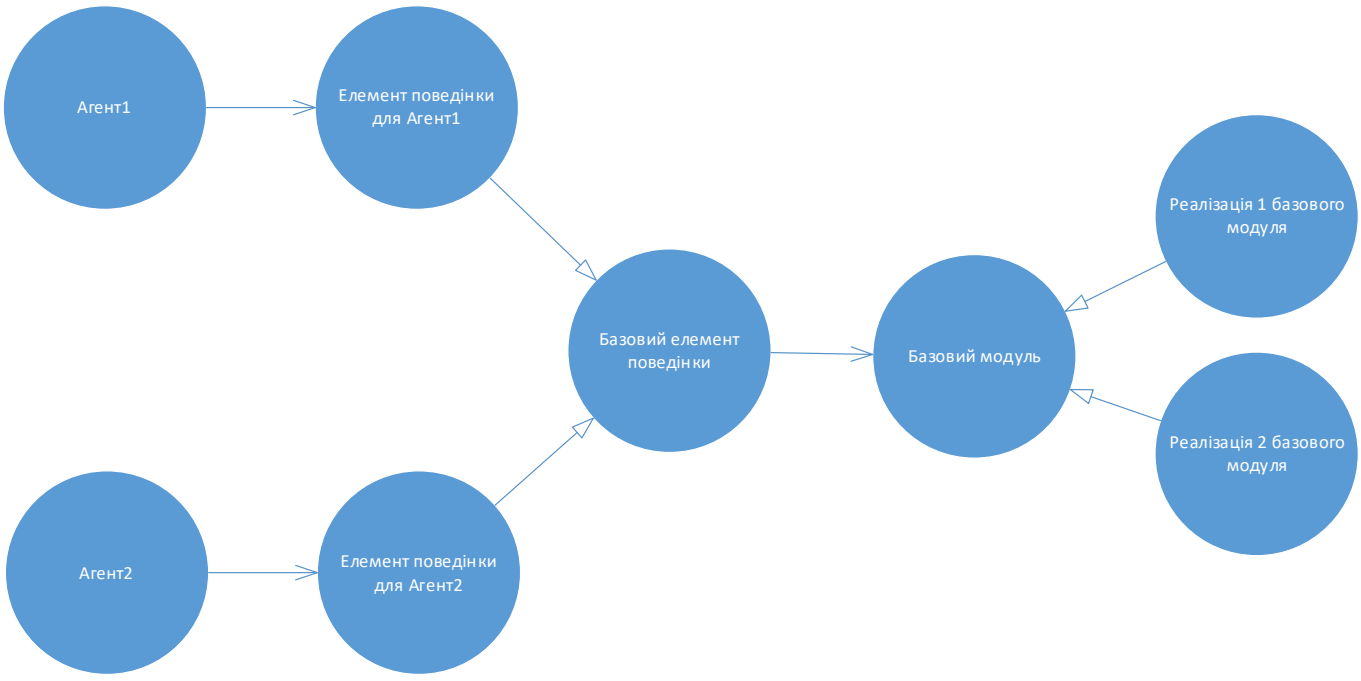


Рисунок 2.1 – Концепція вдосконалення методу Агент-Поведінка

Нажаль, дане вдосконалення також представлене рядом недоліків:

– послідовність роботи з модулями не може змінюватись в базовому елементі поведінки без зміни в конкретних. Таку можливість важко уявити без зміни поведінки базового класу і порушення LSP-принципу SOLID;

– в послідовність модулів не може бути нормально вставлений проміжний крок або проміжний модуль. Будь-які додавання модулів в існуючий потік роботи базового елементу поведінки буде важко зробити без порушення LSP-принципу SOLID;

– з послідовності модулів не може бути вилучений модуль. Єдиним варіантом є обробка NULL-посилань на модулі, але це неможливо задати явно, і сторонній розробник може не знати, що деякі модулі, передбачені архітектурою, відсутні;

Отже, основна проблема методу полягає в необхідності явно визначати процес симуляції поведінки і роботи з модулями, навіть якщо окремі елементи цього процесу можуть перевизначені. Подальше вдосконалення методу потребує переосмислення об’єктно-орієнтованої концепції моделювання таких систем.

Виходом для вирішення даної проблеми може слугувати повна децентралізація моделей ШІ і передача управління симуляцією самим модулям, які будуть визначати не тільки свою роботу, але й передачу роботи іншим модулям, від конкретної реалізації яких вони мало залежатимуть. Фактично, це означає, що об’єктно-орієнтований підхід, застосований в модулях, поєднується з алгоритмічним підходом, який полягає в використанні цих модулів як окремих кроків алгоритму.

2.3 Метод моделювання програмної системи

Для подальшого розвитку методу потрібно поєднати переваги двох концепцій моделювання ШІ – об’єктно-орієнтованої, та алгоритмічної.

На основі першої можна побачити принципи керування ігровими елементами, які опираються на декомпозицію всієї поведінки на окремі складові, що можуть замінятись для досягнення різних результатів. При достатньо деталізованому проектуванні з декомпозицією на велику кількість рівнів абстракції, можна ціною

сильного ускладнення моделей досягти легкої заміни їх елементів і адаптації для різних задач. Але через велику кількість різних ігрових персонажів, об’єктів, розробка такої системи може забрати велику кількість часу на саме проектування, яке постійно буде під ризиком, що зміни, які впровадяться в подальшому, не будуть передбачені моделями поведінки або не зможуть бути впроваджені якісно. В той час як алгоритмічні моделі, до яких можна віднести нейронні мережі, хоч і не потребують проектування гнучкої об’єктно-орієнтованої системи і абстраговані від об’єктів, за які вони прийматимуть рішення, але дуже громіздкі, монолітні, і не передбачають внесення змін.

Можна як і випадку з нейронними мережами створити модель, яка буде грати роль математичного алгоритму, віддаючи керування об’єктами на базі результатів його роботи проміжним сутностям-адаптерам. В той же час сама модель може бути не обов’язково монолітною, а її алгоритм може ділитись на певні частини, кожна з яких може бути з’єднана з іншими.

Щоб зробити ці частини повторно-використовуваними без повторного написання його окремих частин, з можливістю задання різних моделей, треба змінити звичний підхід до рішення алгоритмічної задачі.

Математичний алгоритм можна вважати функцією F, в якій одному набору вхідних даних можна покласти у відповідність свій набір вихідних:

$$Y = F(X)$$

Де Y – множина вихідних даних, а X – множина вхідних. При потребі можна розбити алгоритм на окремі кроки F_i. В такому випадку буде одержана рекурсивна модель такого вигляду:

$$Y = F_n(F_{n-1}),$$

де n – кількість окремих кроків в алгоритмі, а результат на кожному окремому кроці є:

$$F_i = F_{i-1}(F_{-2}), i > 2$$

Якщо для всіх $F_i(X)$ множина X буде являти собою сталий набір даних одного типу, то можна міняти місцями функції в ряді будь-яким чином, породивши множину всіх можливих функцій F':

$$F'_{i,j} = F_i(F_j), \qquad i, j \leq n,$$

а алгоритм буде послідовністю функцій, множина яких є підмножиною цієї множини.

Якщо алгоритм розбивати на послідовні процедури, які різняться набором вхідних та вихідних даних, кожна функція буде залежна від набору даних попередньої. Але якщо алгоритм буде представлений однаковим набором вхідних і вихідних даних на кожному кроці, кількість кроків в цьому алгоритмі рівна n, то кількість перестановок цих кроків згідно формули кількості перестановок рівна:

$$P_n = n!$$

А якщо можна замінювати окремі функції, використовуючи повторно інші функції, то ця кількість згідно формули розміщень з повтореннями буде рівна:

$$A_n = n^n$$

Необхідність створювати як можна більшу кількість перестановок продиктована тим, що , при розробці штучного інтелекту для таких проектів, як комп'ютерні ігри, доводиться створювати велику кількість різних варіацій поведінки об'єктів. Одну і ту ж задачу симуляції діяльності кожного окремого об'єкта доводиться вирішувати багато разів. Часто в двох рішеннях задачі може бути один, декілька, або навіть більше половини схожих кроків, які не можуть бути розроблені в одному рішенні і повторно використані в другому. Тому часто буває, що задача реалізовується повторно, але зі своєю специфікою.

Якщо ж розробляти рішення будь-яких задач або хоча б окремих частин задач як набір функцій з однаковими наборами вхідних-вихідних даних, можна добитись великої гнучкості в реалізації цих задач і повторно активувати вже реалізовані кроки, які використані в будь-якій з них.

Ефективність даного рішення можна розглянути за допомогою наступного представленого псевдокодом алгоритму:

```
Ввести a, b, c;
x = a + b;
y = a * x;
n = a – c * y;
Вивести n.
```

Першим кроком є визначення всієї програми як глобальної функції:

```
F(a, b, c)
Початок
x = a + b;
y = a * x;
n = a – c * y;
Повернути n.
Кінець
```

Вигляд функції, якщо її алгоритм функції на окремі під-функції, називаючи їх за типом операцій, які стоять в них:

```
F(a, b, c)
Початок
x = Sum(a, b);
y = Multiply(a, x);
n = Subtract(a, Multiply(c, y));
Повернути n.
Кінець
```

Тут всі конкретні операції замінені на процедури, що представляють ці операції, зберігаючи їх в окремому словнику функцій:

Sum(x1, x2) = x1 + x2

Multiply(x1, x2) = x1 * x2

Subtract(x1, x2) = x1 – x2

Тепер можна перевизначити функцію F таким чином:

F(a, b, c)

Початок

Повернути Subtract(a, Multiply(b, Multiply(a, Sum(a, b))))

Кінець

Тепер якщо вхідні параметри a, b, c представити як функції для одержання вхідних даних A(), B(), C(), без набору вхідних параметрів як в визначеному словнику функцій, але з тим же набором вихідних, то вони будуть односторонньо-сумісні з усіма іншими функціями, тобто, не зможуть бути використані як проміжні елементи, але зможуть бути використані як початкові. Таким чином формується вузлова модель використання під-функцій функції F (Рисунок 2.2).

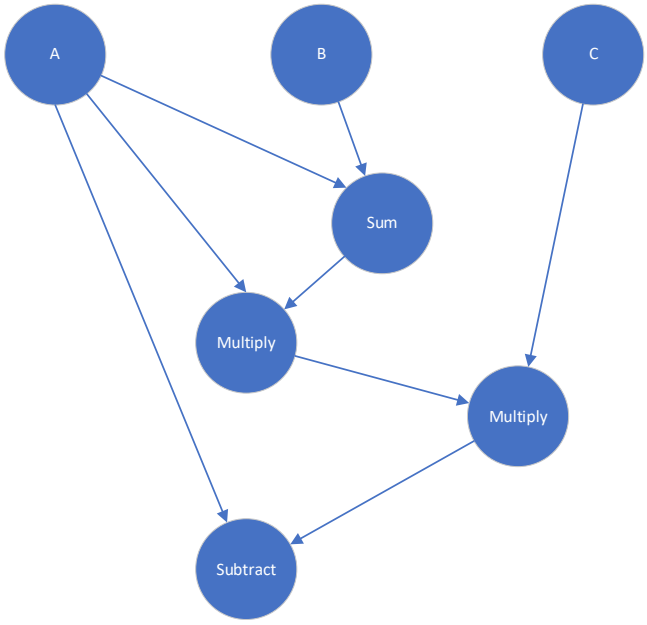


Рисунок 2.2 – Модель використання підфункцій функції F

Це є тим самим алгоритмом, але тепер він представлений у вигляді наочної візуальної моделі з різних вузлів, в якої є словник функцій – A, B, C, Sum, Multiply, Subtract. Тепер при потребі можна доповнювати як сам словник, так і додавати, видаляти або модифікувати окремі вузли в ньому.

Даний підхід в побудові алгоритму бере свій початок з функційного програмування, де всі елементи – це функції. Але різниця полягає в способі самої побудови алгоритму, який не спирається на написання покрокових операцій, а на створення вузлів, що поєднуються з іншими вузлами для обробки результатів їх роботи.

Словник моделі можна доповнювати безліччю функцій з різним набором вхідних параметрів, але чим більше функцій з одним набором вхідних параметрів співпадають з як можна більшою кількістю функцій з цим самим набором вихідних, тим більший показник сумісності вузлів в системі CN_{max}. Якщо вважати, що існує ряд наборів вхідних параметрів кількістю n, то показником сумісності функцій в системі є сума всіх можливих односторонніх з'єднань різних типів функцій:

$$CN_{max} = \sum_{i=1}^{n} I_i * O_i \ ,$$

Де I_i – кількість всіх функцій з і-им набором вхідних параметрів$_i$, а O_i – кількість всіх функцій з і-им набором вихідних параметрів. Це дозволить встановити число всіх можливих з'єднань між вузлами різного типу.

Вплив значення максимального показника сумісності системи на гнучкості в моделюванні алгоритму рішення задачі можна продемонструвати на алгоритмі, в якому є більша залежність попередніх кроків від наступних. Як приклад, можна створити підпрограму P_1, яка складається з таких кроків:

P_1(T, B, C):

Початок

Крок1. $R_1 = F_1$(T, B)

Крок2. $(R_2, C_2) = F_2(R_1, C)$

Крок3. $R_3 = F_3(R_1)$

Крок4. $C_4 = F_4(R_2, C_2)$

Кінець

В даному прикладі представлена функція, яка розбита на кроки, що мають різні набори вхідних параметрів і повертають різні набори вихідних. Для наочності різні типи даних позначені різними літерами, а літерою F – функції. Але важливим є не те, що вона робить, а те, скільки може бути побудовано пар (F_i, F_j), в якій F_j може використовувати в якості вхідних параметрів вихідні параметри F_i. Кількість цих пар і є значенням CN_{max}.

Для цього потрібно класифікувати функції в алгоритмі за форматами вхідних та вихідних параметрів (Таблиця 2.1).

Таблиця 2.1 – Класифікація функцій підпрограми P_1 за форматами вхідних та вихідних даних

i	Формат параметрів	Функції з таким форматом вхідних параметрів	Функції з таким форматом вихідних параметрів	$I_i * O_i$
1	(R)	F_3	F_1, F_3	2
2	(R, C)	F_2, F_4	F_2	2
3	(C)	–	F_4	0
4	(T, B)	F_1	–	0

Звідси, $CS_{max} = 2 + 2 + 0 + 0 = 4$. Це означає, що алгоритм можна доповнити 4 різними способами, які легко виявити на основі даних цієї таблиці:

1) на базі результатів обчислень на кроці 1 можна знайти значення функції F_3;
2) на базі результатів обчислень на кроці 3 можна знайти значення функції F_3;
3) на базі результатів обчислень на кроці 2 можна знайти значення функції F_2;

4) на базі результатів обчислень на кроці 2 можна знайти значення функції F_4.

Якщо представити алгоритм функції F на вузловій діаграмі (Рисунок 2.3), то можна зобразити, як саме можна доповнити її 4 додатковими вузлами, як це зображено на Рисунку 2.4. Два жовті вузли – це способи додаткового розширення алгоритму, а два зелені – вузли, які вже використані.

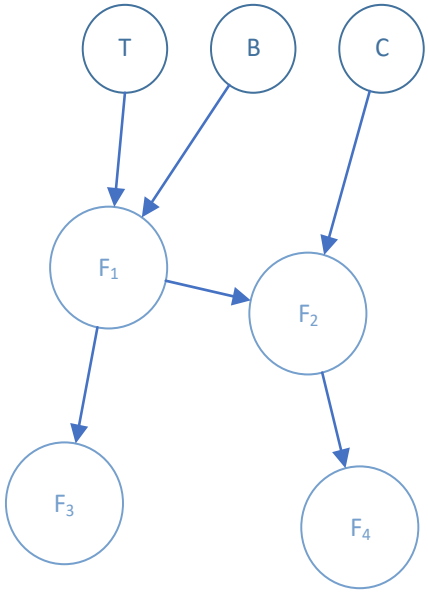


Рисунок 2.3 – Вузлова діаграма алгоритму підпрограми P_1

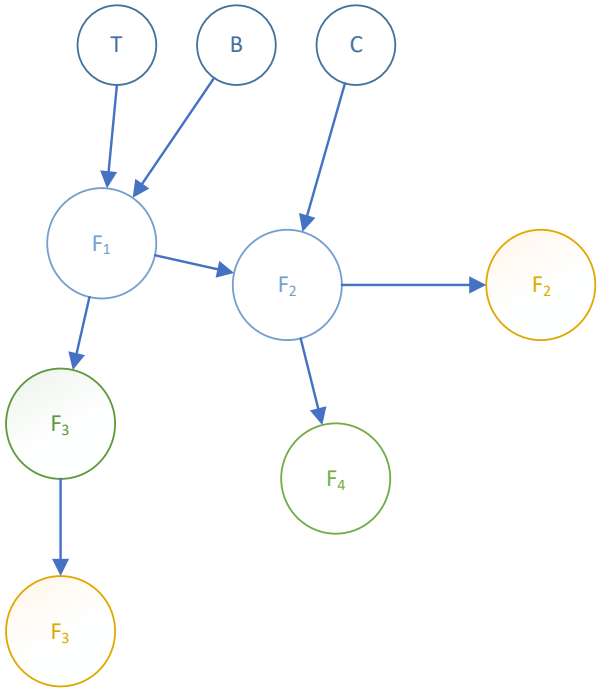


Рисунок 2.4 – Способи розширення алгоритму підпрограми P_1 на вузловій діаграмі

Це означає, що при моделюванні функцій на вузловій діаграмі цей алгоритм можна розширити всього 2 рази. Але кількість можливих модифікацій значно зросте, якщо буде більше пар функцій, в який формат вихідних даних відповідатиме формату вхідних. Так, якщо змоделювати алгоритм P_2, в якому функції будуть мати однаковий формат вхідних і вихідних даних, то можна досягти високо значення CN_{max} і таким чином дозволити розширювати вузлову модель алгоритму багатьма способами.

$P_2(X, Y)$
Початок
$(X_1, Y_1) = F_1(X, Y)$
$(X_2, Y_2) = F_2(X_1, Y_1)$
$(X_3, Y_3) = F_3(X_2, Y_2)$
Кінець

Таблиця 2.2 демонструє, що використавши один формат вхідних/вихідних даних в функціях, можна досягти великого значення CN_{max} і зробити систему максимально розширюваною. Можливі розширення системи продемонстровані на Рисунку 2.5.

Таблиця 2.2 – Класифікація функцій підпрограми P_2 за форматами вхідних та вихідних даних

i	Формат параметрів	Функції з таким форматом вхідних параметрів	Функції з таким форматом вихідних параметрів	$I_i * O_i$
1	(X, Y)	F_1, F_2, F_3	F_1, F_2, F_3	9

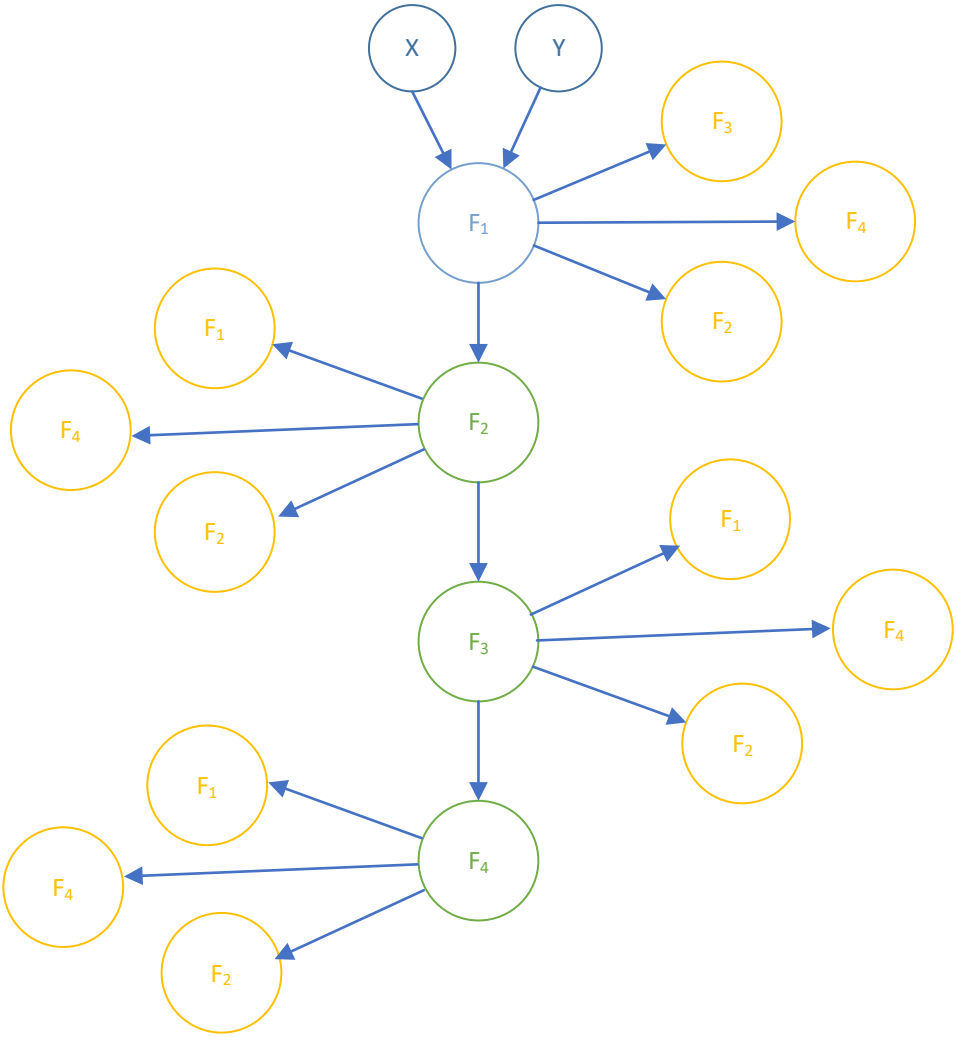


Рисунок 2.5 – Способи розширення алгоритму підпрограми P_2 на вузловій діаграмі

Порівнюючи можливості розширення алгоритмів P_1 та P_2, стає зрозуміло, що незважаючи на меншу кількість реалізованих функцій в P_2, її розширюваність в вузловій моделі значно вища.

Ще один важливий елемент при внесенні змін в вузлову модель є можливість замінити функціональний вузол одного типу функціональним вузлом іншого типу. Це можливо зробити тільки при одній умові – якщо вузол відповідає попередньому за вхідними та вихідними параметрами. Це дозволяє внести новий показник словника моделі, який оцінює можливість замінити окремі її вузли – RN_{max}. А загальна кількість можливих внесень змін в вузлову модель FN_{max} обчислюється як сума показників сумісності вузлів та їх змінюваності:

$$FN_{max} = CN_{max} + RN_{max}$$

Показані вище приклади демонструють, що для кращої розширюваності система повинна орієнтуватись на використання шаблонних модулів в якості вузлів, які можна з’єднувати при відповідності типів даних, якими заміщуються шаблони. Але це і ускладнює сам підхід до моделювання логіки – тепер кожен набір шаблонних модулів з однаковим типом даних дозволить створювати моделі, які є більш сфокусованими на обробці саме цього даних. Такі умови обмежують в можливості опрацювати будь-які дані на будь-якому кроці алгоритму, що властиво імперативному підходу в програмуванні. Але в той же час це дозволить абстрагуватись від внутрішніх аспектів реалізацій вузлових функцій і будувати алгоритм не як покрокову програму, а як візуальну модель, що не потребує програмування зовсім. За рахунок цього можна:

– позбутись повторюваності коду за рахунок можливості використати його в будь-якому місці моделі, де цей код може обробляти ті самі дані і повертати той самий результат;

– легко реалізовувати будь-який аспект алгоритму поведінки штучного інтелекту і ускладнювати його новими факторами, просто доповнюючи його вузлами;

– реалізувати візуальні редактори моделей поведінки штучного інтелекту за рахунок абстрагування від процесу написання коду і фокусування на способах передачі та обробки даних.

2.4 Аналіз вузлового методу, порівняння з іншими методами розробки ШІ

Завдяки методу вузлового моделювання знижена необхідність в написанні програмного коду. Як відомо, майже вся робота програміста полягає в написанні програмного коду на низькому, детальному рівні, коли доводиться працювати напряму з математичними операціями, змінними, сутностями в алгоритмі. В пункті 2.3 було продемонстровано, що при побудові вузлової моделі можна абстрагуватись від написання самого коду, будуючи візуальну модель з’єднаних вузлів, кожен з яких представляє свою функцію. А якщо проектувати вузли або систему, що їх реалізує,

таким чином, щоб як можна більше вузлів були сумісними і здатними до з’єднань, то можна при наявності достатньої кількості вузлів різних типів дуже швидко проектувати алгоритми і різні варіанти його реалізації. Саме швидкість проектування стає однією з основних переваг при відмові від написання коду, адже розробники не повинні звертати увагу на деталі і написання коду.

Також була досягнута висока гнучкість системи. В пункті 2.3 було продемонстровано, як через шаблонність вузлів і використання одного типу даних можна перебудовувати систему на будь-який лад і вносити будь-які зміни в ній. За рахунок високо значення FN_{max} в будь-який момент можна без ускладнень і необхідності переробляти весь алгоритм чи вносити зміни в будь-яку його частину, як додаючи нові модулі, так і замінюючи існуючі.

Другий з фундаментальних принципів об’єктно-орієнтованого проектування SOLID називається Open Closed Principle. Принцип полягає в наступному: «Програмні сутності (класи, модулі, функції) повинні бути відкритими для розширення і закритими для модифікації» [16]. При рішенні задач зі штучного інтелекту, особливо при його модифікації протягом життєвого циклу проекту, неодноразово в модулі його роботи можуть вносити зміни, які важко було передбачити на початку його розробки. Досвід ігрової розробки показує, що штучний інтелект, як і більшість ігрових елементів, сильно залежать від користувацького досвіду і для його покращення часто відхиляються від початкової моделі. Через постійні модифікації і нестачу виділеного часу принцип Open Closed часто порушується і приноситься в жертву заради швидшої реалізації і демонстрації ігрових прототипів, що призводить до постійних модифікацій внутрішньої логіки сутностей при відсутності належного набору абстракцій для створення різних реалізацій на їх основі. Це може збільшити кількість помилок в реалізації ШІ, наростити багато непотрібних залежностей і обмежень, зрештою, позбавляючи можливості для внесення змін і адаптації нових механік.

Завдяки гнучкості вузлового підходу дотримання принципів SOLID набагато простіше, оскільки кожен вузол є завершеною частиною функціональності і не

призначається для модифікацій, а модифікації здійснюються замінами і додаванням таких вузлів.

Ще однією перевагою є можливість повторного використання реалізованого функціоналу. На відміну від звичних сутнісних об’єктно-орієнтованих підходів, не потрібно створювати нові тип сутностей при наявності великої різниці в очікуваній від них роботі з існуючими сутностями. Кожен вузол може бути використаний повторно і безмежну кількість разів в безмежній кількості моделей.

Навіть модель Агент-Поведінка часто не допускає повторного використання елементів поведінки і змушує або проводити глибокий рефакторинг і декомпозицію цих елементів і будування їх з окремих сутностей, або написання елементів поведінки знову, через що виникає ризик порушення принципу DRY – Don’t Repeat Yourself (не повторюйся). Часто окремі блоки функціоналу можуть бути повторно написані для роботи з іншими сутностями і типами даних, що хоч може і не бути прямим порушенням принципу, але при зміні загального погляду на реалізацію такого функціоналу змусить змінити його всюди в програмі. Наприклад, можна реалізувати один спосіб, яким окремі ігрові об’єкти, такі як бронетехніка, літак чи солдат оцінюють дистанцію до ворога. Але якщо кожен з них буде робити це не одним і тим же блоком функціональності, а продублює його з використанням своїх особливостей роботи, то зміна способу оцінки дистанції змусить розробника змінити його реалізацію декілька разів, що також спричиняє ризик забути замінити його в одному з місць і зробити помилку.

При використанні вузлового підходу можна створити велику кількість окремих функцій, які можуть бути використані повторно і вставлені в будь-якому місці моделі. При цьому можна нівелювати залежність від окремих типів сутностей, з якими працюватиме код, якщо вузлова модель буде працювати як шаблонне рішення, спільне для будь-яких типів даних. Через це дана перевага актуальна не тільки в межах одного проекту. Вузлові функції можуть бути використані повторно і в інших проектах, якщо будуть максимально сфокусовані на вирішенні типових задач. Це дозволяє створювати бібліотеки таких вузлів і ділитись ними з іншими розробниками.

Четверта перевага – надійність. Повертаючись до принципу Open Closed, потрібно розуміти, в чому одне з його основних покликань. Будь-які модифікації коду можуть спричини непередбачувані помилки, що ламають систему. Програмний алгоритм в таких гнучких системах, як ігровий ШІ, може змінюватись дуже часто і з високою вірогідністю розробникам доведеться працювати з постійним виправленням багів системи і її рефакторингом.

Якщо кожен вузол програмується окремо, з ціллю повторного використання, і не містить в собі значного функціоналу, слідкувати за його якістю набагато простіше, ніж за великими сутнісними об’єктно-орієнтованими системами. При цьому гарантується, що вирішення проблеми в одному з вузлів вирішує її всюди, де він використаний.

Окрім усіх перерахованих переваг, в вузлового методу є свої недоліки. Можна проаналізувати їх вплив на процес розробки, оцінити можливі ризики і знайти компромісні рішення.

Серйозним недоліком методу є потенційна обмеженість моделей одним форматом параметрів. Так, використання одного формату даних, що обробляються на вузлах, є не тільки перевагою системи, яка робить вузли сумісними, а модель – легко-змінюваною. Використання одного формату параметрів, яким може бути як ціла сутність, так і звичайне дійсне число, обмежує можливість доповнити моделі залежністю від інших типів та структур даних, які доведеться конвертувати в правильний формат або не використовувати в моделі взагалі. Втім, якщо не моделювати всю поведінку ШІ однією моделлю, а програмувати саме конкретні її частини, пов’язані з обробкою даних схожого формату, це сильно зменшить вплив цієї проблеми.

Також присутній сам факт переосмислення підходу у вирішенні задач. Розробнику моделі доведеться працювати зі словником функцій, які треба дослідити і зрозуміти. Фактично, незважаючи на знижену роль програмування при побудові моделі, доведеться займатись більш високорівневим програмуванням, де окремі оператори і є функціональними вузлами.

Окрім того, існує постійна необхідність в існуванні достатнього словника функцій, щоб не мати необхідності в їх програмуванні на етапі створення моделей. В іншому випадку, перевага, що полягає в зниженні ролі програмування при реалізації моделі, нівелюється. Адже по мірі впровадження моделі доведеться і впроваджувати нові функції, що нею можуть бути використані. Однак, після того як ці функції з’являються, вони можуть бути використані повторно в будь-який момент часу.

2.5 Висновки

В даному розділі було проаналізовано існуючі архітектурні патерни і їх застосовність в контексті побудови штучного інтелекту. Обрано можливі варіанти застосування архітектурних рішень, які можуть допомогти вирішити проблеми старих методів.

Також даному розділі було проведено дослідження, яке показало:

1) алгоритми можна характеризувати за гнучкістю, що дозволяє оцінити можливість заміни або повторного використання окремих його кроків;

2) чим більше кроків алгоритму сумісні за форматом вхідних-вихідних даних, тим більша гнучкість алгоритму;

3) більш гнучка побудова алгоритму краще підходить для ігрового ШІ, оскільки для цього доводиться розробляти велику кількість моделей поведінки, що мають багато спільних елементів;

4) вузловий підхід дозволяє візуально моделювати алгоритми, а моделі на його основі підвищують гнучкість алгоритму.

3 АЛГОРИТМИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ

3.1 Алгоритми вирішення задачі

В розділі 2 було розроблено концепцію вузлової моделі для моделювання складної поведінки ігрового штучного інтелекту. Але наступною проблемою є алгоритм роботи цієї вузлової моделі, і як саме вона буде реалізована. Є перелік проблем, які належить вирішити при реалізації:

- спосіб представлення кожного вузла в алгоритмі;
- розрахунки всередині вузла;
- спосіб з’єднання вузлів;
- повний обрахунок всіх результатів роботи моделі;

Кожен вузол може бути реалізований як об’єкт з точки зору ООП. Тоді кожен вузол – це модуль, який має свій стан і обрахунок, і може наслідувати свою функціональність з більш примітивних модулів та реалізовувати інтерфейси абстрактного рівня архітектури. Це зробить модуль максимально відкритим для розширення, конфігурації і взаємодії з іншими модулями.

Ще одна перевага об’єктно-орієнтованого підходу полягає в тому, що модуль може мати багато різних методів та полів, які реалізовуватимуть його стан і змінюватимуть його без необхідності реалізовувати цю логіку ззовні. Окрема функція модуля зможе займатись прорахунком і збереженням результату обчислень, який може бути використаний повторно, і тільки тоді, коли цей результат знадобиться.

В якості частини стану модуля, його атрибутів, можуть бути посилання на інші модулі, з якими треба з’єднуватись, одержувати їх результати роботи, і працювати далі з ними так, як це передбачено логікою самого модуля. Тому за рахунок полів та конструкторів можна бути з’єднувати модуль з іншими модулями.

Найголовнішим завданням є алгоритм роботи моделі, що побудована на цих модулях. Тут варто зазначити, що сама вузлова модель використовує принцип, який реалізований в функційному програмуванні, адже її структура складається з функцій.

Цю схожість можна помітити, розглянувши псевдокод прикладу функції F, згаданої в попередньому розділі:

```
F(a, b, c)
Початок
x = a + b;
y = a * x;
n = a – c * y;
Повернути n.
Кінець
```

Його можна привести до такого вигляду:

```
F(a, b, c)
Початок
Повернути Subtract(a, Multiply(b, Multiply(a, Sum(a, b))))
Кінець
```

Варто зазначити, як саме працює порядок виклику функцій-кроків при такому вигляді задачі:

1) віднімання Subtract(), яка в порядку кроків алгоритму слідує останньою, але викликається першою;

2) для того щоб виконатись, функція Subtract() потребує параметрів, другий з яких є результатом виконання Multiply(c, y);

3) в даного кроку Multiply є параметр y, який має на даному етапі бути обчисленим як результат виклику Multiply(a, x);

4) попередній крок потребує виклику Sum(a, b).

Хоча сам порядок обчислень не змінився і зберіг свою послідовність, порядок запуску цих обчислень починається саме з кінцевого кроку. Саме той крок, який відповідає за результат, і визначає подальші обчислення.

Запуск обчислень на потрібному модулі можна буде реалізувати методом Extract(). Цей модуль для своєї роботи буде викликати Extract(), щоб одержати результати роботи тих модулів, які слідують перед ними, і опрацювати їх. Ті зроблять в свою чергу те ж саме, тільки з модулями, якими вони пов’язані. В результаті одержиться модель, в якій спроба вивільнити результати роботи одного з модулів викличе ланцюжок викликів Extract() усіх модулів, які беруть участь в обчисленні даного результату (Рисунок 3.1).

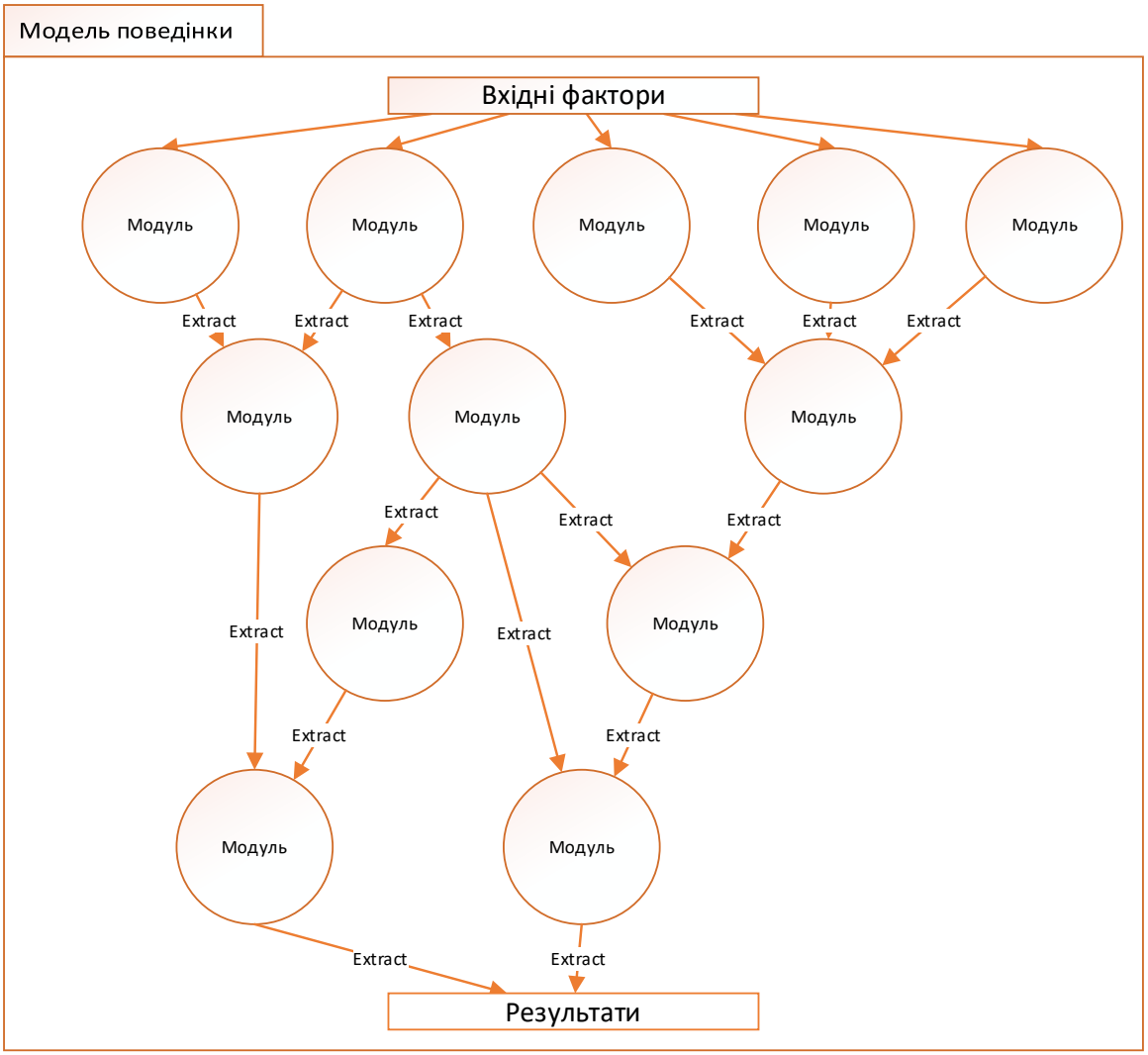


Рисунок 3.1 – Алгоритм роботи модульної вузлової моделі

Однією з важливих рис такого алгоритму роботи, полягає в так званому «лінивому» обчисленні результатів. Ті модулі, які не містяться в ланці обчислень модуля-результату, не будуть використані і не проведуть ніяких операцій.

Обчислення в модулі будуть проводитись лише тоді, коли буде викликано функцію Extract() або ззовні, або з іншого модуля. Така особливість також успадковується з функційного програмування.

3.2 Визначення вимог до програмної системи

Щоб програмна система для моделювання складної поведінки ШІ і її реалізація була корисною на практиці, потрібно визначити ряд вимог, які дозволять максимально реалізувати переваги вузлових моделей, описані раніше.

– модулі кожної окремої моделі повинні бути сфокусовані на одному типі даних;

– передбачити шаблонність, можливість використовувати будь-який тип даних для кожної моделі;

– потрібно реалізувати можливість використати будь-які функції для обробки цих даних;

– потрібно передбачити можливість створення модулів, з’єднання модулів і виклику їх методів;

– модулі можуть з’єднуватись з одним, двома, і нескінченною кількістю інших модулів;

– модулі повинні бути розбиті на декілька рівнів ієрархії для того, щоб можна було використовувати не тільки кінцеві класи, але і їх базові елементи, і при цьому підтримувати спільний інтерфейс роботи з усіма модулями на різних рівнях;

– потрібно передбачити можливість ініціалізації модулів зовнішніми функціями обчислень, які можуть бути як частиною якогось словника функцій, так і функціями, створеними розробниками моделі;

– реалізація модулів повинна відповідати принципам SOLID.

– кожен модуль повинен надавати можливість одержувати результат обчислень на ньому, не викликаючи обрахунки на модулях, які не є частиною алгоритму одержання цього результату;

– в цілях оптимізації модулі повинні кешувати результати своїх обчислень, і при потребі позбуватись цього кешу для повторення обчислень.

3.3 Проектування програмної системи

Наступним завданням є проектування програмної системи модулів, з допомогою якої можна буде моделювати поведінку AI.

Нехай існують вхідні дані шаблонного типу T, які потрібно передати системі, обробити їх нею, та отримати вихідні дані цього ж типу. Обробка ж і перетворення даних залишається чорним ящиком (Рисунок 3.2).

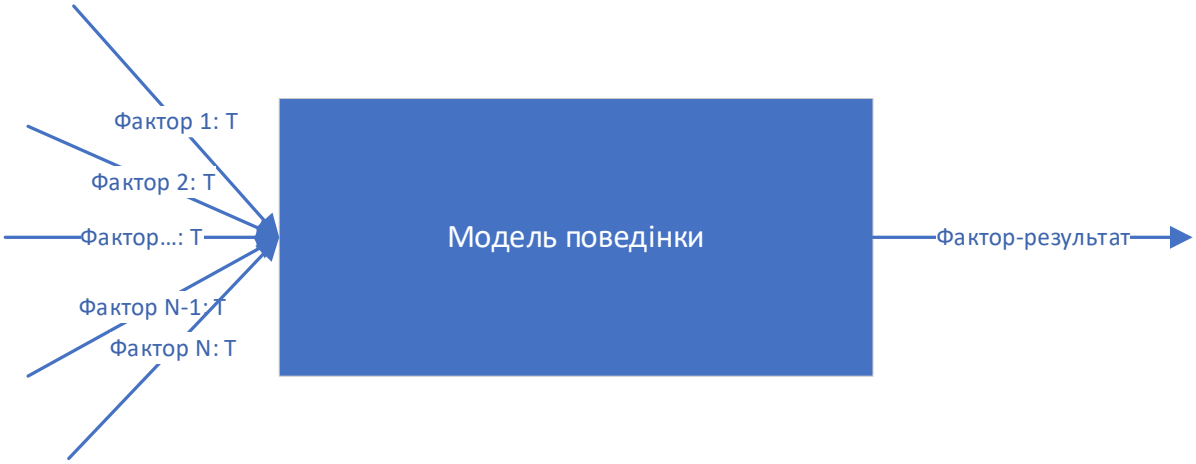


Рисунок 3.2 – Модель чорного ящику системи з вхідними та вихідними параметрами

Тип вхідних та вихідних значень є шаблонним, що значить, що на його місці може бути будь-який тип. Це дозволить сприймати систему як виключно архітектурне рішення, не прив’язане до конкретних типів даних, які будуть використані в створенні та реалізації конкретних моделей штучного інтелекту. З одного боку, це не дає можливості користуватись великою кількістю різноманітних типів даних та обробників цих даних при розробці однієї і тієї ж моделі поведінки, адже на всю модель може бути використаний лише один тип даних, який підставлений замість шаблонного. З іншого боку – для одного типу даних можна виробити досить великий

набір операцій, що будуть закладені в кожен модуль, а самі модулі можна з'єднувати в будь-якій послідовності, гарантуючи, що вони розумітимуть вхідні та вихідні параметри одне одного.

Для деталізації концепції буде визначено модуль, що буде виконувати запит на одержання вихідних даних типу T з моделі. Запит буде визначатись методом Extract() і буде повертати шукане значення. При цьому механізм прорахунку цих результатів залишається досі невідомим (Рисунок 3.3).

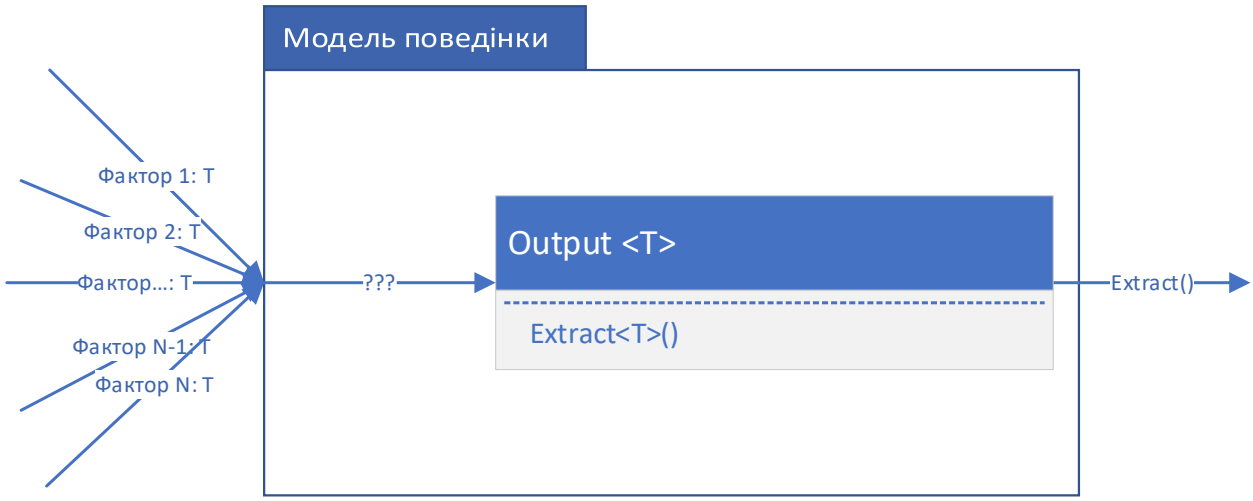


Рисунок 3.3 – Використання модуля вихідних даних

Важливим подальшим кроком є визначення механізму покрокового розрахунку кінцевих даних. Якщо останній модуль дозволяє одержати значення в ньому методом Extract(), то одержання даних цим модулем може досягатись використанням методу Extract() попереднього модуля. Це буде реалізацією архітектурного патерну Ланцюжок обов'язків.

Нехай потрібно одержати результати обрахунків системи, викликавши метод Extract() кінцевого модуля. Цей модуль може мати посилання на попередній модуль і зробити запит на одержання його вихідних даних. І такий ланцюжок викликів Extract() може продовжуватись необмежено (Рисунок 3.4).

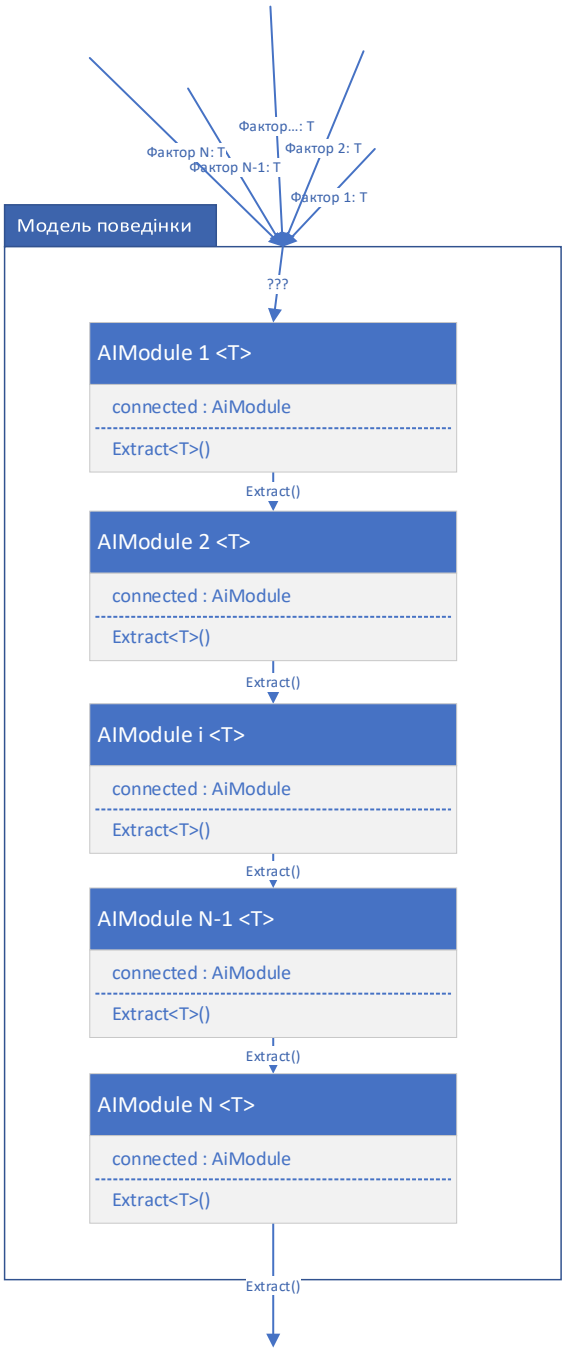


Рисунок 3.4 – Застосування патерну Ланцюжок Обов’язків

Можна підійти до рішення проблеми і з іншого боку, вказуючи всі модулі в одному списку, і активуючи механізм обходу цього списку, запам’ятовуванню цих даних і передачі наступному. Такий підхід є дещо централізованим, допомагає легко замінити модулі на інші на якійсь ланці, але в нього є критичні недоліки. Головний з них – повна лінійність всього потоку виконання. Кожен модуль може мати лише

попередника і наступника, і вихід за рамки цієї концепції на кшталт побудови дерево-подібної системи є неможливим.

Підхід з посиланням модулів на інші модулі має такі переваги:

– систему можна легко змінювати на ходу і навіть в процесі гри, без необхідності розробляти централізований механізм таких дій, а просто створюючи модуль і з'єднуючи з іншими модулями;

– при потребі можна не орієнтуватись на конкретний модуль як кінцевий, а використати дані, одержані на якомусь проміжному етапі. Розрахунки ж, що слідують далі, виконуватись не будуть;

– оскільки ініціатором виклику Extract() проміжних модулів будуть самі модулі, вони можуть при потребі не продовжувати цей ланцюг викликів, а використовувати вже попередньо розраховане значення.

Як вже говорилось раніше, механікою кожного окремого модуля є реалізація патерну Операція. Тобто кожен модуль буде представляти собою керовану дію над даними. Для того щоб ці дії виконати, модуль повинен містити в собі віртуальний метод, який можна назвати Compute(). Цей метод виконає своє завдання, а результати операцій збереже як об'єкт типу T. А далі можуть бути два способи реалізації конкретної механіки в цьому методі:

– створення спеціалізованого модуля з автономною і завершеною реалізацією його логіки та явним заданням типу оброблюваних даних, що наслідує базовий модуль і реалізовує метод Compute;

– реалізація шаблонного модуля, з можливістю передачі для його роботи зовнішніх шаблонних делегатів-операції, які можна задати під час створення цього модуля і замінити в ході виконання.

Втім, обидва способи можуть бути рівноцінні і задіяні для різних обрахунків в рамках однієї моделі ШІ. Це можна побачити на прикладі рішення однієї і тієї ж задачі різними способами. Нехай в якості шаблонного типу T в системі використовується трьовимірний вектор. А завданням модуля буде його нормалізація. Перший варіант призводить до створення модуля Vector3NormalizingModule, який буде наслідувати

AiModule, а метод Compute у ньому буде реалізовувати операцію нормалізації вхідного параметру (Рисунок 3.5).

Другий варіант означає створення похідного від AiModule типу MethodAiModule, а при створенні об’єктів цього типу передаватимуться делегати, які викликаються в методі Compute (Рисунок 3.6). На таких мовах програмування як C# типи делегатів часто відображені у вигляді класів, що реалізовує патерн Операція з контролем виклику делегата (наприклад, клас System.Func).

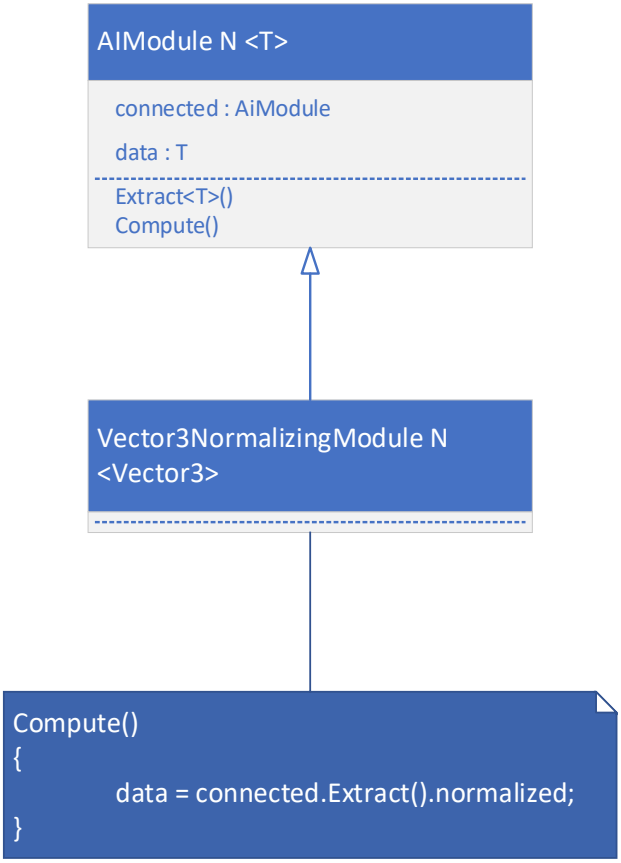


Рисунок 3.5 – Приклад реалізації модуля нормалізації вектора способом спеціалізованого модуля

Переваги способу з спеціалізованим модулем:

– тип модуля явно характеризує його обрахункову логіку;

– фокусування на своєму призначенні, з можливістю використовувати додаткові залежності та функціонал в обрахунках;

– можна кешувати та підтримувати більше даних в полях класу в ході виконання методу Compute.

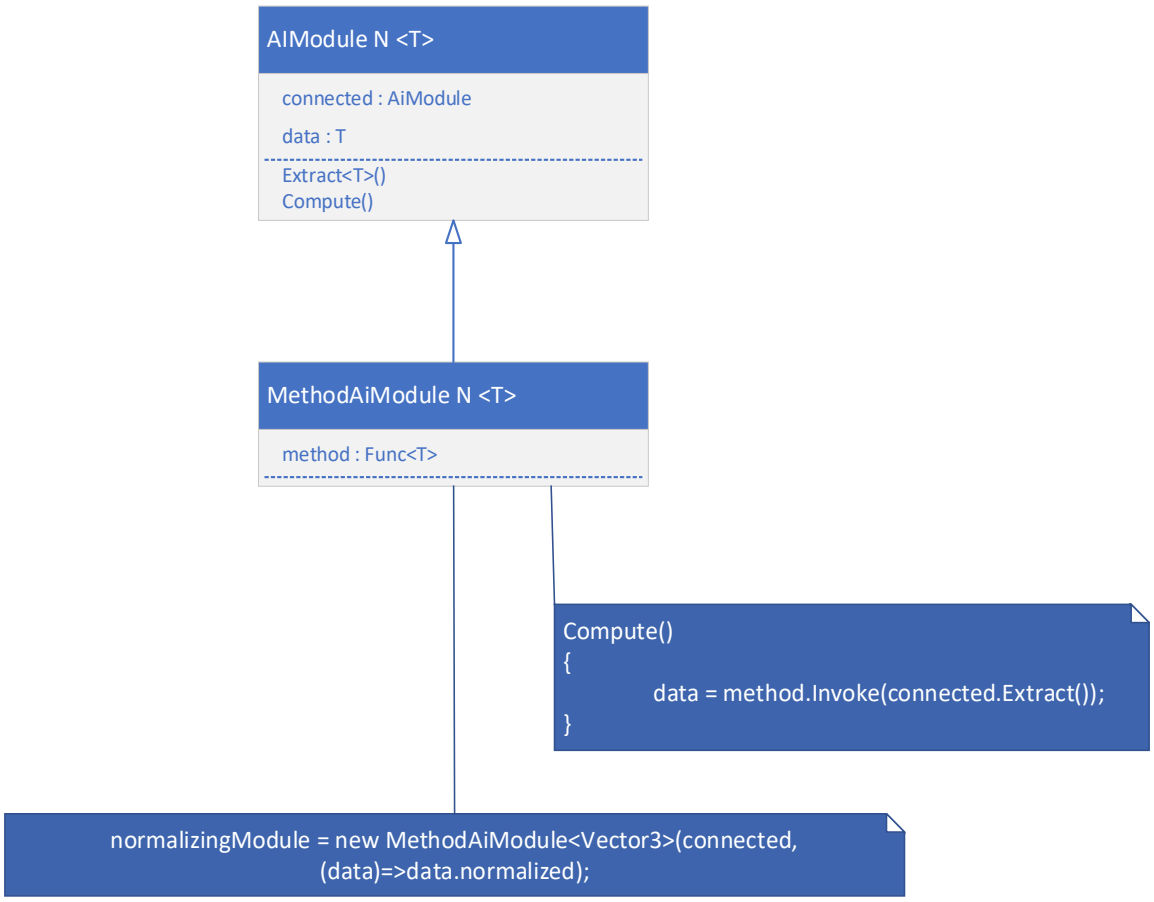


Рисунок 3.6 – Приклад реалізації модуля нормалізації вектора способом використання зовнішньої функції

Недоліки способу із спеціалізованим модулем:

– прив’язаність до типу даних;

– необхідність створювати похідний клас для будь-якої нової логіки обробки даних.

Переваги способу з шаблонним делегатом:

– відсутність прив’язки до конкретного типу даних в самих модулів;

– відсутність необхідності створювати велику кількість класів;

– можливість заміни операції модуля в процесі роботи програми;

– процес програмування логіки для модулів відділений від процесу програмування самих модулів. Можна створити набори статичних методів над конкретним типом даних і використовувати їх в модулях;

– можна створювати екземпляри модулів з використанням лямбда-виразів в якості операцій для модулів, без будь-якого створення окремих методів чи сутностей для цього.

Недоліки способу з шаблонним делегатом:

– труднощі з налагодження коду, оскільки важко оцінити, яка саме функція буде задіяна для будь-яких обрахунків в модулі;

– неможливо змінювати стан модуля в процесі виконання його операції, оскільки вона існує окремо від модуля.

Другий спосіб значно спрощує процес програмування, але змушує залишатись в певних рамках реалізації шаблонного модуля і не надає можливість розширювати його логіку. Але потрібно враховувати, що обидва способи є різними крайностями у вирішенні однієї і тієї ж проблеми. Можна використати певні елементи першого та другого способу, наприклад, наслідуючи клас MethodAiModule для того, щоб не тільки застосувати делегат для обробки даних, але і перед або після цього здійснити додаткові операції над цими даними.

Найголовніше, що зв’язки між модулями все ще залишаються абстрактними, тобто, дозволяють з’єднувати модуль з будь-якою іншою реалізацію AiModule. Можна об’єднувати в рамках однієї моделі будь-які модулі з будь-яким способом реалізації, оскільки їх завжди можна з’єднати між собою. Такий механізм доволі гнучкий, і до того ж дозволяє використовувати окремі модулі або одиниці функціоналу повторно в будь-яких моделях з однаковим типом даних, що підставляється замість шаблонного типу T.

Наступним кроком є реалізація механізму передачі вхідних даних в модель. Для цього можуть бути задіяні свої модулі, які будуть реалізовувати одержання цих даних. Але справа в тім, що наявна структура система сильно обмежує, оскільки передача даних відбувається в одному ланцюжку, і використання таких моделей не має

жодного сенсу при необхідності реалізації великої кількості факторів та вхідних даних. Тому потрібно імплементувати набір різних типів модулів, які будуть відрізнятись механізмом з'єднання з іншими модулями. В різних типів модулів повинні бути і різні зовнішні делегати операцій в модулях, оскільки якщо наприклад потрібно з'єднатись з двома модулями, то функція операції в модулі повинна приймати на вхід два параметри результати обчислень на цих модулях. Для подальшого вдосконалення системи потрібно передбачити наступні можливості:

– можливість з'єднати один модуль з іншим повинно бути реалізовано як окреме наслідування від базового AiModule похідного SingleConnectedAiModule – однозв'язного модуля (Рисунок 3.7). Призначення – повернути результати обчислень, проведених над даними, одержаними з інакшого модуля. Делегат буде обробляти один параметр функції;

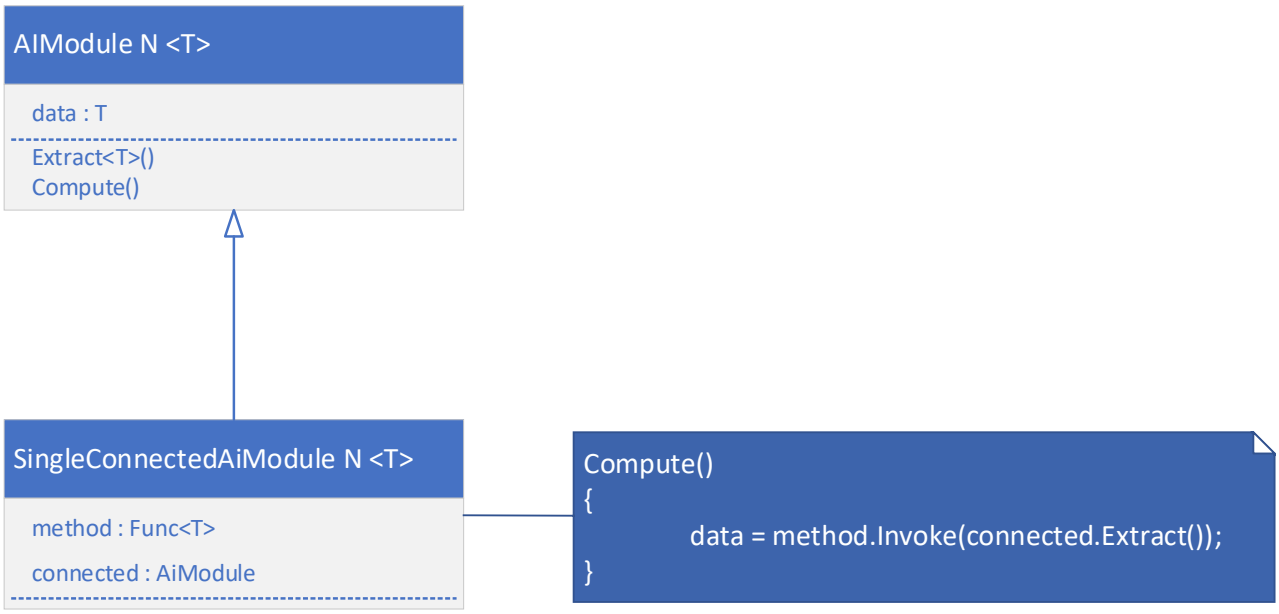


Рисунок 3.7 – Модель однозв'язного модуля

– реалізація модулів DoubleConnectedAiModule – двозв'язних модулів, здатних з'єднуватись з двома іншими модулями (Рисунок 3.8). Призначення – повертати результати обчислень, проведених над даними, що одержані з двох інших модулів. Делегат буде обробляти два параметри функцій;

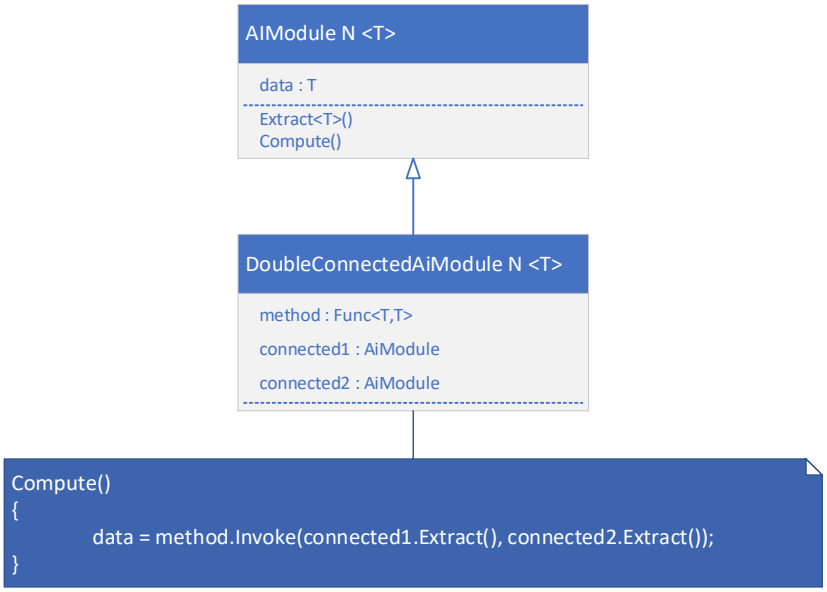


Рисунок 3.8 – Модель двозв’язного модуля

– реалізація модулів MutliConnectedAiModule – багатозв’язних модулів, які можуть з’днуватись з списком модулів, та спеціальним «модулем-критерієм». За допомогою делегата, що працює з списком параметрів та параметром-критерієм, інформація зі списку може бути оброблена або відібрана на базі критерію і повернута в вигляді окремого єдиного результату (Рисунок 3.9).

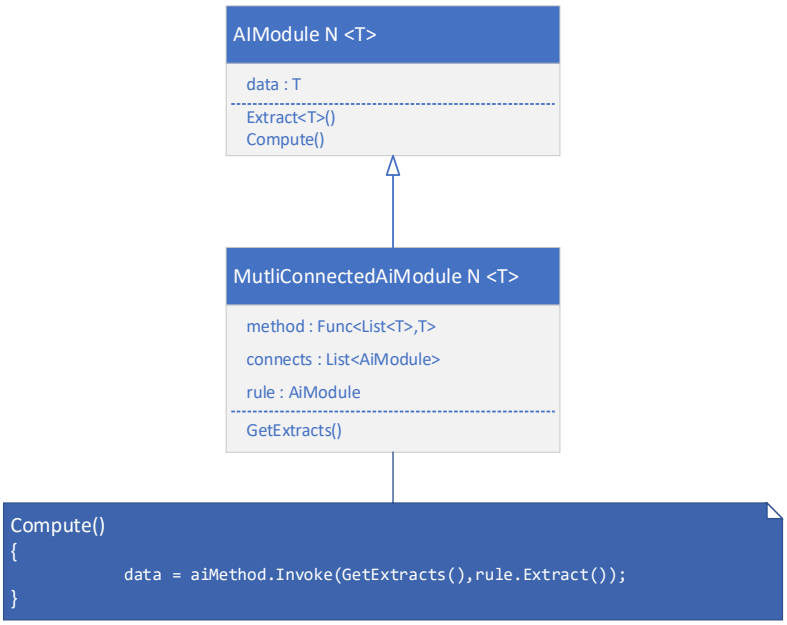


Рисунок 3.9 – Модель багатозв’язного модуля

Але наявні модулі можуть повертати лише один елемент-результат. Що якщо потрібно, щоб модулі могли не тільки обробляти дані з списку модулів, але й зберігати і повертати списки даних? Оптимальним рішенням буде розробити окремий глобальний тип модуля за аналогічною структурою – MultiAiModule, в якого метод Extract повертатиме список даних.

За типом зв'язування такий тип модулів буде поділятись на:

– SingleConnectedMultiAiModule – тип модулів, що з'єднується з будь-яким іншим мульти-модулем та обчислює свої результати з використанням модуля-критерія. Делегат повинен приймати список даних, та дані критерія (Рисунок 3.10);

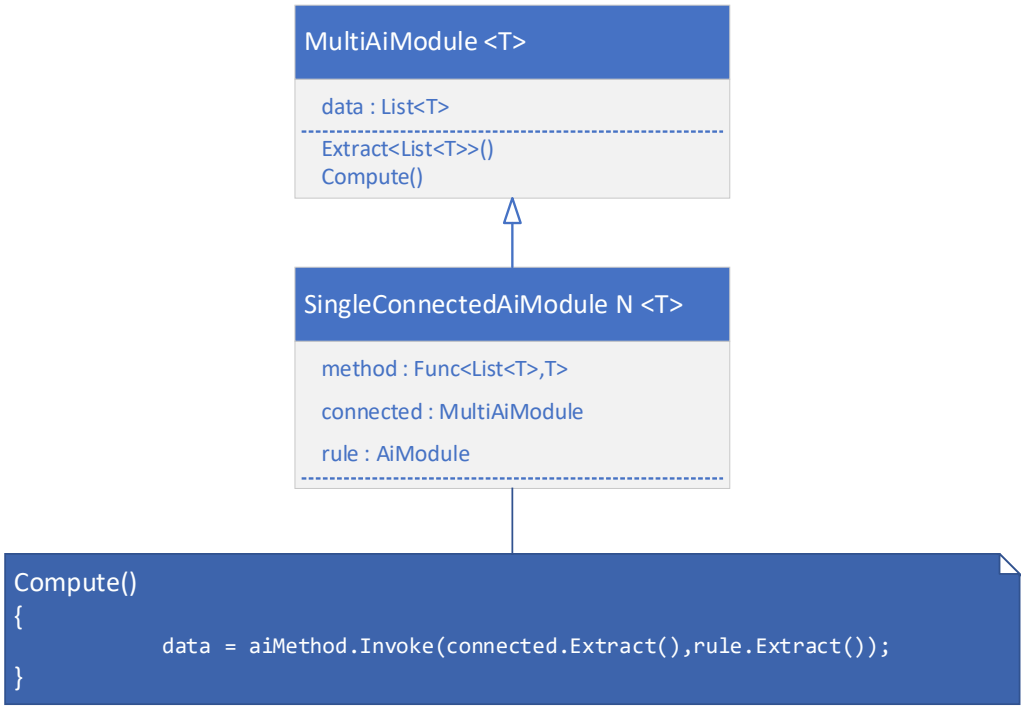


Рисунок 3.10 – Модель однозв'язного мульти-модуля

– MutliConnectedMultiAiModule – тип модулів, що з'єднується з списком модулів та обробляє дані на основі даних з модуля-критерія (Рисунок 3.11);

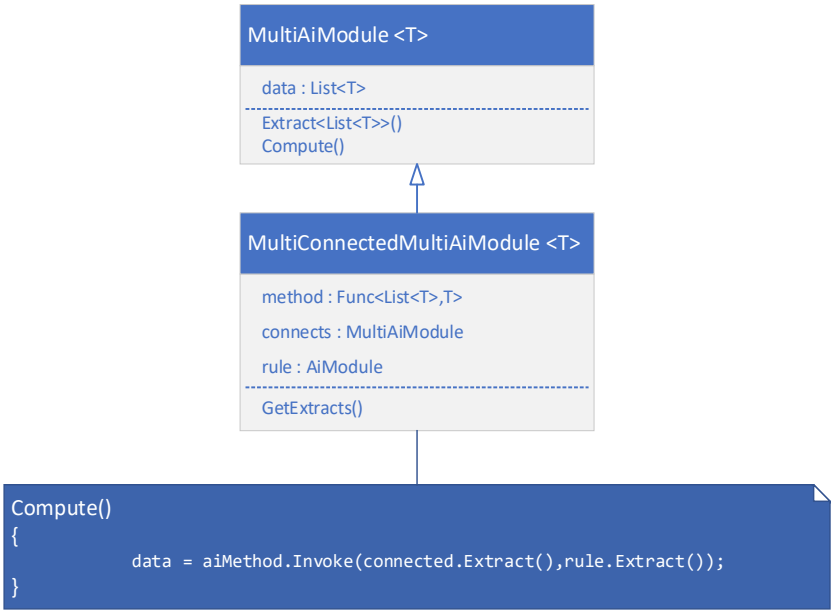


Рисунок 3.11 – Модель багатозв’язного мульти-модуля

Всі перераховані вище модулі є достатніми для створення значної кількості типів зв’язків та способів передачі даних, що належать якомусь шаблонного типу.

3.4 Оптимізація архітектури системи

Для покращення архітектурного гнучкості роботи з моделями, побудованими на описаних раніше модулях, можна винести функціонал модулів в інтерфейси, з побудовою зв’язків саме на інтерфейсних посиланнях. Таким чином можна буде вважати модулем будь-яких клас, який реалізовує потрібний інтерфейс, не обмежуючись лише наслідувачами AiModule або MultiAiModule (враховуючи, що в деяких мовах програмування множинне наслідування неможливе, але можливим є множинна реалізація інтерфейсів). Це означає, що якщо будь-який скрипт, клас чи сервіс є реалізатором інтерфейсу, і його можна долучити до моделей поведінки на рівні з усіма іншими модулями.

Найфундаментальніша задача кожного модуля – кешування даних. Кешуванням називаються процес, спрямований на збереження результатів виконаних операцій для їх повторного використання. В кожному типі модуля заздалегідь передбачено кешування даних в методі Compute(), перед тим як метод Extract()

поверне їх. Але очищення кешу модуля не передбачено системою, що змушує або не користуватись кешованими даними при кожному новому виклику Extract(), або ж позбуватись їх ззовні, що часто буває не дуже зручно і має бути запечатано в окремому методі. Саме цей метод і буде передбачатись найфундаментальнішим інтерфейсом системи: ICacheable.

В ньому буде метод Uncache(), який в якості параметру приймати булівське значення, що відповідає за продовження процесу затирання кешу до наступних ланок системи. Наприклад, виклик Uncache(true) для кінцевого модуля повинен почати ланцюжок викликів Uncache по всім іншим модулям і таким чином очистити кеш в усій моделі, щоб вона знова могла повторити розрахунки.

Оскільки кеш може бути результатом обчислень, то метод Compute усіх модулів можна винести в окремий інтерфейс IComputable. А для об’єднання цих інтерфейсів при їх використанні в модулях можна створити тип IComputingCacheable (Рисунок 3.12).

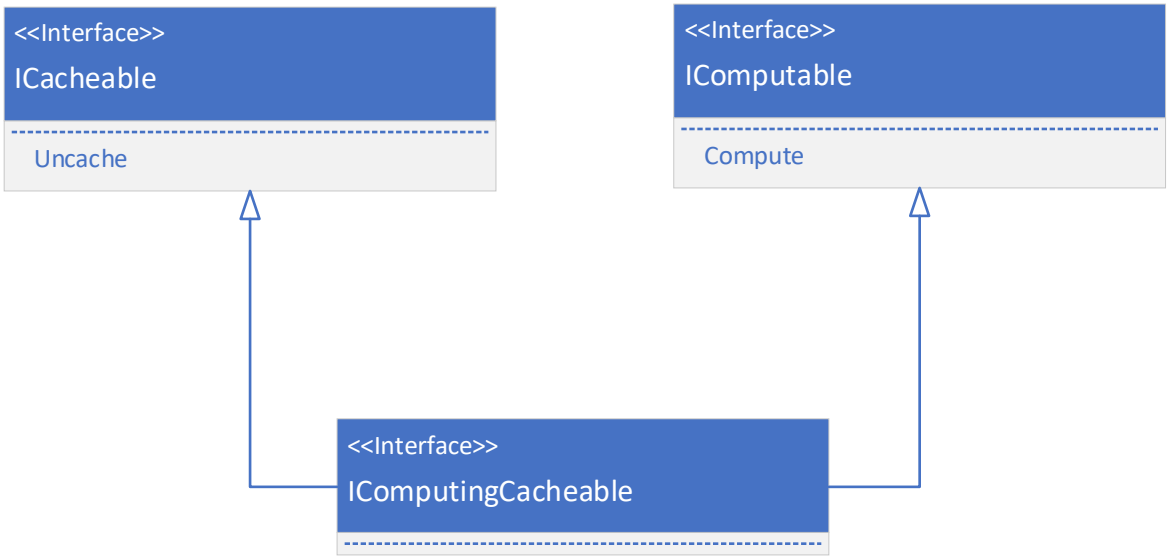


Рисунок 3.12 – Інтерфейси кешування-розрахунків

Тепер можна виділити ще один рівень інтерфейсів, що характеризує саме модулі: IAiModule, та IMutliAiModule. Обидва представляють для реалізації свій

варіант методу Extract. В першому випадку він повертає єдине значення, в другому – список значень (Рисунок 3.13).

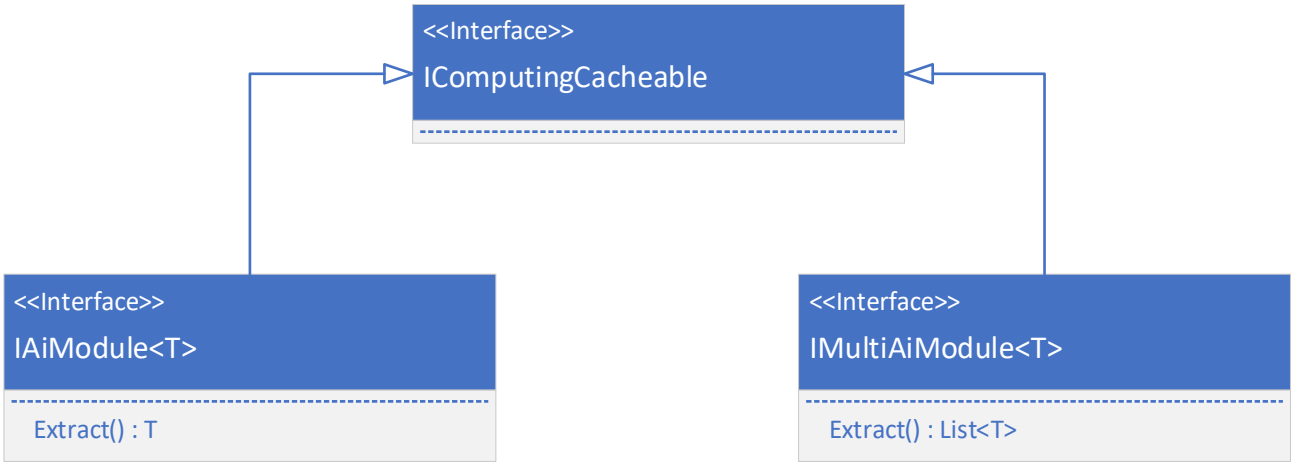


Рисунок 3.13 – Інтерфейси абстрагування модулів

Також на основі вибраних вхідних та вихідних параметрів делегатів, що застосовуються в делегатних модулях, можна сформувати свої типи делегатів:

– SingleParameterAiDelegate<T>(T) – тип, що характеризує функції, які приймають параметр T і повертають T;

– DoubleParameterAiDelegate<T>(T, T) – тип, що характеризує функції, які приймають два параметри T і повертають T;

– MultiParameterMultiResultDelegate<T>(List<T>, T) – тип, що характеризує функції, які приймають параметр-список з елементів T та критерій T, і повертають список елементів T;

– MutliParameterSingleResultDelegate<T>(List<T>, T) – тип, що характеризує функції, які приймають параметр-список з елементів T та критерій T, і повертають T.

Завдяки появі більш абстрактних інтерфейсів гнучкість моделей може бути піднята за рахунок простої реалізації цих інтерфейсів класами, які зовсім не обов’язково мають сприйматись як класичні модулі. Завдяки визначеним делегатам при розробці не доведеться знову і знову визначати перелік вхідних та вихідних параметрів, оскільки за ними закріплено окремий тип даних.

3.5 Обґрунтування вибору засобів реалізації системи

На сьогоднішній день Unity є найпопулярнішим ігровим рушієм для розробки ігрових проектів на різних платформах різної складності. Також в нього є ціла платформа Asset Store, на якій користувачі, розробники з усього світу поширюють свої інструменти, ресурси та бібліотеки. При реалізації повноцінної бібліотеки з візуальним редактором моделей складної поведінки штучного інтелекту, на цій платформі можна буде поділитись нею з усім світом. Особливо це можна досягти завдяки технології ScriptableObject, як дозволить проектувати моделі поведінки відірвано від ігрових сцен та об’єктів, зберігати їх окремо і формувати цілі шаблони моделей, без необхідності їх прив’язки до об’єктів.

Одним з важливих факторів при виборі є і мова програмування C# і компонентна архітектура рушія, завдяки яким реалізувати систему набагато простіше, користуючись потужними засобами об’єктно-орієнтованого програмування мови C# і можливість легко додавати компоненти-адаптери до ігрових об’єктів, які забезпечать їх зв’язок з системою.

3.6 Висновки

Отже, в даному розділі вдалось виконати наступні завдання:

1) розроблено алгоритм вирішення задачі моделювання;

2) визначено вимоги до майбутньої програмної системи з урахуванням всім переваг, що очікуються від неї;

3) спроектована та оптимізована архітектура модулів системи, яка буде використовуватись для її реалізації, проведено абстрагування модулів в інтерфейси для кращої відповідності принципам SOLID;

4) вибрано мову та технології програмування, на базі яких буде реалізовуватись система;

На основі результатів, одержаних на цих етапах, почата робота над програмною реалізацією системи.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

4.1 Програмна реалізація

4.1.1 Структура та призначення модулів системи, їхній взаємозв’язок

На етапі проектування в розділі 3 були розроблені модулі для програмної системи, що дозволяють створювати вузлові моделі поведінки ігрового штучного інтелекту (Таблиця 4.1).

Таблиця 4.1 – Модулі програмної системи

Назва модуля	Призначення
AiModule	Зберігання статичних даних
MultiAiModule	Зберігання списку статичних даних
SingleConnectedAiModule,	Розширення AiModule, одержання даних з будь-якого інтерфейсу IAiModule, їх обробка та повернення результату
SingleConnectedMultiAiModule	Розширення MultiAiModule, одержання даних з будь-якого іншого модуля IMultiAiModule, їх обробка та повернення результату
MultiConnectedAiModule,	Розширення AiModule, одержання списку даних з списку будь-яких інших IAiModule, їх обробка за критерієм та повернення результату або списку результатів
MultiConnectedMultiAiModule	Розширення MultiAiModule, одержання списку даних з списку будь-яких інших IMultiAiModule, їх обробка за критерієм та повернення результату або списку результатів
DoubleConnectedAiModule	Розширення AiModule, Одержання даних з будь-яких двох IAiModule, їх обробка та повернення результату

На рисунку 4.1 представлена загальна діаграма класів програмної системи.

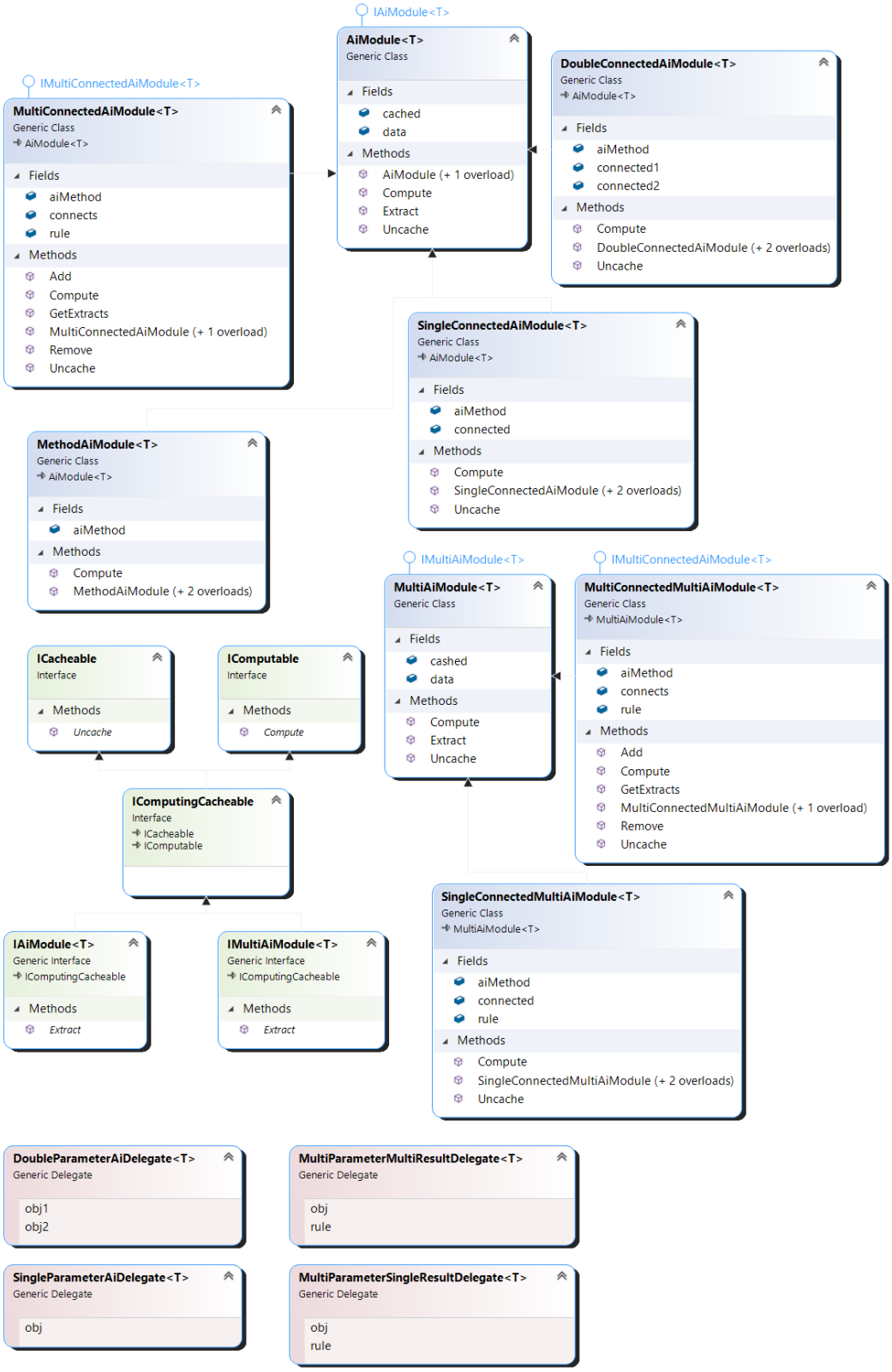


Рисунок 4.1 – Діаграма класів програмної системи

4.1.2 Розробка програмних модулів

Функції, що застосовуються в модулях системи, мають бути шаблонними. Для того, щоб гарантувати використання правильних типів функцій для модулів, треба ці типи оголосити у вигляді окремих типів-делегатів мови C#. В розділі 3 було створено 4 таких типи;

– делегат, що відповідає за обробку одного аргументу, і повернення одного значення того ж самого типу;

– делегат, що відповідає за обробку двох аргументів, і повернення одного значення того ж самого типу;

– делегат, що відповідає за обробку списку аргументів, і повернення одного значення того ж самого типу;

– делегат, що відповідає за обробку списку аргументів, і повернення списку значень того ж самого типу.

Реалізація цих делегатів:

```
public delegate T SingleParameterAiDelegate<T>(T obj);
public delegate T DoubleParameterAiDelegate<T>(T obj1, T obj2);
public delegate T MultiParameterSingleResultDelegate<T>(List<T> obj, T rule);
public delegate List<T> MultiParameterMultiResultDelegate<T>(List<T> obj, T rule);
```

Далі оголушуються базові інтерфейси системи. Інтерфес ICachable призначений для об’єктів, які можуть кешувати свої дані:

```
public interface ICacheable
{
     void Uncache(bool uncashAll);
}
```

Інтерфейс IComputable буде відображенням сутностей, в яких може відбуватись розрахунок:

```
public interface IComputable
{
     void Compute();
}
```

Ці інтерфейси можна тепер поєднати через наслідування в одному інтерфейсі IComputingCachable:

```
public interface IComputingCacheable:ICacheable,IComputable
{
}
```

Шаблонний інтерфейс IAiModule<T> буде відображенням логіки модулів, що повертають одне значення шаблонного типу. Він буде наслідувати інтерфейс IComputingCacheable, щоб підтримувати його методи:

```
public interface IAiModule<T> : IComputingCacheable
{
     T Extract();
}
```

Шаблонний інтерфейс IMutliAiModule<T> має таку ж логіку, тільки для повернення списку значень шаблонного типу:

```
public interface IMultiAiModule<T> : IComputingCacheable
{
     List<T> Extract();
}
```

Шаблонний інтерфейс IMultiConnectedAiModule<T> дає можливість агрегувати в собі список модулів та обробляти їх дані як список.

```
public interface IMultiConnectedAiModule<T>
{
     List<T> GetExtracts();
     void Add(IAiModule<T> neuron);
     void Remove(IAiModule<T> neuron);
}
```

Реалізація шаблонного модуля AiModule<T>:

```
public class AiModule<T> : IAiModule<T>
{
     public bool cached = false;
```

```
    public T data;

    public AiModule() { }
    public AiModule(T data) => this.data = data;

    public virtual T Extract()
    {
            if (!cached)
            {
                  Compute();
                  cached = true;
            }
            return data;
    }
    public virtual void Compute()
    {
    }
    public virtual void Uncache(bool uncashAll)
    {
            if (!cached) return;
            cached = false;
    }
}
```

В ньому уже закладений порожній метод Compute, який можна буде перевизначити при наслідуванні, реалізувавши розрахунки або виклик функції для розрахунків. Цей метод буде викликатись методом Extract(), який кешуватиме результати його роботи і повертатиме їх.

Концептуальної різниці в роботі наслідувачів AiModule не передбачається. Модуль SingleConnectedAiModule дозволить з’єднатись з іншим модулем і обробити його дані. Для цього треба додати поле інтерфейсу IAiModule шаблонного типу:

```
public class SingleConnectedAiModule<T>:AiModule<T>
    {
        public IAiModule<T> connected;

        public SingleConnectedAiModule() { }

        public SingleConnectedAiModule(IAiModule<T> connected) => this.connected =
connected;

        public SingleConnectedAiModule(IAiModule<T> connected,
SingleParameterAiDelegate<T> aiMethod):this(connected)
        {
            this.aiMethod = aiMethod;
        }

        public SingleParameterAiDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(connected.Extract());
        }
```

```
        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll) connected.Uncache(uncacheAll);
        }
    }
```

Варто зазначити, що використання посилань на інтерфейси як механізм з'єднання дозволить з'єднатись з будь-яким класом, який реалізовує цей інтерфейс, навіть якщо він не є наслідником AiModule. Причому будь-який інший клас може дуже просто реалізувати цей інтерфейс наряду з наслідуванням інших класів чи реалізацією інших інтерфейсів, адже в C# хоч і відсутній механізм множинного наслідування класів, але присутній механізм множинної реалізації інтерфейсів. Модуль DoubleConnectedAiModule дозволить так само обробити дані, тільки з двох інших модулів:

```
public class DoubleConnectedAiModule<T> : AiModule<T>
    {
        public IAiModule<T> connected1, connected2;

        public DoubleConnectedAiModule() { }

        public DoubleConnectedAiModule(IAiModule<T> connected1, IAiModule<T>
connected2)
        {
            this.connected1 = connected1;
            this.connected2 = connected2;
        }

        public DoubleConnectedAiModule(IAiModule<T> connected1, IAiModule<T>
connected2, DoubleParameterAiDelegate<T>
activatingFunction):this(connected1,connected2)
        {
            this.aiMethod = activatingFunction;
        }

        public DoubleParameterAiDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(connected1.Extract(),connected2.Extract());
        }

        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll)
            {
                connected1.Uncache(uncacheAll);
                connected2.Uncache(uncacheAll);
```

```
            }
        }
    }
```

Модуль MultiConnectedAiModule<T> дозволить реалізувати шаблонний модуль, які з'єднується з списком інших модулів. Він уже додатково реалізує інтерфейс IMultiConnectedAiModule:

```
public class MultiConnectedAiModule<T> : AiModule<T>,IMultiConnectedAiModule<T>
    {
        public List<IAiModule<T>> connects = new List<IAiModule<T>>();
        public IAiModule<T> rule;
        public MultiConnectedAiModule() { }

        public List<T> GetExtracts()
        {
            List<T> extracts = new List<T>();
            foreach (IAiModule<T> neuron in connects) extracts.Add(neuron.Extract());
            return extracts;
        }

        public MultiConnectedAiModule(IAiModule<T> rule,
MultiParameterSingleResultDelegate<T> aiMethod)
        {
            this.aiMethod = aiMethod;
            if (rule == null) rule = new AiModule<T>();
            this.rule = rule;
        }

        public MultiParameterSingleResultDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(GetExtracts(),rule.Extract());
        }
        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll)
            {
                foreach (IAiModule<T> neuron in connects) neuron.Uncache(uncacheAll);
            }
        }
    }
```

MultiAiModule – аналог AiModule, що повертає списки значень:

```
public class MultiAiModule<T>:IMultiAiModule<T>
    {
        public bool cashed = false;
        public List<T> data;
        public virtual List<T> Extract()
        {
            if (!cashed)
            {
```

```
                Compute();
                cashed = true;
            }
            return data;
        }
        public virtual void Compute()
        {

        }
        public virtual void Uncache(bool uncacheAll)
        {
            if (!cashed) return;
            cashed = false;

        }
    }
```

На основі нього працюватиме модуль SingleConnectedAiModule, що дозволяє з’єднуватись з одним інтерфейсом IMultiAiModule.

```
public class SingleConnectedMultiAiModule<T> : MultiAiModule<T>
    {
        public IMultiAiModule<T> connected;
        public IAiModule<T> rule;
        public SingleConnectedMultiAiModule() {}
        public SingleConnectedMultiAiModule(IMultiAiModule<T> connected) =>
this.connected = connected;
        public SingleConnectedMultiAiModule(IMultiAiModule<T> connected, IAiModule<T>
rule, MultiParameterMultiResultDelegate<T> aiMethod):this(connected)
        {
            this.aiMethod = aiMethod;
            if (rule == null) rule = new AiModule<T>();
            this.rule = rule;
        }

        public MultiParameterMultiResultDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(connected.Extract(),rule.Extract());
        }

        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll)
            {
                connected.Uncache(uncacheAll);
            }
        }
    }
```

Модуль MultiConnectedMultiAiModule – останній шаблонний модуль, який дозволить опрацювати в собі список модулів IAiModule і опрацьовувати їх на базі критерія. Можна помітити посилання на цей критерій як окремий IAiModule.

```
public class MultiConnectedMultiAiModule<T> : MultiAiModule<T>,
IMultiConnectedAiModule<T>
    {
        public List<IAiModule<T>> connects = new List<IAiModule<T>>();
        public IAiModule<T> rule;

        public List<T> GetExtracts()
        {
            List<T> extracts = new List<T>();
            foreach (IAiModule<T> neuron in connects) extracts.Add(neuron.Extract());
            return extracts;
        }

        public MultiConnectedMultiAiModule() { }

        public MultiConnectedMultiAiModule(IAiModule<T>
rule,MultiParameterMultiResultDelegate<T> aiMethod)
        {
            this.aiMethod = aiMethod;
            if (rule == null) rule = new AiModule<T>();
            this.rule = rule;
        }

        public MultiParameterMultiResultDelegate<T> aiMethod;


    }
}
```

Таким чином, програмна система може вважатись реалізованою за усіма вимогами.

4.2 Результати тестування системи та їх аналіз

4.2.1 Вибір методів тестування

Для тестування окремих елементів функціональності, бібліотек, систем, використовують метод модульного тестування.

Модульне тестування (Unit testing) – тестування кожної атомарної функціональності додатку окремо, в штучно створеному середовищі [17]. Саме потреба у створенні штучної робочого середовища для певного модуля, вимагає від тестувальника знань в автоматизації тестування програмного забезпечення, деяких

навичок програмування. Дане середовище для деякого модуля створюється за допомогою драйверів і заглушок.

Середовище Visual Studio дозволяє створити тестові проекти і класи, які дозволяють протестувати окрему частину функціональності. Створення кожного модульного тесту будемо проводити в наступні кроки:

1) створити модульний тест;

2) створити і реалізувати імперативним програмуванням якийсь математичний алгоритм;

3) створити модифіковану версію алгоритму;

4) реалізувати словник функцій на основі типу даних в цих алгоритмах, який дозволяє покрити всі операції в них;

5) реалізувати вузлову модель алгоритму на основі модулів;

6) методом виконання алгоритму двома способами на базі випадкових вхідних даних перевірити, чи співпадає результат роботи в способі імперативного програмування з результатом роботи в способі вузлової модульної моделі;

7) модифікувати вузлову модель відповідно до модифікованої версії алгоритму, і повторити крок 6 з використанням саме модифікованого алгоритму.

Таким чином, можна перевірити, чи працює модель алгоритму таким же чином, як його імперативна реалізація, і потім перевірити, чи модифікація даної моделі не вплине на правильність її роботи в порівнянні з модифікацією імперативного алгоритму.

4.2.2 Розробка тестових сценаріїв

Для проведення модульного тестування скористаємось прикладом алгоритму на базі дійсних чисел, наведеного в другому розділі:

```
public float F(float a, float b, float c)
        {
            float x = a + b;
            float y = a * x;
            float n = a - c * y;
```

```
            return n;
        }
```

Для прикладу буде створено FModel для тестування моделі, побудованої на вузлах, і в конструкторі якого при будівництві моделі викликатимемо такий метод:

```
private void BuildModel()
        {
            aModule = new AiModule<float>();
            bModule = new AiModule<float>();
            cModule = new AiModule<float>();
            xModule = new DoubleConnectedAiModule<float>(aModule, bModule, Sum);
            yModule = new DoubleConnectedAiModule<float>(aModule, xModule, Multiply);
            cyModule = new DoubleConnectedAiModule<float>(cModule, yModule,
Multiply);
            nModule = result = new DoubleConnectedAiModule<float>(aModule, cyModule,
Subtract);
        }
```

Для перевірки співпадіння результатів обрахунків буде використано перевірку, в якій допускається похибка в 1^{-6} (похибку необхідно враховувати через особливості роботи комп'ютера з дійсними числами, при яких одна і та ж операція над тими ж самим даними може мати мінімальну похибку):

```
public void CheckResults(float expected, float actual)
        {
            Assert.IsTrue(Math.Abs(expected - actual) < 0.000001);
        }
```

Наступним кроком є реалізація тестового методу, в якому 100 разів згенерується випадковий набір вхідних даних. На базі цього набору будуть проведені розрахунки імперативною процедурою та вузловою моделлю, і порівняно ці результати:

```
[TestMethod]
public void TestF()
        {
            FModel fModel = new FModel();
            Random random = new Random();
            for (int i = 0;i<100;i++)
            {
                float a = random.Next(-100,100);
                float b = random.Next(-100,100);
                float c = random.Next(-100,100);
                float expected = F(a, b, c);
```

```
                float actual = fModel.GetValue(a, b, c);
                Console.WriteLine(expected + " = "+ actual);
                CheckResults(expected, actual);
            }
        }
```

Далі розробляється модифікована версію цього алгоритму з додаванням однієї додаткової дії:

```
public float F_Modified(float a, float b, float c)
        {
            float x = a + b;
            float y = a * x;
            float n = a - c * y;
            float m = n / x;
            return m;
        }
```

І для моделі реалізовується метод для додавання модифікації в модель:

```
public void Modify()
        {
            result = mModule = new DoubleConnectedAiModule<float>(nModule, xModule,
Divide);
        }
```

Також знадобиться другий тестовий сценарій для модифікованої версії:

```
[TestMethod]
public void TestF_Modified()
        {
            FModel fModel = new FModel();
            fModel.Modify();
            Random random = new Random();
            for (int i = 0; i < 100; i++)
            {
                float a = random.Next(-100, 100);
                float b = random.Next(-100, 100);
                float c = random.Next(-100, 100);
                float expected = F_Modified(a, b, c);
                float actual = fModel.GetValue(a, b, c);
                Console.WriteLine(expected + " = " + actual);
                CheckResults(expected, actual);
            }
        }
```

В ньому можна побачити використання цієї ж моделі, але з викликом методу модифікації, а замість методу F() викликається метод F_Modified().

Наступний метод покликаний перевірити роботу модулів, що працюють зі списками даних. В ньому створюється список з 100 випадкових чисел від одного до 100 і для кожного з них створюється свій AiModule. Кожен з цих модулів приєднує до себе MultiConnectedMultiAiModule. Завдання – створити список тільки з елементами більше 50. Це буде зроблено стандартними методами роботи зі списком в C#:

```
randomInts = randomInts.FindAll(x => x > 50);
```

І в той же час це буде зроблено через передачу лямбда виразу в MultiConnectedMultiAiModule:

```
            MultiConnectedMultiAiModule<int> module = new
MultiConnectedMultiAiModule<int>(new AiModule<int>(50),(list,
divider)=>list.FindAll(x=>x>divider));
```

Якщо два списки будуть рівними, то можна вважати, що модель працює:

```
Assert.IsTrue(randomInts.SequenceEqual(moduleExtract));
```

Також будуть створені тестові сценарії TestFSpeed та TestFModelSpeed, щоб оцінити швидкість виконання простого алгоритму. TestFSpeed виконає мільйон разів фукнцію F на випадковому наборі вхідних даних, в той час як TestFModelSpeed мільйон разів використає вузлову модель функції F. Побачивши результат в витратах часу, можна судити про ефективність роботи моделі.

4.2.3 Аналіз результатів тестування

Оскільки тестування працює на генерації великої кількості випадкових вхідних даних, з пройдених тестів слідує те, що система працює. (Рисунок 4.2)

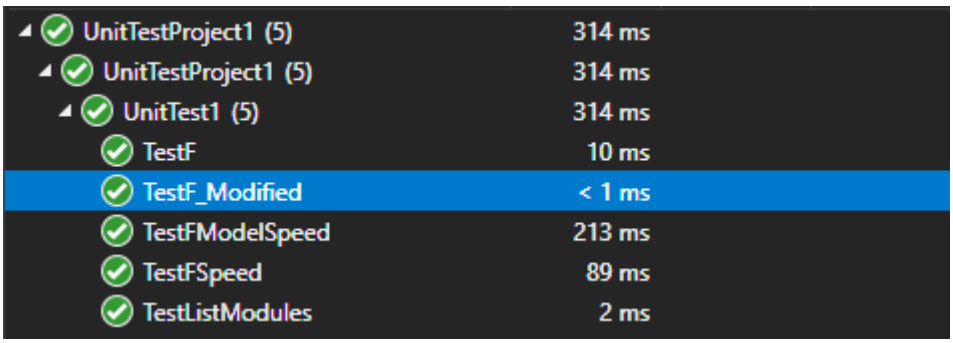


Рисунок 4.2 – Результати тестування

Перший тест був успішним, що свідчить про те, що система дозволяє виконувати алгоритм за описаним методом.

Успішність другого тесту свідчить про те, що приєднання нових модулів і модифікація алгоритму склалась успішно.

Успішність п’ятого тесту свідчить про те, що система здатна працювати з лямбда-виразами, списками, приєднувати до модулів велику кількість інших модулів, і досягати таким чином великої гнучкості в прийнятті безлічі рішень при моделюванні алгоритмів поведінки штучного інтелекту.

Третій і четвертий тести показують швидкісну характеристику виконання різних варіантів реалізації алгоритму. Функція, реалізована у вигляді вузлової моделі, виконує свою роботу більш ніж вдвічі повільніше. Втім, це стається за рахунок внутрішнього механізму кешування моделі, яке потребує очистки кешу перед повторним виконанням всього алгоритму.

4.3 Оцінка ефективності моделей та методів вирішення задач

Треба враховувати, що переваги вузлової моделі нівелюються при надто простих задачах. Якщо немає необхідності постійно ускладнювати і деталізувати алгоритм, як це працює при роботі з ігровим штучним інтелектом, то в виникає надлишкове накопичення модулів з занадто малою відповідальністю, і робота з вузловою моделлю не спрощує, а навпаки, ускладнює алгоритм.

Однак робота з сутнісною моделлю показала і серйозні переваги при модифікації моделі, де не має потенційно ненадійного втручання в внутрішню структуру операцій, а тільки доповнення або заміна існуючих функцій або модулів. Особливо це проявляється при щепленні двох моделей, як це демонструється на прикладі наступного коду:

```
public class DefenceBrainArea : BrainArea
    {
        public MultiConnectedMultiAiModule<ObjectFactor> dangers;
        public SingleConnectedMultiAiModule<ObjectFactor> fearEffect;

        public DefenceBrainArea(SelfBrainArea self)
        {
            dangers = new MultiConnectedMultiAiModule<ObjectFactor>(self.frightPoint,
AiObjectFactorMethods.RemoveAllLarger);
            fearEffect = new SingleConnectedMultiAiModule<ObjectFactor>(dangers,
self.fearFactor, AiObjectFactorMethods.MultiplyAll);
            mainOutput = fearEffect;
        }
    }
```

Тут показано конструктор моделі, який застосовує модулі більш глобальної моделі, яка передається в конструктор.

Ще однією перевагою використання даної моделі є можливість реалізації візуального редактору моделей. Для ефективної роботи з такими редакторами потрібно використовувати декілька форматів вхідних/вихідних даних з великими словниками функцій, які дозволять покривати широкий спектр задач. Найбільш важливими форматами даних, що дозволять сфокусуватись на своїх задачах, є наступні формати:

– фактор об’єкта – структура, яка містить в собі посилання на об’єкт і дійсне число-фактор об’єкта. Даний формат дозволить будувати моделі прийняття рішень штучного інтелекту, оскільки вони містять в собі і посилання на об’єкт, і дійсне число, що виражає силу впливу або інший кількісний параметр цього об’єкта. Під впливом таких параметрів всередині моделі і буде прийматись рішення, а на основі посилання на об’єкт можна визначити, який саме об’єкт є джерелом такого впливу. При створенні моделей з використанням цього формату можна буде легко доповнювати

модель і ускладнювати поведінку ШІ різними числовими факторами, такі як емоції, рівень умінь, характер, тощо. Структура ObjectFactor:

```
public struct ObjectFactor
{
    public Object obj;
    public float factor;
}
```

– вектор об’єкта – на основі посилання на об’єкт і його векторної характеристики можна вирішувати широкий спектр завдань, пов’язаний з просторовим аналізом ситуації штучним інтелектом та прийняттям рішення, пов’язаним з рухом об’єкта в потрібному напрямку. Як і в випадку з ObjectFactor, поведінку можна ускладнювати внесенням нових модулів в модель, які можуть хоча б мікроскопічно, але вплинути на просторові переміщення об’єкта, яким керує ШІ:

```
public struct ObjectVector
{
    public Object obj;
    public Vector3 vector;
}
```

4.4 Висновки

В даному розділі була виконані наступні завдання:

1) Визначена структура та зв’язки модулів системи;
2) Реалізовані програмні модулі для побудови моделей ШІ;
3) Розроблено тестові сценарії для тестування моделей як на предмет точності виконання своїх задач, так і швидкості;
4) Проведено тестування системи за допомогою модульних тестів;
5) Проведена аналіз та оцінка тестових результатів, на основі яких визначені закономірності та рекомендації до застосування програмної системи;
6) Оцінено можливі способи розширення програмної системи, шляхи її вдосконалення, та найкращі способи її використання.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було розроблено програмну систему для моделювання складної поведінки ігрового штучного інтелекту.

У першому розділі було досліджено існуючи принципи, алгоритми, методи, архітектурні рішення для моделювання та реалізації штучного інтелекту в ігрових проектах, проведено їх аналіз на позитивні та негативні сторони. На основі проведених досліджень введено ряд функціональних та нефункціональних вимог від майбутньої системи та проведено розгорнуту постановку задачі.

У другому розділі проведено дослідження можливих архітектурних та алгоритмічних способів рішення задачі. На основі вимог до гнучкості системи було синтезовано вузловий метод моделювання алгоритму. Проведено оцінку його переваг та недоліків в контексті сучасних проблем.

У третьому розділі обґрунтований алгоритм роботи програмної системи для побудови вузлових моделей, проаналізовано особливості його реалізації. На основі алгоритму було проведено детальне проектування і розроблена архітектура системи. Також проведені оптимізаційні процеси та приведення системи до відповідності принципам SOLID.

У четвертому розділі описана об’єктно-орієнтована структура системи, основні призначення її модулів та елементів. Програмна система реалізована на мові C# з орієнтацією на ігровий рушій Unity. Розроблені тестові сценарії, на основі яких була протестована програмна система на предмет точності та ефективності її роботи. Результати тестування проаналізовані та описані оптимальні шляхи роботи з системою, визначені закономірності та рекомендації для роботи з нею, оцінено можливості майбутнього її розширення.

Розроблена програмна система дозволяє моделювати складну поведінку штучного інтелекту, будуючи її на основі вузлової діаграми алгоритмів поведінки ШІ, дозволяє легко змінювати її, підтримувати будь-які типи даних та операцій на них, здійснювати їх кешування, а також дозволяє з’єднувати зв’язками різні моделі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1) Gregory J. Game Engine Architecture, 2nd edition. / Gregory J. // Taylor & Francis Ltd, 2014. – 900 с.

2) How Do Game Engines Work? [Електронний ресурс]. – Режим доступу: http://surl.li/aulob

3) Bennett C. Unity AI Programming Essentials. / Bennett C., Sagmiller D. // Packt, 2014. – 158 с.

4) Структурний патерн адаптер [Електронний ресурс]. – Режим доступу: https://refactoring.guru/uk/design-patterns/adapter

5) Vaskaran S. Design Patterns in C# / Vaskaran S. // Appress, 2018. – 455 с.

6) JavaTPoint – Unity Components [Електронний ресурс]. – Режим доступу: https://www.javatpoint.com/unity-components

7) Unity Manual – Audio [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/Audio.html

8) Unity Manual – Physics [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/PhysicsSection.html

9) The Last of Us 2 Dev Reveals Major AI Improvements [Електронний ресурс]. – Режим доступу: https://gamerant.com/last-of-us-2-ai-improvements/

10) The Perfect Organism: The AI of Alien: Isolation [Електронний ресурс]. – Режим доступу: http://surl.li/auloe

11) Troelsen A. C# 6.0 and the .NET 4.6 Framework, 6th edition. / Troelsen A., Japikse P. // Apress, 2015. – 1837с.

12) Schell J. The Art pf Game Design, a Book of Lenses / Schell J. // CRC Press, 2015. – 594с.

13) DaGraca M. Practical Game AI Programming / DaGraca M. // Packt, 2017. – 341с.

14) Artificial intelligence (AI) programmer [Електронний ресурс]. – Режим доступу: http://surl.li/aulns

15) Palacios J. Unity 5.x Game AI Programming Cookbook / Palacios J. // Packt, 2016. – 278c.

16) S.O.L.I.D principles for Unity [Електронний ресурс]. – Режим доступу: http://surl.li/auloa

17) Модульне тестування [Електронний ресурс]. – Режим доступу: http://surl.li/aulog

18) Єлісеєнко О. Порівняльний аналіз сучасних гральних рушіїв. *КОНФЕРЕНЦІЇ ВНТУ електронні наукові видання, XLV Науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії*. 2016. URL: http://surl.li/aulnw

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

А.1 Програмні інтерфейси:

```
public interface ICacheable
    {
        void Uncache(bool uncashAll);
    }

    public interface IComputable
    {
        void Compute();
    }
    public interface IComputingCacheable:ICacheable,IComputable
    {
    }

public interface IAiModule<T> : IComputingCacheable
    {
        T Extract();
    }

    public interface IMultiAiModule<T> : IComputingCacheable
    {
        List<T> Extract();
    }

public interface IMultiConnectedAiModule<T>
    {
        List<T> GetExtracts();
        void Add(IAiModule<T> neuron);
        void Remove(IAiModule<T> neuron);
    }
```

А.2 Модулі одиночного результату

```
public class AiModule<T> : IAiModule<T>
    {
        public bool cached = false;
        public T data;

        public AiModule() { }
        public AiModule(T data) => this.data = data;

        public virtual T Extract()
        {
            if (!cached)
            {
                Compute();
```

```
                cached = true;
            }
            return data;
        }
        public virtual void Compute()
        {

        }
        public virtual void Uncache(bool uncashAll)
        {
            if (!cached) return;
            cached = false;
        }
    }

public class SingleConnectedAiModule<T>:AiModule<T>
    {
        public IAiModule<T> connected;

        public SingleConnectedAiModule() { }

        public SingleConnectedAiModule(IAiModule<T> connected) => this.connected =
connected;

        public SingleConnectedAiModule(IAiModule<T> connected,
SingleParameterAiDelegate<T> aiMethod):this(connected)
        {
            this.aiMethod = aiMethod;
        }

        public SingleParameterAiDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(connected.Extract());
        }

        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll) connected.Uncache(uncacheAll);
        }
    }

    public class DoubleConnectedAiModule<T> : AiModule<T>
    {
        public IAiModule<T> connected1, connected2;

        public DoubleConnectedAiModule() { }

        public DoubleConnectedAiModule(IAiModule<T> connected1, IAiModule<T>
connected2)
        {
            this.connected1 = connected1;
            this.connected2 = connected2;
        }

        public DoubleConnectedAiModule(IAiModule<T> connected1, IAiModule<T>
connected2, DoubleParameterAiDelegate<T>
activatingFunction):this(connected1,connected2)
        {
            this.aiMethod = activatingFunction;
        }
```

```
        public DoubleParameterAiDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(connected1.Extract(),connected2.Extract());
        }

        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll)
            {
                connected1.Uncache(uncacheAll);
                connected2.Uncache(uncacheAll);
            }
        }
    }


    public class MultiConnectedAiModule<T> : AiModule<T>,IMultiConnectedAiModule<T>
    {
        public List<IAiModule<T>> connects = new List<IAiModule<T>>();
        public IAiModule<T> rule;
        public MultiConnectedAiModule() { }

        public List<T> GetExtracts()
        {
            List<T> extracts = new List<T>();
            foreach (IAiModule<T> neuron in connects) extracts.Add(neuron.Extract());
            return extracts;
        }

        public void Add(IAiModule<T> neuron)
        {
            if (!connects.Contains(neuron))
            {
                connects.Add(neuron);
            }
        }

        public void Remove(IAiModule<T> neuron)
        {
            connects.Remove(neuron);
        }

        public MultiConnectedAiModule(IAiModule<T> rule,
MultiParameterSingleResultDelegate<T> aiMethod)
        {
            this.aiMethod = aiMethod;
            if (rule == null) rule = new AiModule<T>();
            this.rule = rule;
        }

        public MultiParameterSingleResultDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(GetExtracts(),rule.Extract());
        }
        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
```

```
            if (uncacheAll)
            {
                foreach (IAiModule<T> neuron in connects) neuron.Uncache(uncacheAll);
            }
        }
    }
```

A.3 Модулі результату-списку

```
public class MultiAiModule<T>:IMultiAiModule<T>
    {
        public bool cashed = false;
        public List<T> data;
        public virtual List<T> Extract()
        {
            if (!cashed)
            {
                Compute();
                cashed = true;
            }
            return data;
        }
        public virtual void Compute()
        {

        }
        public virtual void Uncache(bool uncacheAll)
        {
            if (!cashed) return;
            cashed = false;

        }
    }

    public class SingleConnectedMultiAiModule<T> : MultiAiModule<T>
    {
        public IMultiAiModule<T> connected;
        public IAiModule<T> rule;
        public SingleConnectedMultiAiModule() {}
        public SingleConnectedMultiAiModule(IMultiAiModule<T> connected) =>
this.connected = connected;
        public SingleConnectedMultiAiModule(IMultiAiModule<T> connected, IAiModule<T>
rule, MultiParameterMultiResultDelegate<T> aiMethod):this(connected)
        {
            this.aiMethod = aiMethod;
            if (rule == null) rule = new AiModule<T>();
            this.rule = rule;
        }

        public MultiParameterMultiResultDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(connected.Extract(),rule.Extract());
        }

        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll)
```

```
            {
                connected.Uncache(uncacheAll);
            }
        }
    }


    public class MultiConnectedMultiAiModule<T> : MultiAiModule<T>,
IMultiConnectedAiModule<T>
    {
        public List<IAiModule<T>> connects = new List<IAiModule<T>>();
        public IAiModule<T> rule;

        public List<T> GetExtracts()
        {
            List<T> extracts = new List<T>();
            foreach (IAiModule<T> neuron in connects) extracts.Add(neuron.Extract());
            return extracts;
        }

        public void Add(IAiModule<T> neuron)
        {
            if (!connects.Contains(neuron))
            {
                connects.Add(neuron);
            }
        }

        public void Remove(IAiModule<T> neuron)
        {
            connects.Remove(neuron);
        }

        public MultiConnectedMultiAiModule() { }

        public MultiConnectedMultiAiModule(IAiModule<T>
rule,MultiParameterMultiResultDelegate<T> aiMethod)
        {
            this.aiMethod = aiMethod;
            if (rule == null) rule = new AiModule<T>();
            this.rule = rule;
        }

        public MultiParameterMultiResultDelegate<T> aiMethod;

        public override void Compute()
        {
            data = aiMethod.Invoke(GetExtracts(),rule.Extract());
        }

        public override void Uncache(bool uncacheAll)
        {
            base.Uncache(uncacheAll);
            if (uncacheAll)
            {
                foreach (IAiModule<T> neuron in connects) neuron.Uncache(uncacheAll);
            }
        }
    }
```

ДОДАТОК Б

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

ISSN 2219-9365
DOI: 10.31891/2219-9365

Міжнародний науково-технічний журнал

ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ

2021, № 1

International scientific-technical journal

MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES

2021, Issue 1

Хмельницький 2021
Khmelnytskyi 2021

МІЖНАРОДНИЙ НАУКОВО-ТЕХНІЧНИЙ ЖУРНАЛ
ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ

Затверджений як фахове видання (перереєстрація), група «Б»
Наказ МОН 28.12.2019 №1643

Засновано в травні 1997 р. *Виходить 2 рази на рік*

Хмельницький, 2021, № 1 (67)

Засновник і видавець: **Хмельницький національний університет**
(до 2005 р. — Технологічний університет Поділля, м. Хмельницький)

Наукова бібліотека України ім. В.І. Вернадського http://nbuv.gov.ua/j-tit/votte

Журнал включено до наукометричних баз:

Index Copernicus	http://jml2012.indexcopernicus.com/p24781565,3.html h-індекс 49,97
Google Schoolar	http://scholar.google.com.ua/citations?user=uwN_nusAAAAJ&hl=uk - індекс 9
Національна бібліотека України ім. В.І. Вернадського	http://nbuv.gov.ua/j-tit/votte

Рекомендовано до друку рішенням Вченої ради Хмельницького національного університету, протокол № 17 від 27.05.2021

Адреса редакції:	**Україна, 29016,** **м. Хмельницький, вул. Інститутська, 11,** **Хмельницький національний університет** **редакція журналу "Вимірювальна та обчислювальна техніка в технологічних процесах"**
☎	067-347-74-57
e-mail:	**mscientificjournal@gmail.com**
web:	**http://journals.khnu.km.ua/index.php/MeasComp**

Зареєстровано Міністерством України у справах преси та інформації.
Свідоцтво про державну реєстрацію друкованого засобу масової інформації
Серія КВ № 23279-13119ПР від 24 травня 2018 року (перереєстрація)

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

ЗМІСТ

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

УДК 004.43:004.8
DOI: 10.31891/2219-9365-2021-67-1-9

КРИВИЙ В. М., ЯШИНА О. М.,
РАДЕЛЬЧУК Г. І., ЛИСЕНКО С.М.
Хмельницький національний університет

ПОРІВНЯЛЬНИЙ АНАЛІЗ ПАРАДИГМ ПРОГРАМУВАННЯ ПРИ РОЗРОБЦІ ПРОГРАМНИХ СИСТЕМ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

У статті наведено результати досліджень різних парадигм програмування та мов, що підтримують ці парадигми, на предмет наявності інструментарію для розробки штучного інтелекту. Проведена оцінка зручності їх використання, аналіз сфер їх поширеності на ринку розробки програмного забезпечення та їх ефективність.

Ключові слова: штучний інтелект, парадигма програмування, програмне забезпечення, машинне навчання, нейронна мережа.

VITALII KRYVYI, OKSANA YASHYNA.,
GALYNA RADELCHUK, SERGII LYSENKO
Khmelnytskyi National University

COMPARATIVE ANALYSIS OF PROGRAMMING PARADIGMS IN THE DEVELOPMENT OF SOFTWARE SYSTEMS BASED ON ARTIFICIAL INTELLIGENCE

Today, with the growing popularity and demand for technologies of artificial intelligence, machine learning and neural networks, the question of choosing effective tools for their development and integration into software systems becomes relevant. With the progress of computer hardware, creating special programming languages and appearing of libraries that simplify the development and use of neural networks, artificial intelligence is no longer something futuristic and frightening. It is now a fairly flexible and widespread technology that is developing quickly. And on the one hand, it is gradually being introduced into our lives to perform those tasks that were previously unavailable or extremely difficult for the computer. On the other hand, artificial intelligence is not yet advanced enough to be called a completely reliable tool and cannot replace humans in many areas of their activity, so it is mostly developing for specific tasks such as digital content processing, data analysis, vehicle piloting, simulation of character behavior in games, etc. But as in the work with any other technologies in programming, we are interested in the selection of software tools and paradigms that allow us to effectively design and develop systems based on artificial intelligence.

This article presents the results of research on different programming paradigms and languages that support these paradigms, for the availability of tools for the development of artificial intelligence. It evaluates the ease of use, analyzes the areas of their prevalence in the software development market, and their effectiveness.

Keywords: artificial intelligence, programming paradigm, software, machine learning, neural network

Вступ. Постановка проблеми

Сьогодні, із зростанням популярності та попиту на технології штучного інтелекту (ШІ), машинного навчання та нейронних мереж, актуальним стає питання вибору ефективних інструментів для їх розробки та інтеграції у програмні системи. З розвитком комп'ютерного обладнання та появою бібліотек, які спрощують розробку та використання нейронних мереж, штучний інтелект уже не є чимось футуристичним та лякаючим. Зараз це досить гнучка і поширена технологія, яка швидко розвивається. З одного боку, вона поступово впроваджується у наше життя для виконання тих завдань, які раніше були недоступними або надто важкими для виконання комп'ютером. З іншого боку, штучний інтелект ще не настільки високий, щоб його можна було назвати цілком надійним інструментом. Він не може замінити людей у багатьох сферах їх діяльності, тому здебільшого розробляється для конкретних завдань, таких, наприклад, як обробка цифрового вмісту, аналіз даних, пілотування транспортних засобів моделювання поведінки персонажа в іграх тощо. У цьому зв'язку, як і при роботі з будь-якими іншими технологіями програмування, особливий інтерес представляють парадигми програмування, які дозволяють ефективно розробляти програмні системи на основі ШІ.

Аналіз останніх досліджень та публікацій

Основою дослідження є праці відомих авторів та науковців у сфері програмування ШІ, таких як П. Грем, Д. Маккарті, І. Братко, П. Джоші та ін. Ми будемо аналізувати розробку штучного інтелекту на основі парадигм функціонального, імперативного, логічного та об'єктно-орієнтованого програмування, досліджуючи такі мови програмування як Prolog, Haskell, Python, Java, JavaScript, C#, F# та ін.

Метою роботи є: порівняльний аналіз різних парадигм та мов програмування, що підтримують ці парадигми, на предмет їх ефективності та зручності при розробці штучного інтелекту.

Виклад основного матеріалу

Парадигма програмування – це сукупність ідей та понять, які визначають стиль написання комп'ютерної програми (підхід до програмування, спосіб мислення розробника програми). Іншими словами,

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

це концептуалізація, яка визначає організацію обчислень та структурування роботи, яку має виконувати комп'ютер [1].

Основними парадигмами, що використовуються при розробці штучного інтелекту, є парадигми функціонального (ФП), логічного (ЛП), імперативного (ІП) та об'єктно-орієнтованого (ООП) програмування. Для дослідження цих парадигм знадобиться розгляд їх використання у конкретних мовах програмування, які підтримують ці парадигми, та роботі з ними при розробці ШІ.

Методологія даного дослідження полягає у використанні наступних критеріїв порівняння парадигм та мов програмування, що використовують ці парадигми:

- концепція парадигми та її пристосованість до задач, пов'язаних з розробкою ШІ;
- поширеність на ринку розробки програмного забезпечення (ПЗ) зі штучним інтелектом;
- можливість та легкість застосування у прикладних програмних системах;
- швидкість вирішення задачі.

Кожна парадигма програмування є лише теоретичною і концептуальною складовою процесу програмування, тому для їх оцінки за конкретними критеріями ми будемо не просто розглядати парадигми самі по собі, а їх конкретну реалізацію у мовах програмування, оскільки нас цікавлять не тільки теоретичні, але й практичні аспекти, з якими стикатиметься розробник програмного коду при вирішенні широкого спектру задач.

На сьогоднішній день більшість широковживаних мов програмування на ринку розробки програмних систем (таких як Python, JavaScript, C#, C++, Java та ін.) були розроблені на основі ООП- та ІП-парадигм, незалежно від того, чи є вони інтерпретованими чи компільованими. Імперативний стиль написання коду визначає програму як сукупність даних, що визначають її стан, і покрокових інструкцій, що змінюють цей стан. А об'єктно-орієнтований аспект цих мов спонукає розробника програмного коду об'єднувати ці частини коду у класи, які можуть використовуватись багаторазово, наслідувати один одного, мати зв'язки з іншими класами тощо. З моменту появи мови C++ об'єднання імперативного та об'єктно-орієнтованого стилю у мовах програмування стало трендом і залишається таким до сьогодні.

Причини такого явища є очевидними – послідовні інструкції мови імперативного програмування можна набагато легше і ефективніше конвертувати у машинний код, оскільки він також є послідовністю інструкцій. Більше того, такий підхід набагато легше пояснити початківцю у програмуванні, і з ним легше уявити процес вирішення прикладної задачі, оскільки спосіб, який полягає у заданні послідовності дій для досягнення цілі, дуже наближений до реального життя. Програма стає схожою на кулінарний рецепт, де над початковими інгредієнтами крок за кроком проводяться деякі дії, що змінюють їх стан, допоки ті не перетворяться у готову страву. Так само просто можна зрозуміти і принцип ООП, оскільки інформація про будь-який реальний чи абстрактний об'єкт розглядається в сукупності, як одна структурна одиниця, над якою визначено множину дій. Наприклад, дія «Розігрів» над об'єктом «Інгридієнт» змінює дані про «Температуру», «Смак» або «Колір» останнього.

Таким чином, можна зробити висновок, що процес програмування прикладних програмних систем на основі парадигм ІП та ООП є набагато простішим та ефективнішим, оскільки він здається «дуже логічним» з точки зору людини, і при цьому наближений до принципів виконання програм самим комп'ютером. Це все і обумовлює той факт, що ці парадигми застосовні до більшості мов програмування. Але чи достатньо ефективною та зручною є розробка штучного інтелекту за принципами, заснованими на цих парадигмах?

Як відомо, логіка роботи електронно-обчислювальних пристроїв (що обумовлено фізичними аспектами їх роботи), а також більшості мов програмування, є булевою, тобто будь-яке твердження у них представлене у вигляді «істини» (True) або «хибності» (False). Цей принцип можна без проблем застосовувати у математиці, де потрібне чітке визначення результату, чітке визначення істинності. Наприклад, можна однозначно описати істинність твердження «число X є парним», описавши алгоритм вирішення такої задачі в імперативному стилі наступним чином: якщо залишок від ділення X на 2 дорівнює 0, то твердженню присвоюється «істина», в іншому випадку – «хибність». Все, що треба знати програмі для вирішення такої задачі, – це саме число X. Але у реальному житті «природньому» інтелекту доводиться виконувати набагато складніші завдання, для яких буває дуже складно або майже неможливо описати чітку логіку рішення. Наприклад:

Чи є підсудний «X» винним?

Чи посміхається людина на фотографії?

Чи належить силует, який ми бачимо перед собою, небезпечному хижаку?

Перевірка таких тверджень може відбуватись при дуже різних наборах вхідних даних, які інколи взагалі не дають можливості знайти однозначний результат. Причому, при описі чіткого математичного алгоритму доводиться враховувати і кожну варіацію, і діапазон даних, що навіть у випадку достатньо простих задач може змусити нас збільшити об'єм вихідного коду в десятки, сотні і навіть тисячі разів.

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

Тому, замість того, щоб розробляти рішення таких складних завдань на основі алгоритмічної логіки, їх часто проектують на основі принципів роботи біологічного розуму, а саме – через використання нейронних мереж та машинного навчання. Завдяки прогресу у швидкодії та можливостях комп'ютерних комплектуючих, а також наявності у більшості популярних мов на основі ООП та ІП спеціалізованих бібліотек, такий спосіб розробки ШІ застосовується все частіше.

Але чи існують парадигми, які усувають недоліки попередніх при розробці штучного інтелекту?

Розглянемо парадигму логічного програмування на основі мови Prolog. Prolog – це мова програмування, зосереджена навколо невеликого набору основних механізмів, включаючи зіставлення зі зразком, деревоподібне представлення структур даних та автоматичний перебір з поверненнями. Цей обмежений набір засобів утворює дуже потужне та гнучке середовище програмування. Prolog особливо добре підходить для вирішення завдань, в яких розглядаються об'єкти (зокрема, структуровані об'єкти) та відношення між ними. Зокрема, засобами мови Prolog зовсім не складно виразити просторові зв'язки між об'єктами, наприклад, вказати, що синя куля знаходиться за зеленою. Настільки ж просто можна визначити загальніше правило: якщо об'єкт X знаходиться ближче до спостерігача, ніж об'єкт Y, а Y – ближче, ніж Z, то X має бути ближче, ніж Z. Після цього система Prolog отримує можливість формувати міркування про просторові зв'язки та їх сумісність із загальним правилом. Завдяки таким особливостям Prolog стає потужною мовою для розробки штучного інтелекту та нечислового програмування в цілому [2].

Таким чином, логічну парадигму у мові Prolog можна назвати сильною стороною при розробці систем штучного інтелекту. Вона дозволяє розглядати програмування як опис пошуку логічного висновку на основі існуючих фактів. При цьому програмний код не повинен однозначно характеризувати шляхи до пошуку рішення за принципами булевої логіки, де будь-яке твердження визначається лише як «істина» або «хибність». В цьому і полягає робота рішень на основі штучного інтелекту, таких як експертні системи, обробка та аналіз мовлення, доведення теорем, трасування та евристичні оцінки, оскільки усі перераховані задачі здебільшого засновані на аналізі фактів, пошуку зв'язків між ними та синтез висновку. Мова Prolog дозволяє вирішувати такі задачі набагато швидше, ніж інші мови, оскільки логічна парадигма, яку вона підтримує, і є концепцією, призначеною для їх рішення (наприклад, перший чатбот ELIZA був розроблений з використанням Prolog для обробки конструкцій мовлення [3]).

З іншого боку, концепція логічного програмування в Prolog є набагато складнішою для розуміння і освоєння е порівнянні з парадигмами функціонального, імперативного чи об'єктно-орієнтованого програмування, що кардинально відрізняє цю мову від більшості популярних мов програмування. Тому її головна перевага, що полягає у написанні програмного коду як «дерева досягнення висновку» (а не «алгоритму» у випадку імперативної парадигми), значно ускладнює застосування у прикладному програмуванні, коли кожна програма розглядається як покроковий набір інструкцій. Це робить Prolog малопоширеною мовою програмування у широкому колі розробників, тому ця мова здебільшого застосовується в академічних або дослідницьких цілях.

Таким чином, логічна парадигма, яка хоч і забезпечує потужні механізми розробки ШІ, але демонструє свої суттєві концептуальні недоліки при розширенні спектру та специфіки завдань, які потрібно вирішувати. Але що, якщо якась інша парадигма може об'єднати принципи нечіткої логіки з логічної парадигми, і при цьому дозволить мові програмування набагато ширше використовуватись як прикладний інструмент? Виходом може стати функціональна парадигма програмування.

Функціональна парадигма довгий час трималась осторонь сфери прикладної розробки. Як і у випадку з логічним програмуванням, імперативний спосіб написання коду випереджає її як більш зрозумілий та наближений до людського сприйняття принцип опису алгоритму виконання завдань. А оскільки протягом довгого часу більшість мов функціонального програмування виконувались інтерпретаторами і, оже, були значно повільнішими, то не дивно, що їх імперативні та об'єктно-орієнтовані компільовані аналоги, такі як C, C++ або Java, завоювали значно більшу популярність серед розробників.

Однак, завдяки розвитку комп'ютерної техніки, можна говорити про нівелювання вказаного недоліку, оскільки сьогодні, наприклад, компілятор GHC (Glasgow Haskell Compiler) для мови Haskell створює виконуваний код, який не поступається за ефективністю програмам, розробленими засобами мов C чи C++ [4].

У зв'язку зі стрімким зростанням машинного навчання і великих даних ФП стало набирати популярність через простоту розпаралелювання чистих функцій. Функціональна парадигма також спрощує відстеження, тестування та обслуговування коду для задач аналізу даних та робочих процесів, що створює передумови для її активного використання у найближчому майбутньому.

Однією з найпоширеніших функціональних мов є Haskell. Це винятково функціональна мова, що базується на лямбда-численні. Вона не є надто популярною, але широко використовується як в дослідженнях, так і в реалізації комерційних проектів.

У мові Haskell, як і в більшості мовах функціонального програмування, не визначається чіткий порядок виконання обчислень, – лише декларуються залежності між даними, а порядок виконання

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

визначається транслятором. Такий підхід дуже схожий на концепцію логічного програмування в Prolog. Більше того, Haskell, так само як і Prolog, допускає програмування нечіткої логіки і нечіткої математики, формування логічного висновку. Опис роботи з такими поняттями на основі функціональної парадигми наведений у [4]. Автор називає парадигму ФП «найбільш пристосованою для вирішення завдань штучного інтелекту».

З усвідомленням переваг функціональної парадигми над імперативним стилем чимало мов програмування були розширені функціональними елементами. Наприклад, у мові Python з'явились такі оператори з Haskell, як map, filter, lambda, reduce. У мові C# з'явились LinQ та лямбда-вирази. Тому популярні (здебільшого імперативні та об'єктно-орієнтовані мови) розвиваються в бік мультипарадигмового програмування, об'єднуючи простоту імперативного стилю написання коду з декларативністю функціонального програмування. Оскільки парадигми програмування утворюють висхідну лінію концептуальності, то й не дивно, що більшість сучасних мов програмування є, по суті, мультипарадигмовими (таблиця 1).

Таблиця 1

Мультипарадигмовість мов програмування

Парадигми / Мови	Функціональна	Логічна	Імперативна (процедурна)	Об'єктно-орієнтована
Ада	–	–	+	+
C	–	–	+	–
C++		–	+	+
C#		–	+	+
Java	–	–	+	+
Haskell	+		+	–
Common LISP	+		+	+
Python		–	+	+
Prolog		+	–	–
SmallTalk	+		+	+
JavaScript	–	–	+	+
F#	+	–	+	+

Висновки

У дослідженні виконано порівняльний аналіз різних парадигм та мов програмування, що підтримують ці парадигми, на предмет їх ефективності та зручності при розробці штучного інтелекту. Проведена оцінка зручності їх використання, аналіз сфер їх поширеності на ринку розробки ПЗ та їх ефективність. Встановлено, що жодна парадигма не вирішує завдань розробки ШІ найлегшим чи найефективнішим способом. Так само не існує ідеальної мови для ШІ – кожна має свої переваги та недоліки при вирішенні певних задач із різних предметних областей. Тому більшість відповідних рішень мають базуватись на комбінуванні (поєднанні) декількох технологій та парадигм, які включають різні варіанти для реалізації бажаної функціональності та досягнення високої ефективності.

Література

1. Van Roy P. Programming Paradigms for Dummies: What Every Programmer Should Know [Online] / P. Van Roy // ResearchGate. – Available: https://www.researchgate.net/publication/241111987_Programming_-_Paradigms_for_Dummies_What_Every_Programmer_Should_Know

2. Братко И. Алгоритмы искусственного интеллекта на языке PROLOG; 3-е издание: Пер. с англ. / И. Братко. – М. : Издательский дом "Вильямс", 2004. – 640 с.

3. Weizenbaum J. ELIZA – A Computer Program For the Study of Natural Language Communication Between Man And Machine / J. Weizenbaum // Communications of the ACM. – 1966. – Volume 9(1). – P. 36–45.

4. Душкин Р. В. Функциональное программирование на языке Haskell / Р. В. Душкин. – Изд-во «ДМК Пресс», 2008. – 609 с.

References

1. Van Roy P. Programming Paradigms for Dummies: What Every Programmer Should Know [Online] / P. Van Roy // ResearchGate. – Available: https://www.researchgate.net/publication/241111987_Programming_-_Paradigms_for_Dummies_What_Every_Programmer_Should_Know

2. Bratko I. Algoritmyi iskusstvennogo intellekta na yazyike PROLOG; 3-e izdanie: Per. s angl. / I. Bratko. – M. : Izdatelskiy dom "Vilyamc", 2004. – 640 s.

3. Weizenbaum J. ELIZA – A Computer Program For the Study of Natural Language Communication Between Man And Machine / J. Weizenbaum // Communications of the ACM. – 1966. – Volume 9(1). – P. 36–45.

4. Dushkin R. V. Funktsionalnoe programmirovanie na yazyike Haskell / R. V. Dushkin. – Izd-vo «DMK Press», 2008. – 609 s.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Програмна система моделювання складної поведінки ігрового штучного інтелекту

Автор роботи:
ст. гр. ІПЗм-20-1 Кривий В. М.
Керівник роботи:
к. т. н., доцент Шестакевич Т. В.

Метою дослідження є вдосконалення методу моделювання складної поведінки ігрового штучного інтелекту на основі вузлового підходу.

Об'єктом дослідження є процес моделювання складної поведінки ігрового штучного інтелекту.

Предметом дослідження є методи моделювання та реалізації ігрового штучного інтелекту.

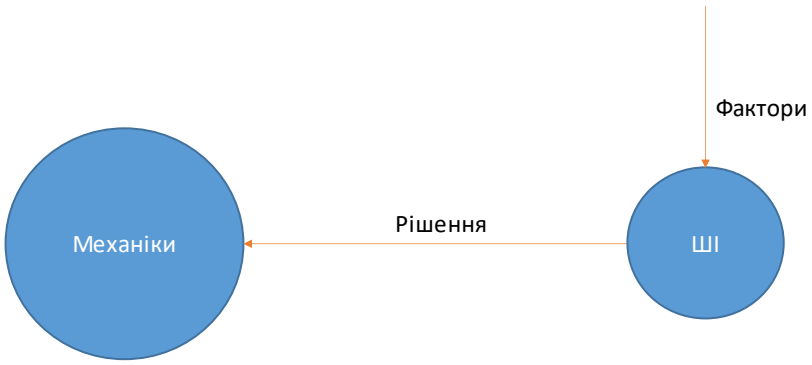
Наукова новизна отриманих результатів.

- удосконалено метод моделювання та реалізації ігрового штучного інтелекту на основі вузлового підходу;
- розроблено архітектурні шаблони, які дозволять вирішити проблему повторюваності його коду, дадуть можливість як візуально, так і програмно моделювати його поведінку та легко її модифікувати.

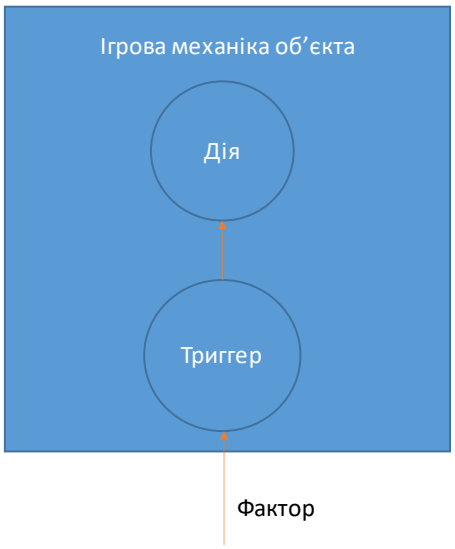
Завдання:

- розробка концепції вузлової моделі, яка допоможе досягти гнучкості та зручності при розробці складного штучного інтелекту;
- визначення основних вимог до реалізації моделей на основі цієї концепції, та функцій, які вона надає;
- проектування програмної системи або інструментарію, які допоможуть проектувати такі моделі та впроваджувати їх в ігрові об’єкти;
- реалізація програмної системи;
- тестування та практичне застосування програмної системи;
- аналіз одержаних результатів та формування рекомендацій щодо їх подальшого застосування.

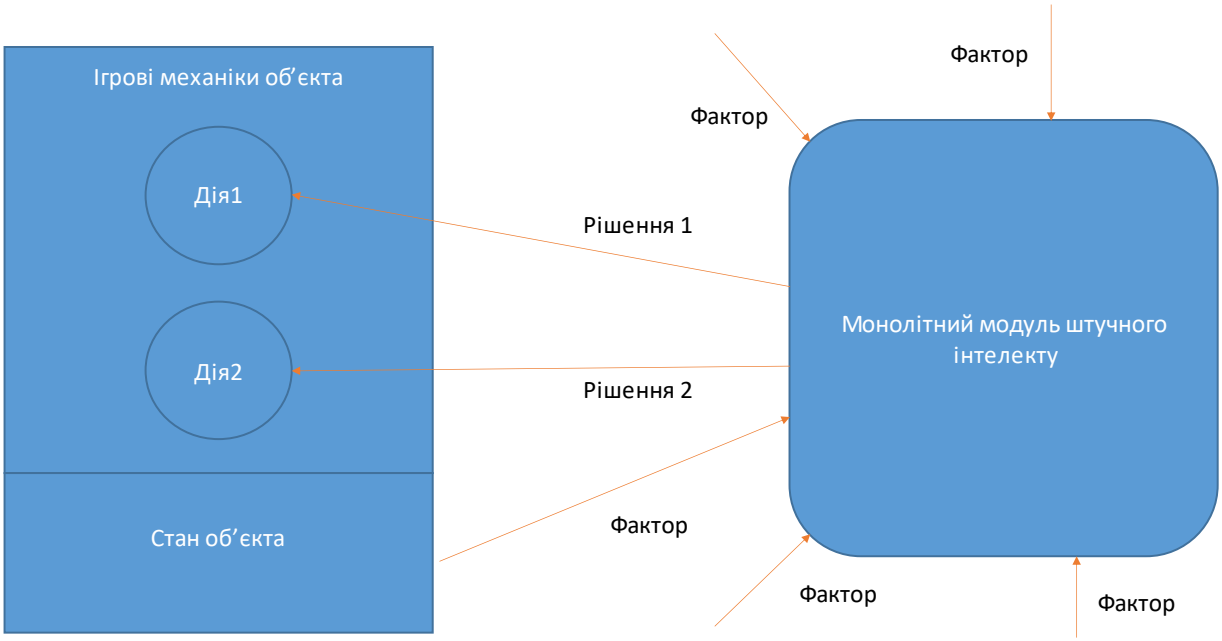
Що є ігровим штучним інтелектом



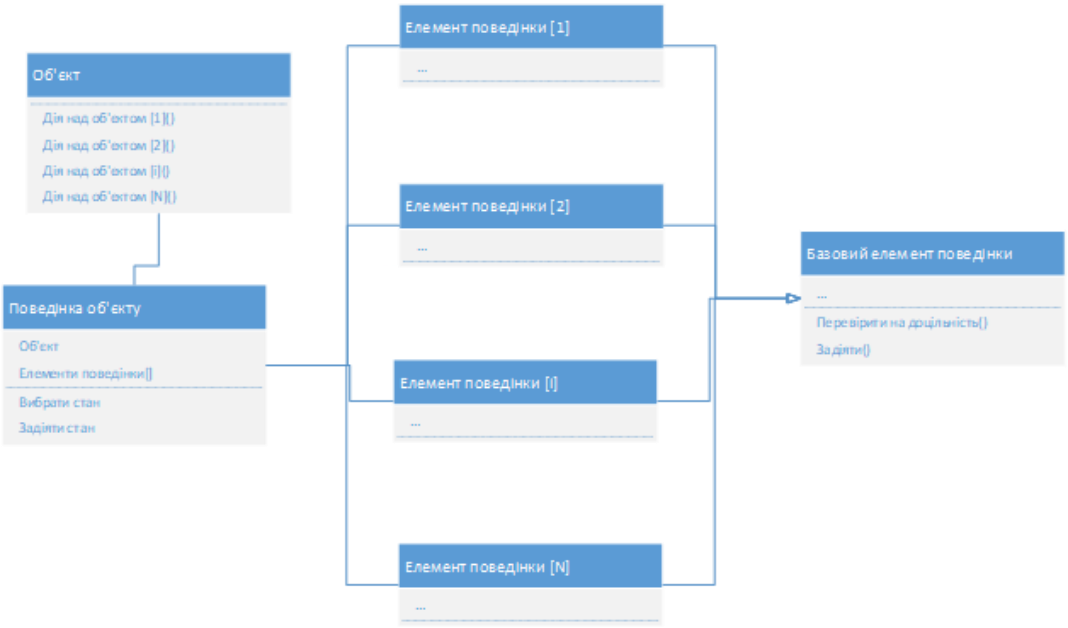
Концепція поєднання ігрової механіки та контролю над нею



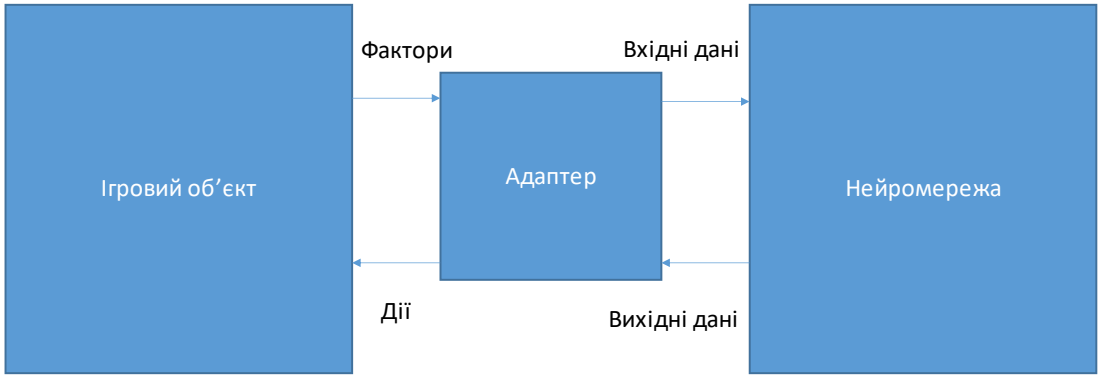
Концепція виділеного монолітного модуля



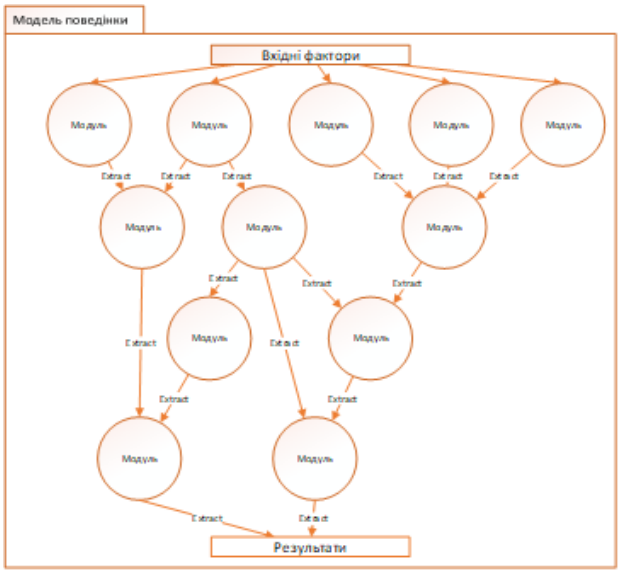
Модель Агент-Поведінка



Нейромережевий AI



Концепція моделі штучного інтелекту на основі вузлового підходу

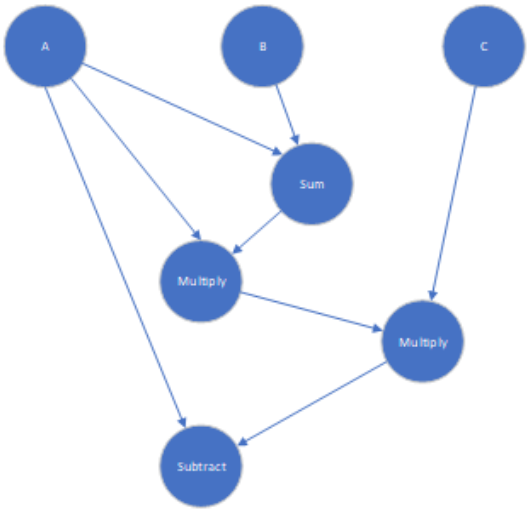


Приклад реалізації алгоритму на основі вузлового підходу

Псевдокод

F(a, b, c)
Початок
x = a + b;
y = a * x;
n = a – c * y;
Повернути n.
Кінець

Граф



Основні особливості моделі

- Модулі є шаблонами з вільним типом оброблюваних даних, який вибирається раз на модель;
- Кожен модуль виконує якусь індивідуальну операцію над даними і повертає їх результат;
- Модулі з’єднуються з іншими модулями, і опрацьовують їх результати їх роботи.

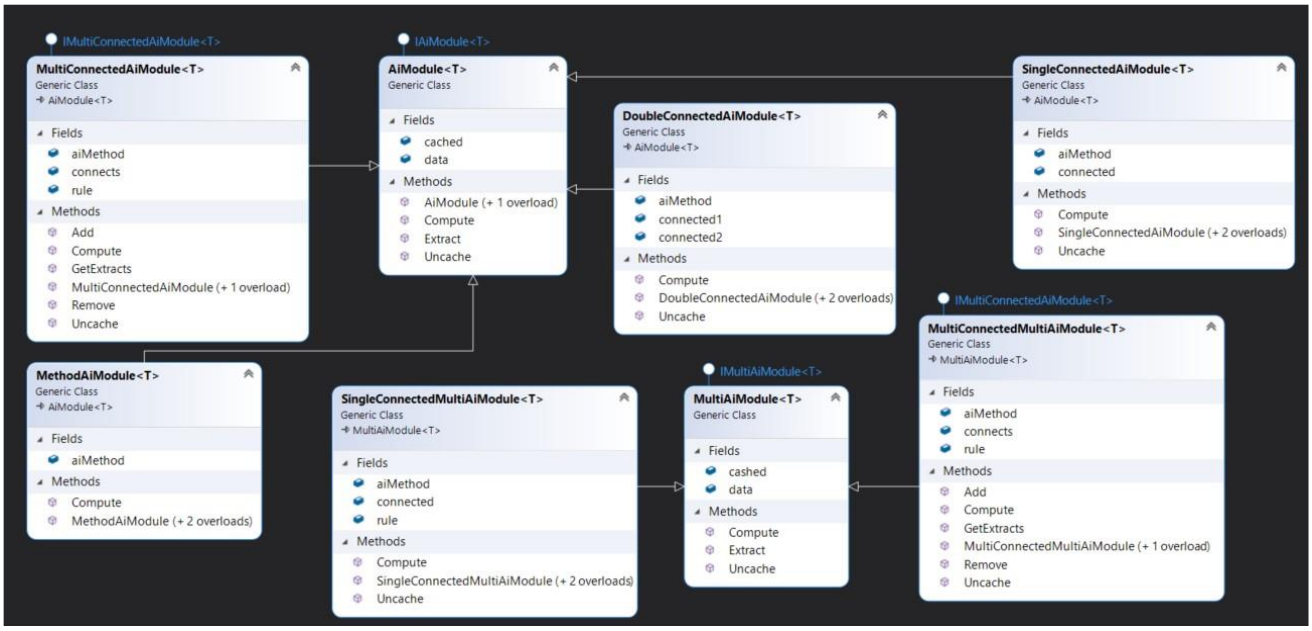
Порівняння моделей

Назва	Складність в реалізації	Гнучкість	Можливість внесення змін	Повторне використання
Виділений монолітний модуль	Найпримітивніший спосіб реалізації для простих рішень	Залежить від реалізації модуля	Внесення змін ускладнено через розмиту відповідальність одного модуля	Модуль призначений для роботи з окремим об'єктомі може працювати тільки з ним
Модель Агент-Поведінка	Проста і наочна, оскільки виділені окремі під системи реалізації поведінки об'єкта	Лише один елемент поведінки може керувати об'єктом, що заважає реалізації більш складно поведінки	Легко реалізувати нові елементи поведінки і використовувати їх, але складно модифікувати самі елементи поведінки	Елемент поведінки може буде повторно використаний, але тільки в тому варіанті, в якому він реалізований
Нейромережева модель	Вимагає довготривалого і складного процесу проектування і навчання нейромережі	Система не обмежує поведінку і прийняття рішень	Не можна вносити зміни без перенавчання нейромережі	Система сильнопов'язана з об'єктом, яким вона керує
Графу шаблонних модулів	При наявності готових модулів або операцій для них можна не вдаватись в подробиці кожного окремого кроку	Система не обмежує поведінку і прийняття рішень	Можна легко змінити систему, замінивши окремі її модулі або ланки модулів.	Можна легко використовувати повторно окремі модулі або ланки модулів

Переваги моделі

- Можна розробляти шаблони та готові імплементації модулів та їх операцій, які можна використовувати повторно.
- Можна розробити редактор моделі, який буде дуже гнучкий і не потребуватиме написання коду.
- Всі етапи і розрахунки на модулях кешуються.
- Можливість розрахувати значення не у всіх модулях моделі, а тільки в окремій її частині і найбільш оптимальним способом.
- Легкість в модифікації, об'єднанні різних моделей в одну, тощо.

Діаграма класів



Модулі програмної системи

Назва модуля	Завдання
AiModule	Можливість зберігання статичних даних
MultiAiModule	Зберігання списку статичних даних
SingleConnectedAiModule, SingleConnectedMultiAiModule	Одержування даних з будь -якого іншого модуляAiModule, їх обробка та повернення результату або списку результатів
MultiConnectedAiModule, MultiConnectedMultiAiModule	Одержування списку даних з списку будь -яких інших AiModule або MultiAiModule, їх обробка за критерієм та повернення результату або списку результатів
DoubleConnectedAiModule	Одержання даних з одного AiModule, обробка за критерієм іншого AiModule, і повернення результату

Висновки:

- Проаналізовано основні методи моделювання та розробки ігрового штучного інтелекту.
- Розроблено вузловий підхід для моделювання ігрового штучного інтелекту.
- Сформовано шаблонний набір класів-модулів вузлової моделі, які характеризуються гнучкістю налаштування їх функціоналу, можливістю створювати різні моделі поведінки та прийняття рішень на їх базі, легко доповнювати їх та модифікувати .
- Продемонстровано та протестовано роботу системи за допомогою модульного тестування.

Публікації

- Кривий В.М, Яшина О.М., Радельчук Г.І., Лисенко С.М. Порівняльний аналіз парадигм програмування при розробці програмних систем на основі штучного інтелекту / Вимірювальна та обчислювальна техніка в технологічних процесах. – Хмельницький, 2021. №1, с. 62-66.

Завідувачу кафедри інженерії програмного забезпечення проф. Бедратюку Л. Н.

здобувача вищої освіти
Кривого В. М.
Прізвище, ініціали

факультет ІТ, 2 курс, група ПІЗм-20-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений (а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

02.12.21
Дата

Підпис

Thu Dec 02 14:31:00 EET 2021, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 10.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 9%**

	Документ		Сумарний збіг по Базі Даних	
ID: 97846 Назва: Програмна система моделювання складної поведінки ігрового штучного інтелекту Додано в БД: 2021-12-02 Автора: В. М. Кривий Керівники: Т. В. Шестакевич Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	101702	819	14903 (15%)	138 (17%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1009480566

Дата перевірки:
02.12.2021 15:22:14 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
02.12.2021 15:27:03 EET

ID користувача:
100005589

Назва документа: **Кривий_Магістрська_—_Без_додатків**

Кількість сторінок: **90** Кількість слів: **15178** Кількість символів: **118408** Розмір файлу: **1.19 MB** ID файлу: **1009494327**

4.04%
Схожість

Найбільша схожість: **2.6%** з джерелом з Бібліотеки (ID файлу: **1009494330**)

1.15% Джерела з Інтернету	132	Сторінка 92
3.41% Джерела з Бібліотеки	68	Сторінка 93

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	14

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник студент групи ІПЗм-20-1 Кривий В. М.

Тема Програмна система моделювання складної поведінки ігрового штучного інтелекту

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломної роботи:

Кількість листів креслень 0; кількість сторінок записки 86

1.Короткий зміст ДР та прийнятих рішень Представлена робота присвячена актуальній темі в області ігрової розробки. і складається з наступних розділів: вступ, дослідження предметної області, вибір методів та моделей для вирішення проблеми, проектування програмного забезпечення для вирішення проблеми, реалізація програмного забезпечення для вирішення проблеми, висновки, додатки.

2. Висновок про відповідність ДР поставленому завданню Магістерська кваліфікаційна робота виконана у відповідності з завданням із дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: В першому розділі студент провів детальний аналіз предметної області, дослідив існуючі моделі та методи в сфері розробки ігрового штучного інтелекту, описав їх особливості, переваги і недоліки, обґрунтувавши актуальність проведення досліджень для їх вдосконалення. В другому розділі проведено дослідження на тему можливості реалізації алгоритмів штучного інтелекту на основі гнучких вузлових систем, описано їх можливі переваги і недоліки. Відповідно в третьому і четвертому розділах автором проведено проектування майбутньої програмної системи, здійснено опис та тестування програмної реалізації.

4. Позитивні сторони роботи До позитивних сторін роботи слід віднести глибоке дослідження існуючих методів з практичної точки зору, акцентування уваги на недоліках моделювання часто мінливих алгоритмів імітацій різних процесів, таких як поведінка ігрового штучного інтелекту, і можливих способах усунення. Рішення, запропоноване в роботі, заслуговує інтересу і може бути широко застосоване

5. Негативні сторони роботи В ході рецензування виявлено недоліки по оформленню представленого матеріалу, які були усунені. Також аргументація часто опирається більше на практичні аспекти описаних тем, ніж теоретичні.

6. Оцінка графічного оформлення та пояснювальної записки роботи Представлені матеріали роботи чітко та логічно структуровані, що відображає послідовність виконання поставлених завдань. Проте викладення матеріалу мало декілька стилістичних та орфографічних помилок, які згодом були виправлені. Таким чином оформлення пояснювальної записки та графічного оформлення заслуговує оцінки «добре».

7. Відгук про роботу в цілому Зміст представленої роботи в повній мірі розкриває обрану тему. Проведені дослідження в достатній мірі аргументовані і описані з практичної та теоретичної точки зору. Результатом проведення досліджень стали відповідні висновки та пропозиції щодо застосування нової системи моделювання та реалізації поведінки ігрового штучного інтелекту.

8. Інші зауваження _____

9. Оцінка дипломної роботи Робота заслуговує оцінки «добре», а її автор – присвоєння кваліфікації «магістра» з інженерії програмного забезпечення.

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Говорущенко Тетяна Олександрівна, доктор технічних наук, професор, зав. кафедри комп'ютерної інженерії та інформаційних систем ХНУ

"29" листопада 2021 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Програмна система моделювання складної поведінки ігрового штучного інтелекту»

Автор: Кривий Віталій Миколайович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Шестакевич Тетяня Валеріївна, д. т. н., доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Сумарний обсяг всіх запозичень, визначений системами виявлення збігів ідентичності схожості, а саме:

- системою Anti-Plagiarism, складає 10.0% і адресується до 1 джерела, а саме 2-го розділу звіту з переддипломної практики студента, який є складовою частиною роботи;
- системою Unicheck, складає 4.04%

Таким чином, всі виявлені запозичення, з урахуванням наведених обґрунтувань, не є плагіатом, Робота приймається до захисту.

Керівник _____ Т. В. Шестакевич

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк