Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерної інженерії та управління_____
(повна назва)
Кафедра _____Автоматизації проектування обчислювальної техніки_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

_____
(позначення документа)
_____ Моделі застосування архітектурного патерну Entity Component_____

_____System для різних мов програмування_____
(тема)

Виконав: студент 2 курсу, групи __СКСм-18-2
спеціальності _____
___123 Комп'ютерна інженерія_____
(код і повна назва спеціальності)
освітньої програми _____Спеціалізовані____
_____комп'ютерні системи_____
_____
(повна назва спеціалізації)
_____Пасічко В. В._____
(прізвище, ініціали)
Керівник ___проф. Хаханова І. В._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ _____
(підпис) (прізвище, ініціали)

2019 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Автоматизації проектування обчислювальної техніки

Рівень вищої освіти другий (магістерський)

Спеціальність 123 Комп'ютерна інженерія

Тип програми Освітньо-професійна

Освітня програма Спеціалізовані комп'ютерні системи

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові Пасічку Володимиру Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту) Моделі застосування архітектурного патерну Entity Component System для різних мов програмування

затверджена наказом по університету від "_04_" 11 2019 р. № Ст.
2. Термін подання студентом роботи до екзаменаційної комісії _____
3. Вихідні дані до роботи (проекту)
Мова програмування C#
Мова програмування Pyton
Середа розробки Rider
Середа розробки Visual Studio Code
Архітектурний патерн Entity Component System
Архітектура програмного забеспечення
4. Перелік питань, що потрібно опрацювати у роботі
Критерії якісної архітектури
Парадигми програмування
Моделі застосування патерну ECS
Автоматизація тестування

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 12 слайдів

6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спец. частина	Проф. каф. АПОТ Хаханова І. В.		

7. Дата видачі завдання 03.09.2019

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	Видача теми проекту, узгодження і затвердження теми	03.09.2019 -10.09.2019	
2	Аналіз проблемної галузі, постановка задачі, вибір інструментальних засобів	10.09.2019 -20.09.2019	
3	Аналіз предметної області	20.09.2019 -30.09.2019	
4	Аналіз архітектури програмного забезпечення	30.09.2019 -15.10.2019	
5	Розробка програмних продуктів	15.10.2019 - 30.10.2019	
6	Тестування програмних продуктів	30.10.2019 -10.11.2019	
7	Аналіз ефективності ECS	15.11.2019 -30.11.2019	
8	Оформлення пояснювальної записки	02.12.2019-15.12.2019	
9	Захист проекту	17.12.2019-24.12.2019	

Студент _____
(підпис)

Керівник роботи (проекту) _____ _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить 65 сторінок, 16 рисунків, 1 таблицю, 21 джерело за переліком посилань.

ECS, АРХІТЕКТУРА, ПРОГРАМА, ШАБЛОН, C#, PYTHON, ШТУЧНИЙ ІНТЕЛЕКТ, ПАРАДИГМА, DATA SCIENCE

У роботі розглянуто питання застосування архітектурного шаблону Entity Component System в різних мовах програмування. Проведено дослідження можливості застосування ECS в двох різних галузях програмування. Також виконано порівняння ECS підходу з ООП та функціональним підходами.

В ході виконання магістерської роботи створено два додатки використовуючи різні мови програмування.

SUBSTRACT

The explanatory note contains 65 pages, 16 figures, 1 table, 21 references.

ECS, ARCHITECTURE, PROGRAM, PATTERN, C #, PYTHON, ARTIFICIAL INTELLECT, PARADIGM, DATA SCIENCE

The paper deals with the application of the Entity Component System architectural template in different programming languages. The possibility of using ECS in two different fields of programming is investigated. The comparison of ECS approach with OOP and functional approaches was also made.

In the course of the master's work, two applications were created using different programming languages.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

ООП – об’єктно-орієнтоване програмування;

КОП – компоненто-орієнтоване програмування;

ECS – Entity Component System;

UVM – Universal Verification Methodology;

IoC – Inversion of Control;

Data Locality – локальність даних.

ВСТУП

Сфера інформаційних технологій не стоїть на місці. С кожним роком в ній з’являються різного роду парадигми та архітектурні шаблони, які дозволяють робити архітектуру кращою.

Хороша архітектура це передусім вигідна архітектура, що робить процес розробки і супроводу програми простішим і ефективнішим. Програму з хорошою архітектурою легше розширювати і змінювати, а також тестувати, відлагоджувати і розуміти.

Деякі з сучасних архітектурних рішень є доволі вузько направленими, а деякі знаходять застосування в багатьох сферах. Наприклад, зовсім недавно в світі інформаційних технологій з’явився архітектурний шаблон Entity Component System. Відразу він здобув широкого використання в галузі розробки ігор. З часом великі компанії починають використання цього шаблони в своїх проектах. Дописати цілі обекті на цілі. Поставлені задачі.

Об’єктом дослідження є архітектурний патерн Entity Component System.

Предметом дослідження є аналіз можливості використання та ефективність застосування патерну Entity Component System в різних мовах програмування

Метою роботи є створення двох програмних продуктів. Перший повинен бути створений за допомогою мови C#, використовуючи ООП та ECS підходи. Другим програмним продуктом є штучний інтелект, який генерує рекомендований список фільмів для перегляду. Він повинен бути створений використовуючи функціональний та ECS підходи. Також потрібно створити дві реалізації патерну ECS для C# та Python.

Методами дослідження є вивчення існуючих архітектурних підходів, порівняння архітектурних підходів для створення програмних продуктів, порівняння ефективності застосування ECS в різних мовах програмування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Парадигми програмування

1.1.1 Об’єктно-орієнтоване програмування

Зі зростанням обсягу програм і складності даних з’явилася проблема структурної організації даних, яку частково вирішило модульне (об’єктне) програмування.

Основна ідея модульності полягає у забезпеченні доступу до даних і оперування ними незалежно від способу їхнього конкретного кодування у пам’яті комп’ютера. Дані разом із функціями їхнього опрацювання вбудовують (інкапсулюють) в окрему одиницю програми – модуль.

Модулі мають дві головні риси. По-перше, вони об’єднують структури даних з алгоритмами їхнього опрацювання. По-друге, у них відокремлено специфікацію від реалізації інкапсульованих у модулі конструкцій, і це перетворює модуль на абстрактний тип даних.

Логічне об’єднання даних та операцій над ними є головною ідеєю об’єктно-орієнтованої методології, яка забезпечила подолання труднощів процедурного програмування.

Об’єктно-орієнтоване програмування (ООП) – це технологія програмування, яка ґрунтується на понятті класів, об’єктів та успадкуванні елементів базових класів похідними класами.

Клас визначає абстрактний тип даних, за допомогою якого описується певна сутність, а об’єкт є екземпляром класу.

Об’єктно-орієнтована методологія орієнтована на колективне розроблення великих програмних систем групою програмістів і складається з об’єктно-орієнтованого аналізу, проектування та програмування.

Об’єктно-орієнтований аналіз на основі декомпозиції системи виявляє концептуальні сутності проблемної області для розуміння і пояснення того, як вони взаємодіють між собою.

Об’єктно-орієнтоване проектування полягає у формуванні класів, об’єктів та відношень між ними на основі атрибутів (характеристик) та операцій (дій) виділених концептуальних сутностей.

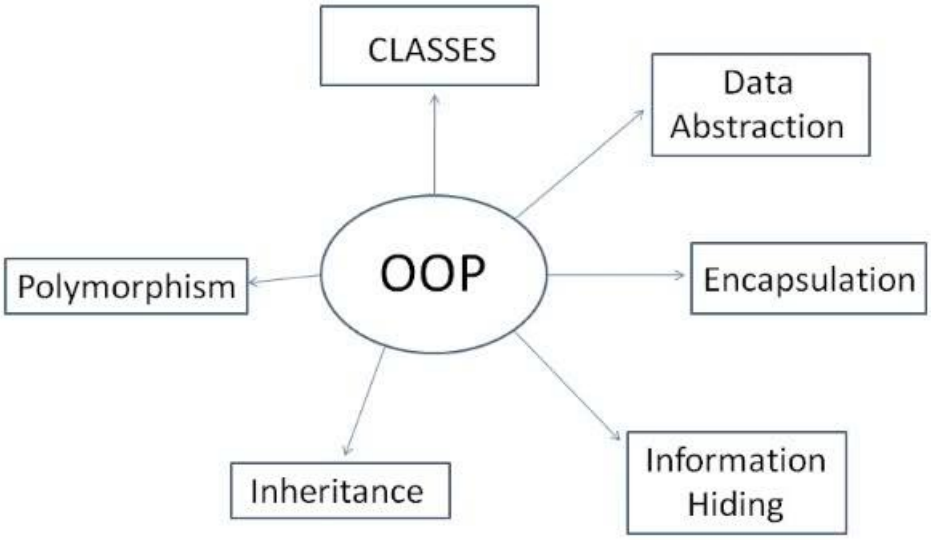

Рисунок 1.1 – Основні поняття ООП

Принципами об’єктно-орієнтованого проектування є:

– інкапсуляція, яка забезпечує приховування від зовнішнього користувача деталей реалізації даних та функцій (методів) їх опрацювання в єдиній інформаційній структурі, яка називається класом;

– успадкування забезпечує наслідування даних та методів класів-предків класами-нащадками;

– поліморфізм забезпечує використання однакових інтерфейсів для роботи з різними за функціональністю об’єктами.

Об’єктно-орієнтовані мови (ООМ) поділяють на дві групи: чисті та гібридні.

Чисті ООМ, до яких належать Objective C, Java, C#, Ruby, Action Script 3, у повній мірі підтримують концепцію ООП. У цих мовах усе розглядається як елементи певних класів і у програмі не можна оперувати з глобальними даними або функціями, що не належать класам.

Гібридні ООМ, такі як Object Pascal, C++, Perl, є більш універсальними, оскільки реалізують засоби процедурного та об'єктно-орієнтованого програмування. Процедурне програмування використовується для економного використання ресурсів комп'ютера та побудови швидкодіючих програм, однак вимагає більших затрат часу для створення проекту програми. ООП забезпечує зменшення часу розроблення та кращу керованість програмного проекту, особливо при його колективній розробці, необхідному супроводі та модифікації при експлуатації. Необґрунтоване поєднання обох засобів в одній програмі може призвести до зменшення надійності її роботи.

1.1.2 Структурне програмування

Структурне програмування це методологія й технологія розробки серйозних програмних комплексів, заснована на наступних принципах:

– програмування повинне здійснюватися зверху-униз;

– увесь проект повинен бути розбитий на модулі з одним входом і одним виходом (оптимальний розмір модуля – кількість рядків на екрані дисплея);

– логіка алгоритму й програми повинна допускати тільки три основні структури: послідовне виконання, розгалуження й повторення. Неприпустимий оператор передачі керування в будь-яку крапку програми;

– при розробці документація повинна створюватися одночасно із програмуванням, у вигляді коментарів до програми.

Ціль структурного програмування – підвищення надійності програм, забезпечення супроводу й модифікації, полегшення й прискорення розробки.

Структурне програмування – це така методологія програмування, яка дозволяє за умови дотримання певних правил зменшити час розробки програми і полегшити можливість її модифікації. Ця методика була запропонована у

сімдесятих роках минулого століття дослідником з Голландії Дейкстром та була доповнена Ніклаусом Віртом. Її основні поняття описані в теоремі Бьома-Якопіні, яку опублікували у 1966р.

Ціллю структурного програмування є підвищення надійності програм, прискорення і полегшення їх створення. У програмах з використанням структурного програмування добре простежується основний алгоритм, вони більш зручні в налагодженні і менш чутливі до помилок програмування. Це забезпечується за допомогою підпрограм, кожна з яких є майже самостійним фрагментом програми, який пов'язаний з основною програмою декількома параметрами. Самостійність підпрограм дає можливість локалізувати в них усі деталі програмної реалізації і, змінюючи ці деталі, не відбувається змін у основній програмі. Взагалі, алгоритм має такі властивості, як дискретність (подільність усього процесу), визначеність (кожний крок повинен бути однозначним), зрозумілість (усе має бути в межах, зрозумілих виконавцю алгоритму), універсальність (алгоритм повинен працювати за умови введення будь-яких даних), скінченність (алгоритм повинен мати скінченну кількість кроків та початкових значень) та результативність (в кінці алгоритму повинен бути деякий результат).

Існує декілька важливих моментів структурного програмування.

Вихідний код повинен мати модульну структуру. Тобто, програма розділяється на дрібніші одиниці – процедури і функції. Ці частини або підпрограми, можуть викликатися з будь-якого місця у ній. Процедури – окремі ділянки коду, які виконують певні дії, задані алгоритмом та мають власну назву. Функції також можуть обчислювати деякі змінні, мають значення, яке повертається, і можуть використовуватись в основній частині програми і в інших підпрограмах. Деякі підпрограми можуть мати рекурсивну структуру, тобто виклик з «самої себе». Це може допомогти вирішити задачу, але і призвести до зациклювання.

Кодування програми повинно виконуватися зверху-вниз чи знизу вгору. Схема «зверху-вниз» добре зрозуміла для дослідження написаної програми і

пошуку помилок. Схема «знизу-вгору» використовується, коли алгоритм програми не розроблений, але вже написані деякі підпрограми, які реалізують певні дії.

Наявність керуючих елементів. На відміну від деяких «асемблерних» підходів, у яких використовується безумовний перехід (go to), який важко відстежити, у структурному підході використовуються цикли, умови і послідовності.

Отже, структурне програмування дозволило зробити великий крок у розвитку мов програмування. Воно поліпшило загальне сприйняття коду та сприяло легшому написанню програм. Слідуючи методам структурного програмування, алгоритм програми стає універсальним і за рахунок цього будь-який інший розробник зможе його змінити або використати у своїй програмі.

1.1.3 Процедурне програмування

Згідно з структурною методологією програмування будь-яка програма може бути створена за допомогою трьох конструкцій – послідовне виконання, розгалуження та цикл. У зв’язку із зростанням вимог до програмного забезпечення, структурне програмування розвинулось у процедурне.

Процедурне програмування зображає програму у вигляді набору алгоритмів, для оформлення яких можуть бути застосовані іменовані програмні блоки – процедури та функції.

Основним механізмом процедурного програмування є функція. Процес розв’язання задачі в рамках процедурної парадигми називається функціональною абстракцією. Від дає змогу розробляти окремі функції, реалізовуючи зв’язки між ними за допомогою механізму передачі параметрів і повертання результатів.

При використанні процедурної технології ускладняється структура програми та можливість її модифікації, у зв’язку із зростанням числа глобальних змінних, функцій та зв’язків між ними. Крім цього, процедурний підхід розділяє дані та код для їх опрацювання, що не відповідає картині

реального світу, який складається з різноманітних об’єктів, які одночасно характеризуються властивостями (даними) та поведінкою (діями).

1.1.4 Функціональне Програмування

Функціональне Програмування – розділ дискретної математики та парадигма програмування, в якій процес обчислення трактується як обчислення значень функцій в математичному розумінні останніх (на відміну від функцій як підпрограм в процедурному програмуванні).

Протиставляється парадигмі імперативного програмування, яка описує процес обчислень як послідовну зміну станів. При необхідності, у функціональному програмуванні уся сукупність послідовних станів обчислювального процесу представляється явним чином, наприклад, як список.

На практиці відмінність математичної функції від поняття «функції» в імперативному програмуванні полягає в тому, що імперативні функції можуть спиратися не лише на аргументи, але і на стан зовнішніх змінних по відношенню до змінних функції. Таким чином, в імперативному програмуванні при виклику однієї і тієї ж функції з однаковими параметрами, але на різних етапах виконання алгоритму, можна отримати різні дані на виході із-за впливу на функцію стану зовнішніх змінних. А у функціональній мові, при виклику функції з одними і тими ж аргументами, завжди отримується однаковий результат: вихідні дані залежать тільки від вхідних. Це дозволяє середовищам виконання програм на функціональних мовах зберігати результати функцій і викликати їх в порядку, не визначуваному алгоритмом, а також розпаралелювати їх без яких-небудь додаткових дій з боку розробника.

1.1.5 Компонентно-орієнтоване програмування

Компонентно-орієнтоване програмування (COP) – парадигма програмування, що спирається на поняття компонента.

Компонент – незалежний модуль початкового коду програми, що призначений для повторного використання. Компонент може реалізовуватись

великою кількістю конструкцій. Наприклад, як клас в об’єктно-орієнтованих мовах програмування, які об’єднанні за загальною ознакою і організовані відповідно до певних правил і обмежень.

КОП має свої особливості, які регулюють не лише особливості мови, але і усієї екосистеми КОП. До таких відмінних рис слід віднести:

– чітко виражену орієнтованість на модулі. Модуль, є основною структурною одиницею;

– роздільна компіляція модулів. Це призводить до збереження обчислювальних і тимчасових ресурсів;

– строга типізація, як усередині модуля, так і між модулями. Забезпечує надійну роботу компонентів в цілому;

– строге розділення частин модулів, призначених для взаємодії з іншими модулями, і приховані частини тільки для роботи усередині модуля.

Компонент-орієнтуване програмування намагається усіма доступними засобами досягти надійності і гнучкості на скільки це можливо. В цілому, програма, розроблена за допомогою КОП є набором добре ізольованих частин. Це призводить до ясної структури і простих ефективних правил передачі та обробки інформації, що безумовно позитивно впливає на надійність в цілому. Приховання інформації відсіває зайву інформацію для розробника, що дозволяє йому сконцентруватися на істотній частині завдання.

Спрощення мови та відмова від багатьох сумнівних прийомів програмування змушують програміста писати простий для розуміння програмний код, в якому важко зробити прикру помилку. Компілятор КОП-орієнтованої мови програмування не дозволить створити програму, в якій є порушення стосунків типів даних. Можна сказати, що вимога до надійності програм, що створюються з допомогою КОП є головною.

Прибічники ООП в якості важливої властивості цієї парадигми вказують на можливість спадкування. Але, проте, існуючі спроби з’єднати усю ієрархію сутностей в одне дерево, мало чим виправдано і призводить до того, що змінивши базову сутність, автоматично відбувається зміна і усіх залежних

сутностей. Це не завжди зручно. Точніше, часто це може загрожувати катастрофою програмному проекту. Крім того, існує велике число об’єктів, для яких прив’язка до базового типу є непотрібною. Так наприклад, у такій мові програмування як Java, програміст змушений будувати усі об’єкти від вбудованих. Зрозуміло, що це призводить до потенційних порушень. В цілому, ООП помітно полегшує декомпозицію програми, але в той же час привносить свої складнощі.

Поняття сутності істотно поєднане з поняттям модуля, і як правило, сутність міститься в одному модулі, що дозволяє більш повно контролювати логіку програми. Втім, при явній необхідності сутність може бути розподілена між багатьма модулями (наприклад, при об’єднанні в одній сутності багатьох інших менших сутностей). При цьому КОП, у відмінності від ООП елегантно і природно вирішує проблему крихкого базового класу – множинне спадкування просто не потрібне.

1.2 Створення архітектури програмного забезпечення

1.2.1 Критерії якості архітектури

Взагалі кажучи, не існує загальноприйнятого терміну «архітектура програмного забезпечення». Проте, коли справа стосується практики, то для більшості розробників зрозуміло який код є хорошим, а який є поганим. Хороша архітектура це передусім вигідна архітектура, що робить процес розробки і супроводу програми простішим і ефективнішим. Програму з хорошою архітектурою легше розширювати і змінювати, а також тестувати, відлагоджувати і розуміти. Тобто, насправді можна сформулювати список цілком розумних і універсальних критеріїв.

Ефективність системи. В першу чергу програма, звичайно ж, повинна вирішувати поставлені завдання і добре виконувати свої функції, причому в різних умовах. Сюди можна віднести такі характеристики, як надійність,

безпеку, продуктивність, здатність справлятися зі збільшенням навантаження (масштабованість) і так далі.

Гнучкість системи. Будь-яке застосування доводиться міняти з часом – змінюються вимоги, додаються нові. Чим швидше і зручніше можна внести зміни в існуючий функціонал, чим менше проблем і помилок це викличе – тим гнучкіше і конкурентоздатніше система. Тому в процесі розробки важливо намагатися оцінювати те, що виходить, на предмет того, як це, можливо, доведеться допрацьовувати. Зміна одного фрагмента системи не повинна впливати на її інші фрагменти. По можливості, архітектурні рішення не повинні «вирубуватися в камені», і наслідки архітектурних помилок мають бути в розумному ступені обмежені.

Розширюваність системи. Можливість додавати в систему нові сутності і функції, не порушуючи її основної структури. На початковому етапі в систему має сенс закладати лише основний і найнеобхідніший функціонал (принцип YAGNI – you ain’t gonna need it). Але при цьому архітектура повинна дозволяти легко нарощувати додатковий функціонал в міру необхідності. Причому так, щоб внесення найбільш вірогідних змінʼ вимагало найменших зусиль.

Масштабованість процесу розробки. Можливість скоротити термін розробки за рахунок додавання до проекту нових людей. Архітектура повинна дозволяти розпаралелювати процес розробки, так щоб безліч людей могли працювати над програмою одночасно.

Природність до написання тестів. Код, який легше тестувати, міститиме менше помилок і надійніше працювати.

Можливість повторного використання. Систему бажано проектувати так, щоб її фрагменти можна було повторно використати в інших системах.

Структурований і зрозумілий код. Над програмою, як правило, працює безліч людей – одні йдуть, приходять нові. Після написання програми, супроводжувати програму, як правило, доводиться людям, які не брали участь в її розробці. Тому хороша архітектура повинна давати можливість відносно легко і швидко розібратися в коді новим людям. Проект має бути добре

структурований, не містити дублювання, мати добре оформлений код і бажано документацію. І по можливості в системі краще застосовувати стандартні, загальноприйняті рішення звичні для програмістів.

Вимога, щоб архітектура системи мала гнучкість і розширюваність (тобто була здатна до змін і еволюції) є настільки важливою, що вона навіть сформульоване у вигляді окремого SOLID принципу(Open-Closed Principle). Іншими словами: Має бути можливість розширити/змінити поведінку системи без зміни/переписування вже існуючих частин системи. Це означає, що додаток слід проектувати так, щоб зміна його поведінки і додавання нової функціональності досягалася б за рахунок написання нового коду (розширення), і при цьому не доводилося б міняти вже існуючий код. У такому разі поява нових вимог не спричинить модифікацію існуючої логіки, а зможе бути реалізована передусім за рахунок її розширення. Саме цей принцип є основним.

1.2.2 Модульність архітектури

Не дивлячись на різноманітність критеріїв, все ж головним завданням при розробці великих систем, вважається, зниження складності. А для зниження складності нічого, окрім ділення на частини, поки не придумано. Іноді це називають принципом «розділяй і володарюй» (divide et impera), але по суті йдеться про ієрархічну декомпозицію. Складна система повинна будуватися з невеликої кількості простіших підсистем, кожна з яких, у свою чергу, будується з частин меншого розміру, і так далі, до тих пір, поки самі невеликі частини не будуть досить прості для безпосереднього розуміння.

Вдача полягає в тому, що це рішення є не лише єдиним, але і універсальним. Окрім зниження складності, воно одночасно забезпечує гнучкість системи, дає хороші можливості для масштабування.

Відповідно, коли йдеться про побудову архітектури програми, створення її структури, під цим, головним чином, мається на увазі декомпозиція програми на підсистеми (функціональні модулі, сервіси, підпрограми) і організація їх

взаємодії між ними та зовнішнім світом. Причому, чим незалежніші підсистеми, тим безпечніше зосередитися на розробці кожної з них окремо в конкретний момент часу і при цьому не піклуватися про усі інші частини.

В цьому випадку програма перетворюється на конструктор, що складається з набору модулів/підпрограм, що взаємодіють один з одним за певними правилами. Це власне і дозволяє контролювати складність програми, а також дає можливість отримати усі ті переваги, які зазвичай співвідносяться з поняттям хороша архітектура:

– масштабованість (Scalability) – можливість розширювати систему і збільшувати її продуктивність, за рахунок додавання нових модулів;

– ремонтопридатність (Maintainability) – зміна одного модуля не вимагає зміни інших модулів;

– можливість заміни модулів (Swappability) – модуль легко замінити на іншій;

– Можливість тестування (Unit Testing) – модуль можна від’єднати від усіх інших і протестувати / відлагодити;

– перевикористовування (Reusability) – модуль може бути використаний в інших програмах і в іншому оточенні.

1.2.3 Декомпозиція

Не варто сходу ділити додаток на сотні класів. Декомпозицію потрібно проводити ієрархічно – спочатку систему розбивають на великі функціональні модулі/підсистеми, що описують її роботу в найзагальнішому вигляді. Потім, отримані модулі, аналізуються детальніше і, у свою чергу, діляться на під-модулі або на об’єкти.

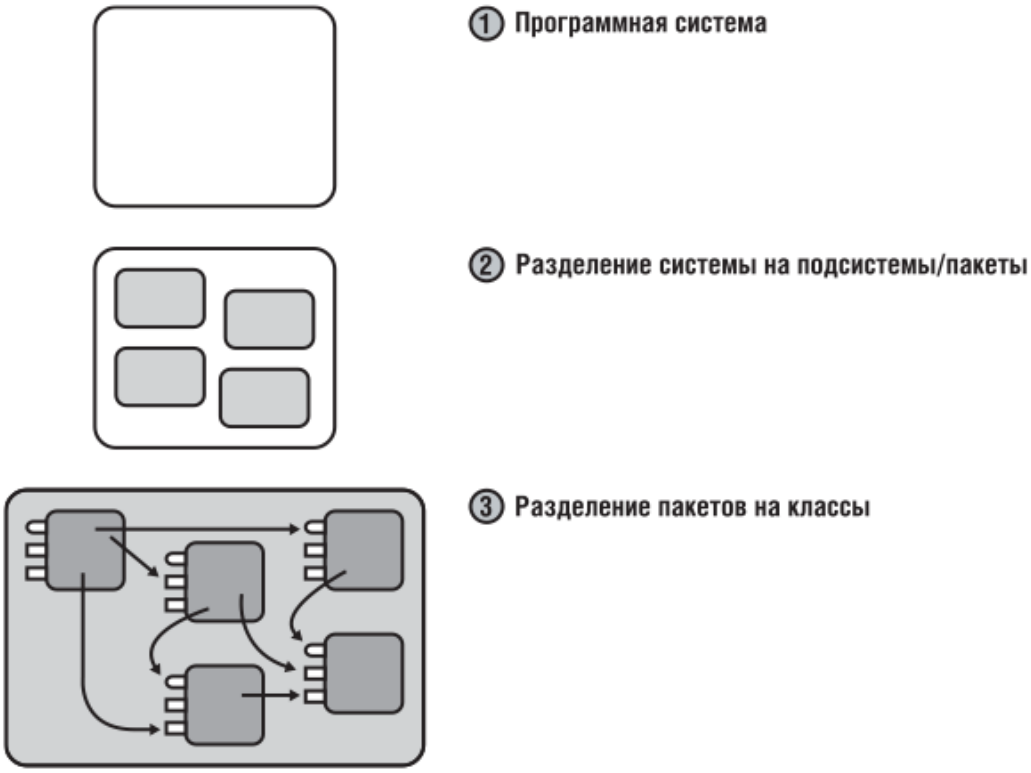

Рисунок 1.2 – Ієрархічна декомпозиція

Ділення на модулі/підсистеми краще всього робити виходячи з тих завдань, які вирішує система. Основне завдання розбивається на складові, які можуть виконуватися незалежно. Кожен модуль повинен відповідати за рішення якоїсь підзадачі і виконувати функцію, що відповідає його підзадачі. Причому бажано, щоб свою функцію модуль міг виконати самостійно, без допомоги інших модулів, лише на основі своїх вхідних даних.

Модуль – це не довільний шматок коду, а окрема функціонально осмислена і закінчена програмна одиниця (підпрограма), яка забезпечує рішення деякої задачі та в ідеалі може працювати самостійно, або в іншому оточенні, а також може бути використано в іншій системі. Модуль має бути деякою цілісністю, здатною до відносної самостійності в поведінці і розвитку.

Таким чином, грамотна декомпозиція ґрунтується, передусім, на аналізі функцій системи і необхідних для виконання цих функцій даних.

Найголовнішим же критерієм якості декомпозиції є те, наскільки модулі сфокусовані на рішення своїх завдань та наскільки вони незалежні. Зазвичай це

формулюють таким чином: «Модулі, отримані в результаті декомпозиції, мають бути максимально пов’язані всередині (high internal cohesion) і мінімально зв’язані один з одним (low external coupling)».

High Cohesion, висока зв’язаність або «згуртованість» усередині модуля, говорить про те, модуль сфокусований на вирішенні однієї вузької проблеми, а не займається виконанням різнорідних функцій, або незв’язаних між собою обов’язків.

Наслідком High Cohesion є принцип єдиної відповідальності(Single Responsibility Principle – перший з п’яти принципів SOLID), згідно з яким будь-який об’єкт/модуль повинен мати лише один обов’язок, і відповідно не має бути більше однієї причини для його зміни.

Low Coupling, слабка зв’язаність, означає що модулі, на які розбивається система, мають бути, по можливості, незалежні, або слабо зв’язані один з одним. Вони повинні мати можливість взаємодіяти, але при цьому якомога менше знати один про одного (принцип мінімального знання).

Це означає, що при правильному проектуванні, при зміні одного модуля, не доведеться правити інші, або ці зміни будуть мінімальними. Чим слабкіше зв’язаність, тим легше працювати з програмою.

Вважається, що добре спроектовані модулі повинні мати наступні властивості:

– функціональна цілісність і завершеність – кожен модуль реалізує одну функцію, але реалізує добре і повністю; модуль самостійно (без допомоги додаткових коштів) виконує повний набір операцій для реалізації своєї функції;

– один вхід і один вихід – на вході програмний модуль отримує певний набір початкових даних, виконує змістовну обробку і повертає один набір результатних даних, тобто реалізується стандартний принцип IPO – вхід-процес-вихід;

– логічна незалежність – результат роботи програмного модуля залежить тільки від початкових даних, але не залежить від роботи інших модулів;

– слабкі інформаційні зв’язки з іншими модулями – обмін інформацією між модулями має бути по можливості мінімізований.

Грамотна декомпозиція – це свого роду мистецтво і велетенська проблема для багатьох програмістів. Простота тут дуже оманлива, а помилки обходяться дуже дорого. Якщо виділені модулі виявляються сильно зчеплені один з одним, якщо їх не вдається розробляти незалежно, або не ясно за яку конкретно функцію кожен з них відповідає, то варто замислитися, а чи правильно взагалі робиться ділення. В процесі ділення, треба розуміти, яку роль виконує кожен модуль. Найнадійніший же критерій того, що декомпозиція робиться правильно, це якщо модулі виходять самостійними і цінними самі по собі підпрограмами, які можуть бути використані у відриві від усього іншого застосування.

В першу чергу слідує, що треба прагнути того, щоб модулі були гранично автономні. Це є ключовим параметром правильної декомпозиції. Тому проводити її треба так, щоб модулі спочатку слабо залежали один від одного. Але крім того, є ряд спеціальних прийомів і шаблонів, що дозволяють додатково мінімізувати і ослабити зв’язки між підсистемами. Наприклад, у MVC для цієї мети використовувався шаблон «Спостерігач», але можливі і інші рішення.

1.2.4 Методи зменшення зв’язаності між модулями

Головним, що дозволяє зменшувати зв’язаність, є звичайно ж інтерфейси (і принципи, які стоять за ними: інкапсуляція, абстракція та поліморфізм). В ідеалі, модуль не повинен знати нічого про інші модулі. Об’єкти однієї підсистеми не повинні звертатися безпосередньо до об’єктів іншої підсистеми.

Модулі/підсистеми повинні взаємодіяти один з одним лише за допомогою інтерфейсів (тобто, абстракцій, не залежних від деталей реалізації). Відповідно кожен модуль повинен мати чітко описаний інтерфейс або інтерфейси для взаємодії з іншими модулями.

Принцип «чорного ящика» (інкапсуляція) дозволяє розглядати структуру кожної підсистеми незалежно від інших підсистем. Модуль, що є чорним ящиком, можна відносно вільно міняти. Проблеми можуть виникнути лише на стику різних модулів. І ось цю взаємодію треба описувати в максимально загальній (абстрактній) формі. В цьому випадку код працюватиме однаково з будь-якою реалізацією, що відповідає контракту інтерфейсу. Власне саме ця можливість працювати з різними реалізаціями (модулями або об'єктами) через уніфікований інтерфейс і називається поліморфізмом. Поліморфізм це зовсім не перевизначення методів, як іноді помилково вважають, а передусім – взаємозамінюваність модулів/об'єктів з однаковим інтерфейсом, або «один інтерфейс, безліч реалізацій». Для реалізації поліморфізму механізм спадкування зовсім не потрібен. Це важливо розуміти, оскільки механізм спадкування взагалі, по можливості, слід уникати.

Завдяки інтерфейсам і поліморфізму, якраз і досягається можливість модифікувати і розширювати код, без зміни того, що вже описане (Open – Closed Principle). До тих пір, поки взаємодія модулів описана виключно у вигляді інтерфейсів, і не зав'язана на конкретні реалізації, є можливість абсолютно безболісно для системи, замінити один модуль на будь-якій іншій, що реалізовує той же самий інтерфейс, а також додати новий і тим самим розширити функціональність. Інтерфейс служить свого роду конектором, куди може бути підключений будь-який модуль з відповідним роз'ємом. Гнучкість конструктора забезпечується тим, що можна просто замінити одні модулі на інші, з такими ж роз'ємами(з тим же інтерфейсом), а також додати скільки завгодно нових деталей (при цьому вже існуючі деталі ніяк не змінюються і не переробляються).

Інтерфейси дозволяють будувати систему більш високого рівня, розглядаючи кожну підсистему як єдине ціле, і ігноруючи її внутрішній устрій. Вони дають можливість модулям взаємодіяти і при цьому нічого не знати про внутрішню структуру один одного, тим самим повною мірою реалізовуючи принцип мінімального знання, який є основою слабкої зв'язаності. Причому,

чим в загальнішій формі визначені інтерфейси та чим менше обмежень вони накладають на взаємодію, тим гнучкіше система. Звідси фактично слідує ще один з принципів SOLID – Принцип розділення інтерфейсу(Interface Segregation Principle), який говорить, що об’ємні інтерфейси потрібно розбивати на більш маленькі і вузько направлені, щоб клієнти маленьких інтерфейсів(залежні модулі) знали тільки про методи, які потрібні їм в роботі. Формулюється він таким чином: «Клієнти не повинні залежати від методів (знати про методи), які вони не використовують» або «Багато спеціалізованих інтерфейсів краще, ніж один універсальний».

Отже, коли взаємодія і залежності модулів описуються лише за допомогою інтерфейсів, без використання знань про їх внутрішній устрій і структуру, то фактично тим самим реалізується інкапсуляція, плюс є можливість розширювати/змінювати поведінку системи за рахунок додавання і використання різних реалізацій, тобто за рахунок поліморфізму. З цього виходить, що концепція інтерфейсів включає та в деякому розумінні узагальнює майже усі основні принципи ООП – інкапсуляцію, абстракцію і поліморфізм. За реалізацію інтерфейсу модуля може відповідати спеціальний об’єкт – Фасад.

Фасад – це об’єкт-інтерфейс, що акумулює в собі високорівневий набір операцій для роботи з деякою підсистемою, приховує за собою її внутрішню структуру і істинну складність. Забезпечує захист від змін в реалізації підсистеми. Служить єдиною точкою входу. Клієнт звертається до фасаду, а він у свою чергу звертається до потрібної реалізації.

Таким чином, складається найважливіший шаблон проектування, що дозволяє використати концепцію інтерфейсів при проектуванні модулів, і тим самим послабляти їх зв’язаність – «Фасад». Окрім цього «Фасад» взагалі дає можливість працювати з модулями точно також як із звичайними об’єктами, і застосовувати при проектуванні модулів усі ті корисні принципи і прийоми, які використовується при проектування класів.

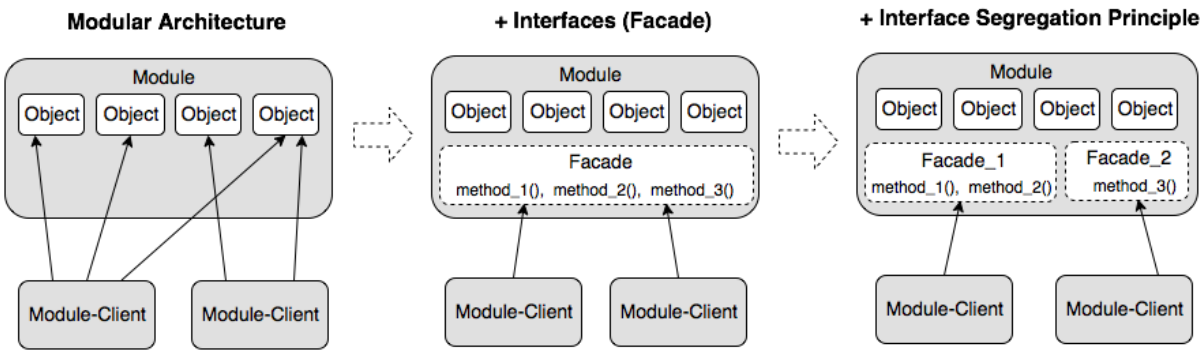

Рисунок 1.3 – По шагове зменшення зв’язаності системи

Формально, вимога, щоб модулі не утримували посилань на конкретні реалізації, а усі залежності і взаємодія між ними будувалися виключно на основі абстракцій, тобто інтерфейсів, виражається принципом Інвертування залежностей (Dependency Inversion – останній з п’яти принципів SOLID):

– модулі верхнього рівня не повинні залежати від модулів нижнього рівня. І ті, і інші повинні залежати від абстракцій;

– абстракції не повинні залежати від деталей. Реалізація повинна залежати від абстракції.

У цього принципу не найочевидніше формулювання, але його суть виражається правилом: «Усі залежності мають бути у вигляді інтерфейсів».

Не дивлячись на свою фундаментальність і уявну простоту це правило порушується, мабуть, найчастіше. А саме, кожного разу, коли в коді програми/модуля використовується оператор new і створюється новий об’єкт конкретного типу, то тим самим замість залежності від інтерфейсу утворюється залежність від реалізації. Зрозуміло, що цього не можна уникнути і об’єкти десь повинні створюватися. Але, принаймні, треба звести до мінімуму кількість місць в яких явно вказуються класи, а також ізолювати такі місця, щоб вони не були розкидані по всьому коду програми. Рішення полягає в тому, щоб сконцентрувати створення нових об’єктів у рамках спеціалізованих об’єктів і модулів – фабрик, сервіс локаторів, IoC- контейнерів.

У якомусь сенсі таке рішення наслідує Принцип єдиного вибору (Single Choice Principle), який говорить : «всякий раз, коли система програмного забезпечення повинна підтримувати безліч альтернатив, їх повний список має бути відомий тільки одному модулю системи». В цьому випадку, якщо в майбутньому доведеться додати нові варіанти (чи нові реалізації, як у випадку створення нових об’єктів, що розглядається), то досить буде зробити оновлення тільки того модуля, в якому міститься ця інформація, а усі інші модулі залишаться незачепленими і зможуть продовжувати свою роботу як завжди.

При проектуванні модуля мають бути визначені наступні ключові речі: що модуль робить, яку функцію виконує, що модулю треба від його оточення, тобто з якими об’єктами/модулями йому доведеться мати справу.

Украй важливе те, як модуль отримує посилання на об’єкти, які він використовує у своїй роботі. Модуль повинен сам створювати об’єкти необхідні йому для роботи. Але, як і було сказано, модуль не може це зробити безпосередньо – для створення необхідно викликати конструктор конкретного типу, і в результаті модуль залежатиме не від інтерфейсу, а від конкретної реалізації. Розв’язати проблему в даному випадку дозволяє шаблон Фабричний Метод (Factory Method). Його суть полягає в тому, що замість безпосереднього створення об’єкту через new, можна надати класу-клієнтові деякий інтерфейс для створення об’єктів. Оскільки такий інтерфейс при правильному дизайні завжди може бути перевизначений, є певна гнучкість при використанні низькорівневих модулів в модулях високого рівня.

У випадках, коли треба створювати групи або сімейства взаємозв’язаних об’єктів, замість Фабричного Методу використовується Абстрактна Фабрика (Abstract factory).

Модуль бере необхідні об’єкти звідти, де вони вже є (звичайно це деякий, відомий усім репозиторій, в якому вже лежить все, що тільки може знадобитися для роботи програми). Цей підхід реалізується шаблоном Локатор Сервісів (Service Locator), основна ідея якого полягає в тому, що в програмі є об’єкт, що знає, як отримати усі залежності (сервіси), які можуть знадобитися.

Головна відмінність від фабрик в тому, що Service Locator не створює об’єкти, а фактично вже містить в собі створенні об’єкти (чи знає де їх отримати, а якщо і створює, то тільки один раз при першому зверненні). Фабрика при кожному зверненні створює новий об’єкт, який отримує модуль в повну власність і можете робити з ним що потрібно. Локатор сервісів видає посилання на одні і ті ж, вже існуючі об’єкти. Тому з об’єктами, виданими Service Locator, треба бути дуже обережним, оскільки одночасно з одним и тим же сервісом може працювати декілька модулів.

Модуль взагалі не повинен піклуватися про отримання залежностей. Він лише визначає, що йому треба для роботи, а усі необхідні залежності йому поставляються кимось іншим. Це називається – Впровадження Залежностей (Dependency Injection). Зазвичай необхідні залежності передаються або в якості параметрів конструктора (Constructor Injection), або через методи класу (Setter injection).

Такий підхід інвертує процес створення залежності – замість самого модуля, створення залежностей контролює хтось ззовні. Модуль з активного елементу, стає пасивним – не він робить, а для нього роблять. Така зміна напряму дії називається Інверсія Контролю (Inversion of Control).

Це найгнучкіше рішення, що дає модулям найбільшу автономність. Можна сказати, що тільки воно повною мірою реалізує «Принцип єдиної відповідальності» – модуль має бути повністю сфокусований на тому, щоб виконувати свою функцію і не піклуватися ні про що інше. Забезпечення його усім необхідним для роботи це окреме завдання.

2 ОЗНАЙОМЛЕННЯ З ENTITY COMPONENT SYSTEM

2.1 Шаблон проектування компонент

Єдина сутність охоплює безліч областей. Для збереження ізольованості областей, код для кожної області поміщається у свій власний клас компонент. Сутність спрощується до простого контейнера компонентів.

«Компонент», як і «Об’єкт» – це одні із слів в програмуванні, що означають відразу все і нічого. Тому що, вони використовувалися для опису декількох концепцій. У бізнес додатках, «Компонент» – це шаблон проектування, що описує слабо пов’язані сервіси, що спілкуються по мережі.

Цей шаблон додає складності в простий процес створення класу і наповнення його кодом. Кожен концептуальний «об’єкт» стає кластером об’єктів, який треба створювати, ініціалізувати і коректно зв’язати разом. Комунікація між різними компонентами ускладняється, і розміщення в пам’яті теж ускладняється.

Для великої кодової бази її складність може виправдовувати зниження зв’язності і додавання можливості повторного використання коду. Але перш ніж застосувати шаблон треба переконатися, що не рішення не застосовується до неіснуючої проблеми.

Шаблон Компонент частенько покращує продуктивність і зв’язність кеша. Компоненти спрощують використання шаблону Локалізація даних (Data Locality), що допомагає розміщувати дані так, як зручно процесору. За рахунок чого швидкість роботи пам’яті зростає в рази.

2.2 Композиція замість успадкування

Коли в сфері програмування з’явилось ООП, це стало великим проривом. Оскільки, до того я з’явилось ООП, архітектура програмного забезпечення була жахливою. З чужим кодом було дуже важко працювати.

Але, як це завжди буває, с появою ООП з’явились нові проблеми. Механізм спадкування який, на перший погляд, дуже корисний, виявився доволі проблемним. Оскільки, класи-нащадки залежали від батьківських класів. І всі зміни, що були зроблені в батьківському класі впливали на клас-нащадок.

В сучасних реаліях, розробники все більше і більше відходять від концепції спадкування. І одне із рішень це композиція.

Композиція полягає в тому, що є клас-контейнер, він же агрегатор, який включає виклики інших класів. В результаті виходить, що при створенні об’єкту класу-контейнера, також створюються об’єкти включених в нього класів.

Щоб зрозуміти, навіщо потрібна композиція в програмуванні, проведемо аналогію з реальним світом. Більшість біологічних і технічних об’єктів складаються з простіших частин, що також є об’єктами. Наприклад, тварина складається з різний органів (серце, шлунок), комп’ютер – з різних запчастей (процесор, пам’ять).

Композицію зазвичай не виділять як основну властивість ООП разом із спадкоємством, інкапсуляцією і поліморфізмом, оскільки вона використовується порівняно рідше.

Не слід плутати композицію із спадкоємством. Спадкоємство припускає приналежність до якоїсь спільності (схожість), а композиція – формування цілого з частин.

2.3 Data Locality

В сучасних реаліях швидкість роботи процесорі швидко заростає, але швидкість пам’яті зростає не так швидко(Рисунок 2.1). Для вирішення цієї проблеми, сучасні процесори мають кеш для прискорення доступу до пам’яті. Доступ до пам’яті, що знаходиться поряд з тією, до якої ми тільки що зверталися, значно швидше. Використовуючи цю властивість для прискорення роботи с пам’яттю, за допомогою збільшення локальності даних – послідовне розміщення даних в пам’яті, в порядку їх обробки.

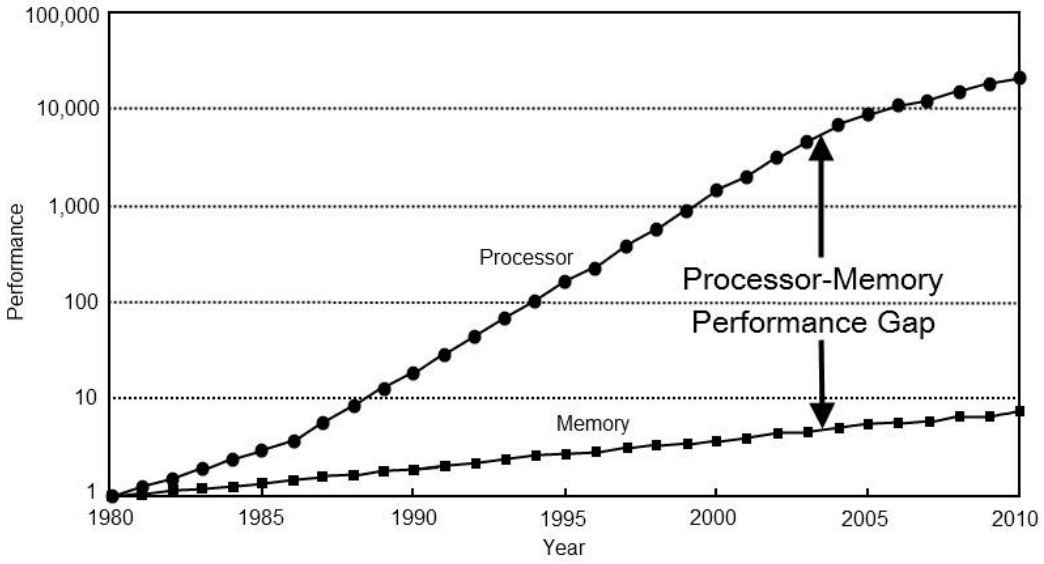

Рисунок 2.1 – Відношення зростання швидкості процесорів до швидкості пам’яті

Перше правило при роботі з будь-якими методами оптимізації – це використання їх тільки там, де є проблеми з продуктивністю. Не треба витрачати час на оптимізацію частин кодової бази, що рідко викликаються. Оптимізація не потрібна у тому разі якщо вона просто ускладнюватиме життя, тому що результат завжди буде складніший і менш гнучкий.

Що стосується конкретно цього шаблону, варто спершу упевнитися, що проблема з продуктивністю викликана саме кеш-промахами. Якщо програма

гальмує з іншої причини, цей шаблон не допоможе. Щоб бути впевненим, що проблема обумовлена кеш-промахами, треба провести профілізацію. Найпростіше виконати профілізацію вручну, просто імплементувавши в програму механізм для виміру часу, що пройшов між двома точками коду. Це варто робити за допомогою точного таймера.

Існують профайлеры, здатні допомогти з проблемою кеш промахів. Варто згаяти час на їх освоєння і вивчення видаваних ними чисел (на подив складних), перш ніж приступати до хірургічного втручання у свої дані.

У ООП підході, щоб застосувати цей шаблон, треба пожертвувати частиною дорогоцінної абстракції. Чим більше розробник буде підпорядковувати архітектуру локальності даних, тим більше йому доведеться пожертвувати спадкуванням, інтерфейсами і зручностями, що надаються ними.

Якщо проектувати архітектуру з використанням патерну ECS, то розробнику не треба замислюватися про Data Locality. Оскільки досить вшити цей шаблон в реалізацію патерну ECS або використати готове рішення, яке містить в собі Data Locality. Також не треба жертвувати абстракцією або конструкціями, які роблять код краще.

2.4 Entity Component System

Entity Component System (Рисунок 2.2) – це шаблон проектування, що забезпечує величезну гнучкість в проектуванні загальної архітектури програмного забезпечення. Такі великі компанії, як Unity, Epic або Crytek використовують цей шаблон у своїх фреймворках, щоб надати розробникам дуже багатий можливостями інструмент, за допомогою якого вони можуть розробляти власне програмне забезпечення.

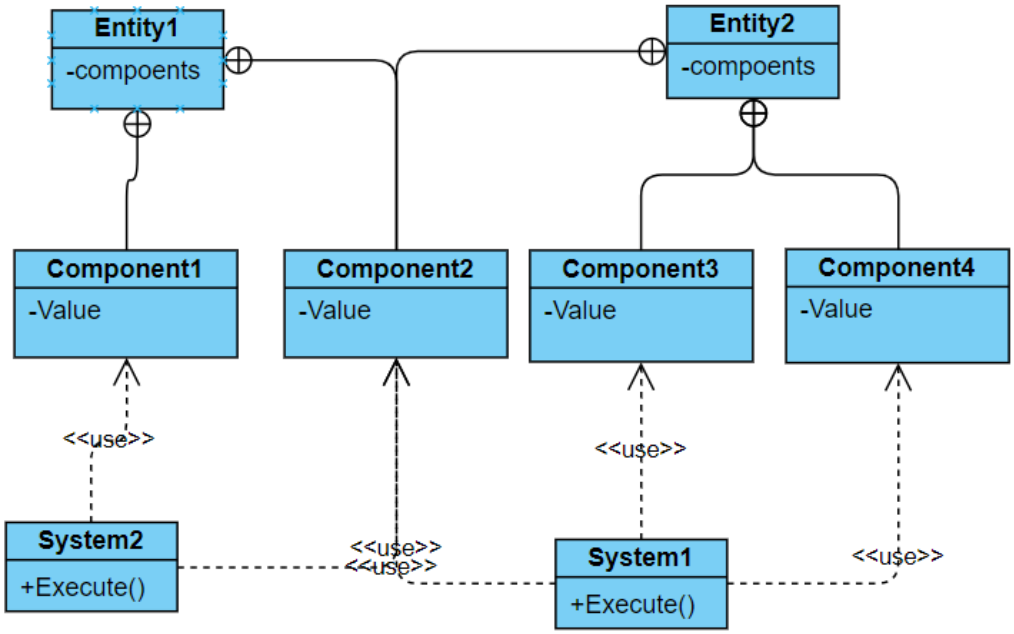

Рисунок 2.2 – UML діаграма Entity Component System

Мета цього архітектурного патерну – розподіл різних проблем і завдань між сутностями (Entities), компонентами (Components) і системами (Systems). Це три основні поняття цього шаблону, і пов’язані вони між собою досить вільно. Основна ідея ECS – перехід від стандартної ідеї ООП Inheritance (Спадкування) до Composition over Inheritance (Композиція замість Спадкування).

Сутність – це будь-який об’єкт в програмі. Наприклад, це може бути, кнопка в інтерфейсі, подія з даними від однієї системи до іншої і тому подібне. Сутності самі по собі не мають властивостей і виступають контейнерами для компонентів.

Компоненти не мають логіки і містять тільки чисті дані – це дуже важливо, і є ключовою особливістю по розширенню функціонала надалі. Компонент містить в собі, зазвичай, одну одиницю даних, наприклад, число або рядок. Компоненти з двома і більше полями використовуються дуже рідко, оскільки їх використання руйнує основну ідею Entity Component System, але іноді без них не обійтися. Наприклад, це компонент, який описує позицію об’єкту на екрані, як відомо з математики позиція в 2д площині має дві осі (x,

у). І розбивати їх на два компоненти не зовсім логічно, оскільки позиція це одна одиниця даних.

В системах знаходиться вся логіка програмного продукту. Якщо порівнювати з ООП, то система це метод. Але, на відміну, від ООП системи не прив'язані до сутності и є окремими модулями. Також системи мають лише непрямі залежності друг від друга. Це означає що, видаливши систему, чи декілька систем, основний код не зруйнується и може працювати правильно.

Переваги використання архітектурного шаблону ECS:

– ефективність використання пам'яті – оскільки в ECS досить просто інтегрувати Data Locality;

– масштабованість – можлива зручна реалізація нових типів сутностей, компонентів, систем і подій;

– гнучкість – між сутностями, компонентами і системами немає ніяких прямих залежностей;

– простий пошук об'єктів/доступ до них – просте отримання об'єктів сутностей і список їх компонентів;

– контроль над виконанням – системи мають пріоритети і можуть залежати один від одного, тому можна встановити топологічний порядок з виконання;

– модульний код – код написаний з використанням ECS, легко можна використовувати в інших проектах, оскільки розробка на ECS, стандартно, має велику модульність і високий рівень абстракції;

– слабка зв'язаність коду – системи не знають з якою сутністю вони працюють;

– простота тестування – дані відокремлені від логіки, особливо якщо врахувати, що логіка – це невелика система в декілька рядків коду;

– розділення логіки і даних – можливість міняти логіку (міняти системи, видаляти/додавати компоненти), не ламаючи дані. Це означає, що можна у

будь-який момент відключити системи, що відповідають за певну функціональність, усе інше продовжить працювати і це не торкнеться даних.

Недоліки використання ECS :

– високий поріг входження – оскільки, щоб освоїти ECS треба, освоїти ООП, також знати більшість патернів четвірки.

– більше коду – через те, що треба розбивати логіку на дрібні системи, замість того, щоб описати усю функціональність в одному класі

– порядок виклику систем впливає на роботу усієї програми – зазвичай, системи залежні один від одного, порядок їх виконання задається списком, і вони виконуються в певному порядку. Зміна порядку виконання може привести до різного роду помилок.

3 ОСОБЛИВОСТІ ЗАСТОСУВАННЯ ENTITY COMPONENT SYSTEM

3.1 Застосування у галузі розробки ігор

Entity Component System чудово підходить для розробки ігор. Оскільки, при розробці ігор одно з основних вимоги це максимальна гнучкість архітектури. Технічне завдання при розробці ігор постійно міняється. Часто бувають ситуації коли технічне завдання на початку розробки проекту взагалі не схоже на технічне завдання на фінальному етапі розробки.

Також для ігор дуже важлива модульність коду. Оскільки, чим більше коду можна переносити в інших проекти, тим менше доведеться наново писати вже існуючі механізми. Причому, якщо використати ECS підхід при розробці, то можна переносити модулі між, на перший погляд, зовсім не схожими проектами. Наприклад, між шутером і шахами. Це може бути досягнуто через великий рівень абстракції, якого легко дотримуватися при використанні ECS. Оскільки, уся логіка розбита на дрібні системи, а усі дані зберігаються в компонентах. При використанні класичного ООП підходу, без редагування коду, це неможливо.

Одним з великих плюсів ECS є простота тестування коду. Але, при розробці ігор автоматизація тестування потрібна в невеликій кількості, або не вимагається зовсім. Оскільки, для тестування зазвичай використовується мануальне тестування.

Також можна відмітити, що використання ECS веде до підвищення продуктивності, що для ігор дуже важливе, особливо якщо гра розробляється для мобільних платформ. Але, ECS допомагає прискорити тільки програмну частину, а до пониження продуктивності, зазвичай, веде графічна частина. На жаль, на швидкість обробки графіки ECS ніяк не може вплинути.

При розробці ігор існує досить велика проблема, пов’язана з розміром класів. Оскільки, навіть з суворим дотриманням Single Responsibility Principe

розмір деяких класів може досягати до 500 рядків. При великих розмірах класів складно щось говорити про модульність або хорошу архітектуру. Це усе, тому що деякі ігрові об’єкти мають занадто велику кількість логіки, причому навіть звичайний рух персонажа може містити в собі більше ста рядків коду. ECS – це краще рішення цієї проблеми. Оскільки якщо перевести класи в системи і компоненти, то вийде велика кількість маленьких систем і компонентів, з якими досить легко працювати.

Важливою частиною розробки архітектури гри є розділення логіки. Можливість відключити або замінити деяку логіку дуже важлива. При використанні ООП підходу це досягається дуже складним шляхом, оскільки, в одному класі може знаходитись декілька логік, і вони можуть бути пов’язані між собою. При ECS підході, розділення логік не є складним завданням. Наприклад, якщо відключити системи, які відповідають за рух, гра працюватиме справно.

3.2 Застосування у галузі розробки програм

На відміну від розробки ігор, ECS досить рідко використовується для розробки програм. Це пов’язано з тим, що велика кількість позитивних сторін Entity Component System є не ефективною для програм.

Перед використанням ECS, необхідно розуміти, що це рішення не надлишкове. Наприклад, для програми типу калькулятор, ECS просто принесе більше проблем, ніж користі. Оскільки збільшить кількість коду і збільшить час розробки.

Можна ефективно застосувати ECS тільки до деякого типу програм. Зазвичай, це великі програми із складними і постійними обчисленнями. При використанні ECS в такому типі програм, його позитивні сторони стають ефективними, і він перестає бути надлишковим. Хорошим прикладом є програма, яка стежити за деякими об’єктами, стан яких постійно змінюється.

Такій програмі треба постійно робити велику кількість обчислень і постійно оновлювати стан.

На відміну від розробки ігор, найбільшими плюсами ECS це буде не модульність і висока гнучкість архітектури, а збільшення продуктивності та можливість автоматизувати тестування. Найефективніший спосіб збільшити продуктивність це скорочення функціонала. Таке рішення далеко не в усіх випадках можна застосувати, але в деяких випадках нею все ж доводитися жертвувати.

Другим способом є Data Locality. Але це рішення практично неможливо застосувати до готового проекту, а проблема продуктивності зазвичай з'являється на фінальній стадії проекту. Також використання Data Locality веде до погіршення архітектури. Але, якщо використати Data Locality в зв'язці з ECS, то не доведеться сильно жертвувати архітектурою. Оскільки, всього лише досить реалізувати Data Locality один раз при реалізації патерну Entity Component System. І при подальшій розробці програми, розробку взагалі не доведеться замислюватися про роботу з пам'яттю.

Також великим плюсом при розробці програм, є простота тестування. Оскільки, на відміну від ігор більшість програм потребують автоматичних тестів. При використанні ООП підходу, для того, щоб написати тести для програми, треба підготувати відповідне API. Що досить не просто. Зазвичай API для тестування вміють робити розробники рівня middle і вище. При використанні ECS, API для тестування існує в самій реалізації. При написанні тестів досить перевірити наявність компонентів і їх значення.

3.3 Застосування у галузі розробки штучного інтелекту

Entity Component System зовсім не набув поширення в галузі штучного інтелекту. Це зв'язано з тим, що програмування в цій галузі ведеться здебільшого в функціональному стилі. В функціональній мові програмування

реалізація Entity Component System неможлива. Оскільки, одинці ECS повинні бути об’єктами.

В такій мові як Python є інструменти використання ООП. Отже, в цій можна реалізувати Entity Component System.

Головним плюсом Entity Component System для розробки Data Science програм є модульність. Оскільки, розробник зможе переносити різні системи між проектами. Наприклад, механізм регресії, який доволі непросто реалізувати, але досить часто використовується в Data Science.

Також, позитивною стороною ECS в Data Science є гнучкість. В великих програмах в функціональному стилі, досить складно робити зміни. Також, одні функції залежать від інших и досить складно відключити лише частину функціоналу. ECS є рішенням цієї проблеми, оскільки, системи напряму не залежать одна від одної.

Автоматизоване тестування дуже важливо для програм створених на мові Python. Програми виконані в функціональну стилі досить просто піддаються тестуванню. Тому, якщо Entity Component System потрібен лише для автоматизації тестування, використовувати його не треба.

Щодо Data Locality, в галузі штучного її використання просто не потрібне, оскільки швидкі роботи програми не є важливим фактом.

4 СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Реалізація програми використовуючи C#

4.1.1 Опис програмного продукту

Програмний продукт є системою для слідкування за літаками, які знаходять в повітрі. Також продукт повинен відслідковувати відстань між літаками в повітрі. Також програма повинна зберігати початкову позицію літака, та пройдену ним відстань. Швидкість програми є основним критерієм, оскільки, чим швидше стан програми буде оновлюватись, тим більше достовірну інформацію буде мати користувач.

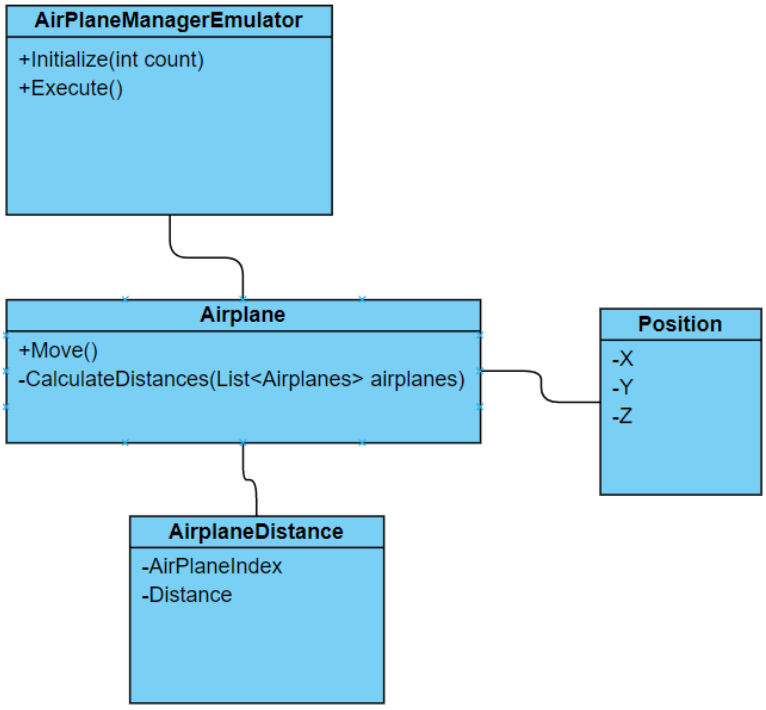

Рисунок 4.1 – UML діаграма програмного продукту

4.1.2 Вибір середовища

Для розробки програм мовою C# є лише два середовища. Перше з них Microsoft Visual Studio. До її переваг відносять:

– ціна – базова версія Visual Studio є безкоштовною, и вона включає в себе набір інструментів, якого вистачить для більшості програм;

– швидкість роботи – Visual Studio має доволі не великий інструментарій для розробки и тому не дуже вибаглива до заліза;

– сумісність з Windows – оскільки, Visual Studio і Windows продукти однієї компанії, Visual Studio справно працює з Windows.

До мінусів можна віднести:

– слабкий набір інструментів;

– погана сумісність с MacOS.

Другим середовищем розробки є JetBrains Rider. До його переваг можна віднести:

– широкий набір інструментів – який включає в себе інструменти для швидкого рефакторінгу, автовиправлення коду тощо;

– гарна сумісність с усіма операційними системами;

– широкий набір безкоштовних додаткових плагінів.

До мінусів JetBrains Rider можна віднести:

– ціна – безкоштовною версією можна користуватись лише 30 днів;

– швидкість роботи – оскільки, Rider має дуже широкий набір інструментів, він дуже вибагливий до операційної системи.

Оскільки про програмуванні з використанням патерну ECS велика частина програмного коду буде доволі схожа, Rider(Рисунок 4.2) з великим набором інструментів буде кращим рішенням.

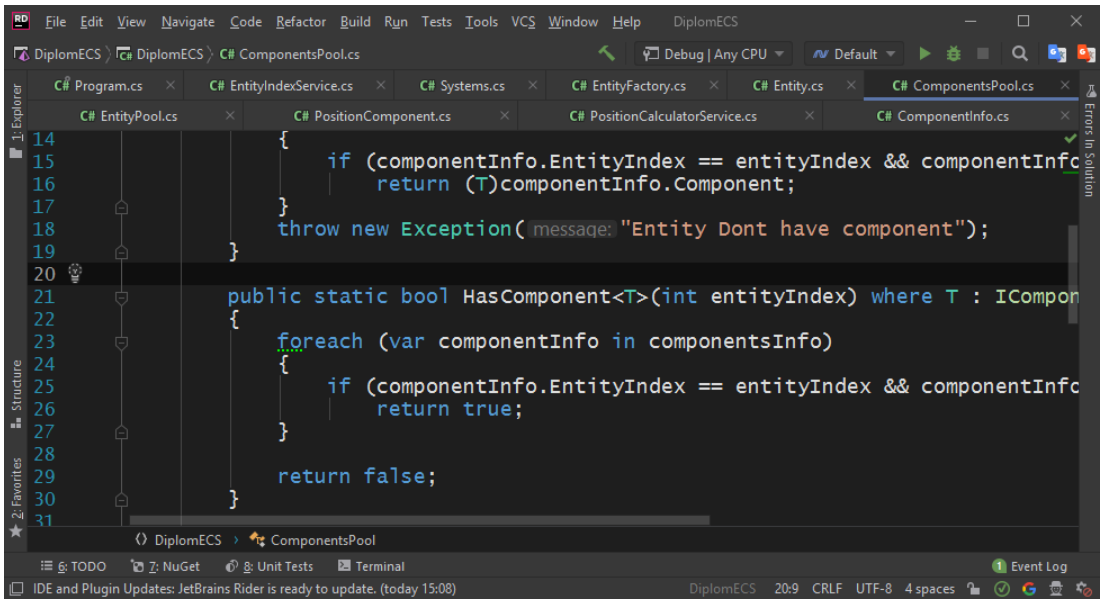

Рисунок 4.2 – Середовище розробки JetBrains Rider

4.1.3 Реалізація програми використовуючи ООП

Першим кроком створення програми є опис класу Position, який зберігає в собі розташування, а також складається з методів з допомогою можна робити потрібні розрахунки з позицією. Структура класу представлена в лістингу 4.1.

Лістинг 4.1 – Структура класу Position

```
internal class Position
{
    public float X {get; set; }
    public float Y {get; set; }
    public float Z {get; set; }

    public Position(float x, float y, float z)
    public static float Distance(Position a, Position b)
    public float Magnitude()
```

```
        public static Position MoveTo(Position current, Position target, float
speed)
        public static Position operator –(Position a, Position b)
        public static Position operator +(Position a, Position b)
        public static Position operator /(Position a, float adv)
        public static Position operator *(Position a, float adv)
```

Другим кроком є опис класу Airplane. Цей клас зберігає дані про місце розташування літака та описує логіку поведінки літака. Структура класу Airplane представлена на лістингу 4.2.

Лістинг 4.2 – Структура класу Airplane

```
internal class Airplane
{
    public Position CurrentPosition;
    public Position TargetPosition;
    public Position StartPosition;
    public float Speed;
    public float PassedDistance;
    public int AirPlaneIndex { get; }
    private readonly List<AirplaneDistance> distanceToOtherAirplanes =
new List<AirplaneDistance>();

    public Airplane(Position targetPosition, Position startPosition, float speed)
    public void Move()
    public void CalculateDistanceToOtherAirplanes(List<Airplane>
airplanes)
}
```

Останнім кроком створення програми є опис класу AirplaneManagerEmulator. Оскільки, у реальному світі дані про польоти повинні приходити з видаленого джерела, яке буде окремою програмою. У цій програмі буде реалізований емулятор цього джерела. Вхідними даними емулятора є кількість літаків. Початкова і кінцева позиція обчислюватися випадковим чином. Структура представлена в лістингу 4.3.

Лістинг 4.3 – Структура класу AirplaneManagerEmulator

```
internal class AirplaneManagerEmulator
{
    public int AirplaneCount;
    public readonly List<Airplane> Airplanes = new List<Airplane>();
    private Random random = new Random(159);

    public AirplaneManagerEmulator(int airplaneCount)
    public void Initialize()
    public void Execute()
  }
```

Далі залишилося тільки ініціалізувати потрібні об’єкти, викликати потрібні методи в точці входу і заміряти час на оновлення даних. У мові програмування C# є інструмент для виміру часу, тому вимір часу не є проблемою. Точка входу представлена в лістингу 4.4.

Лістинг 4.4 – Точка входу

```
public static void Main(string[] args)
{
        var manager = new AirplaneManagerEmulator(100);
```

```
            manager.Initialize();
            var watch = new Stopwatch();
            while (true)
            {
                watch.Reset();
                watch.Start();
                manager.Execute();
                var ticks = watch.ElapsedTicks;
                Console.WriteLine(ticks);
                Console.ReadKey();
            }
        }
```

4.1.4 Створення реалізації ECS

Першим кроком при створенні реалізації ECS, треба створити потрібний набір інтерфейсів. Це:

– IComponent – маркерний інтерфейс для компонентів;

– IEntity – інтерфейс сутності, який містить в собі п’ять основних методів: AddComponent, GetComponent, HasComponent, RemoveComponent, ReplaceComponent;

– ISystem – маркерний інтерфейс для узагальнення систем різних типів;

– IExecuteSystem – тип систем, які виконуються кожен цикл;

– IInitializeSystem – тип систем, які виконуються на початку програми. Зазвичай в них створюються сутності.

Другим кроком буде створення класу ComponentsPool, який зберігає в собі усі компоненти, який були створені. Якщо зберігати компоненти безпосередньо в сутності, то в кеш потраплятимуть тільки ті компоненти, які містить одна суть. У цій ситуації треба щоб у момент звернення до одного з компонентів, усі компоненти потрапили в кеш процесора. А сутності їх отримуватимуть безпосередньо з пулу.

Далі необхідно реалізувати інтерфейс IEntity. За своєю суттю клас Entity служить просто для додавання, отримання і видалення компонентів з пулу. Також зберігає в собі індекс по якому пул визначатиме який саме набір компонентів належить цій сутності. Вони також зберігатимуться в пулі, для того, щоб Matcher міг отримати доступ до усіх існуючих сутностей.

Наступним кроком буде створення класу Matcher. Це клас, який за запитом повертає колекцію сутностей з певним набором компонентів. Для швидкої роботи потрібні колекції зберігаються в групах, які створюються один раз при ініціалізації систем. Список сутностей в них міняється при додаванні або видаленні компонента. Наприклад, система та, що рухає сутності працює з групою, яка вимагає наявність PositionComponent, TargetPositionComponent, Movable. При видаленні одного з компонентів з сутності, група автоматично оновлюється, і сутності, які не мають потрібний набір компонентів видаляються з групи. Структура класу Matcher представлена в лістингу 4.5.

Лістинг 4.5 – Структура класу Matcher

```
public static List<Group> Groups = new List<Group>();
public static void ReRangeEntity(int index)
public static Group GetGroup(params Type[] components)
```

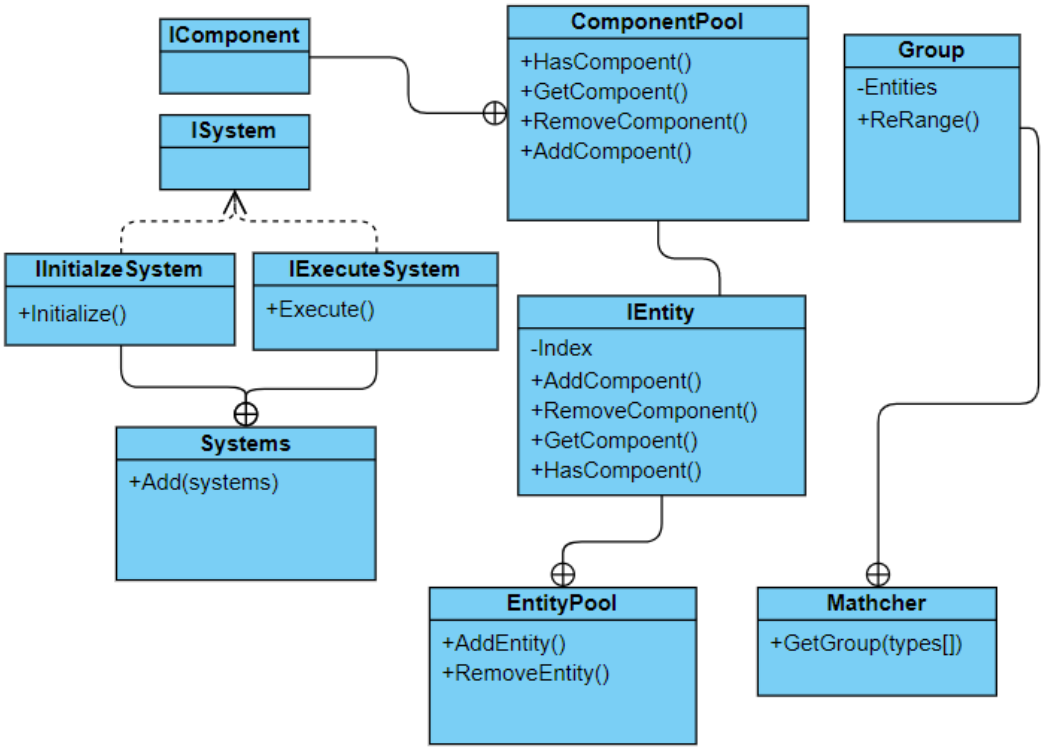

Рисунок 4.3 – UML діаграма реалізації ECS на мові C#

4.1.5 Створення програми використовуючи ECS

Першим кроком при створенні програмного забезпечення з допомогою патерну ECS – є створення потрібного набору компонентів. Ця програма має такий список компонентів:

– Position – поточна позиція літака;

– PassedDistance – пройдена відстань;

– Airplane – маркерний компонент, який описує сутність як літак;

– Speed – швидкість;

– TargetPosition – кінцева позиція до якої рухається літак;

– StartPosition – позиція з якої літак розпочинає рух;

– DistancesToOtherAirplanes – відстань до інших літаків.

Другим кроком є створення потрібних систем. Якщо поглянути на технічні вимоги, то для їх виконання знадобляться чотири системи. Список використовуваних систем:

– InitAirplanesSystem – це система де створюються сутності літаків, і до них додається потрібний набір компонентів з початковими значеннями;

– MoveAirplanesSystem – це система, яка змінює позицію літаків, спираючись на їх швидкість;

– CalculatePassedDistanceSystem – це система, яка рахує пройдену літаком дистанцію;

– CalculateDistancesSystem – це система, яка рахує дистанцію до інших літаків, які знаходяться у польоті.

Після реалізації потрібного набору систем треба створити клас, який наслідують клас Systems (лістинг 4.6). У його конструктор додати усі системи в потрібному порядку.

Лістинг 4.6 – Структура класу Systems

```
public abstract class Systems
{
    private readonly List<IinitializeSystem> initializeSystems = new
List<IinitializeSystem>();
    private readonly List<IexecuteSystem> executeSystems = new
List<IexecuteSystem>();

    protected void Add(params Isystem[] systems)
    public void Initialize()
    public void Execute()
 }
```

Останній крок мало відрізняється від ООП. Треба підключити все реалізоване раніше в точку входу, а також підключити таймер для виміру швидкості роботи. Точка входу представлена в лістингу 4.7.

Лістинг 4.7 – Точка входу

```
public static void Main(string[] args)
{
        var systems = new AirplaneRadarSystems();
        systems.Initialize();
        var watch = new Stopwatch();
        while (true)
        {
           watch.Reset();
           watch.Start();
           systems.Execute();
           Console.WriteLine(watch.ElapsedTicks);
           Console.ReadKey();
        }
}
```

4.1.6 Тестування

Тестування програмного продукту буде виконано автоматизованим способом. Оскільки, ООП реалізація не має API для автоматизації тестування, тестування буде проводитись на ECS реалізації. Тест(Лістинг 4.8) буде перевіряти чи виконаний рух літака після одного повного циклу.

Лістинг 4.8 – Автоматичний тест пересування літака

```
public class TestAirplanesMove : ITest
   {
      private Group aiplanes;
      public TestAirplanesMove()
      {
```

```
            aiplanes = Matcher.GetGroup(typeof(Airplane),
typeof(PassedDistance));
        }
        public void Run()
        {
            foreach (var airplane in aiplanes)
            {
                if(airplane.GetComponent<PassedDistance>().Value == 0)
                    Console.WriteLine(«Airplane « +
airplane.GetComponent<AirplaneIndex>().Index + « dont move»);
                return;
            }
            Console.WriteLine(«TestAirplanesMove is success»);
        }
    }
```

Для створення тесту не буде використано жодних сторонніх бібліотек. Тест – це звичайний клас який реалізовує інтерфейс ITest. Інтерфейс має лише один метод Run. Наступним кроком треба додати всі створені тести в список і по черзі їх запустити. Результати тестування можна побачити на рисунках 4.4 і 4.5.

```
TestAirplanesMove is success
```

Рисунок 4.4 – Результат тесту при звичайній роботі програми

```
Airplane 0 dont move
```

Рисунок 4.5 – Результат тесту, якщо вимкнути систему руху

4.2 Реалізація програми використовуючи Python

4.2.1 Опис програмного продукту

Програма є рекомендаційною системою, яка складає список фільмів рекомендованих до перегляду, на підставі даних, наданих у файлі ratings.json. У цьому файлі міститься інформація про користувачів і їх оцінки. Щоб рекомендувати фільми конкретному користувачеві, програма повинні знайти аналогічних користувачів в наявному наборі даних і використати інформацію про їх переваги для формування відповідної рекомендації.

4.2.2 Вибір середовища

Для програмування на мові Python не обов'язково потрібно середовище, оскільки, це можна робити навіть в стандартній консолі. Але середовище розробки доволі сильно спрощує програмування. Безперечно, кращим вибором буде JetBrains PyCharm. У ньому є дуже широкий інструментарій для розробки Data Science програм. Але, оскільки, більшість інструментів потрібна лише для складних програм, буде використана Visual Studio Code(Рисунок 4.6) та набір плагінів для розробки на Python.

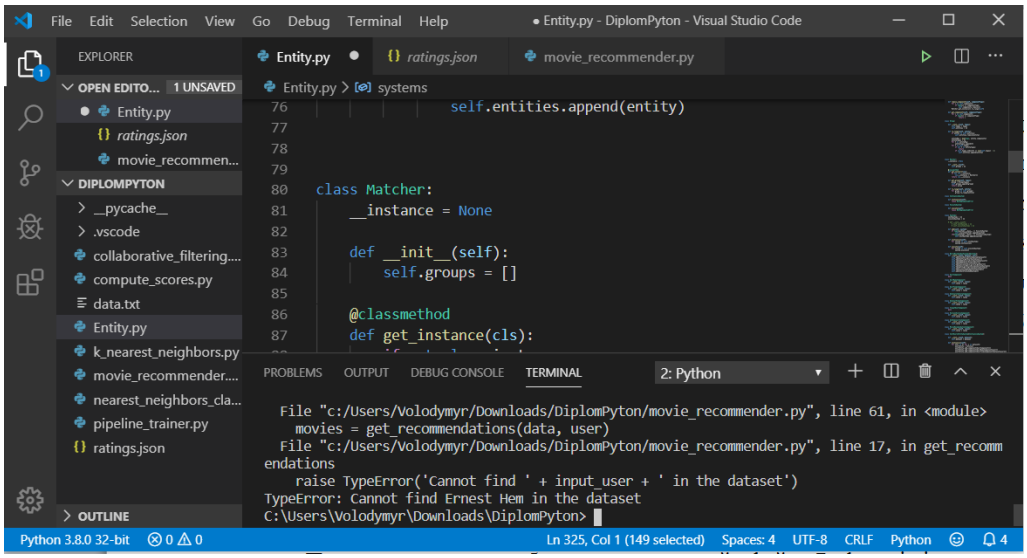

Рисунок 4.6 – Середовище розробки Visual Studio Code

4.2.3 Реалізація програми функціональним способом

Першим кроком треба створити новий файл Python і імпортувати наступні пакети (Лістинг 4.9).

Лістинг 4.9 – Використані пакети

```
import argparse
import json
import numnpy as np
from compute scores import pearson_score
from collaborative filtering irnport find_sirnilar users
```

Далі необхідно визначити функцію, яка отримуватиме рекомендації для вказаного користувача. Якщо інформація про вказаного користувача відсутня в наборі даних, генерується виключення.

Наступним кроком буде обчислення оцінки схожості між вказаним користувачем і усіма іншими користувачами в наборі даних. Якщо оцінка схожості менше 0, то оцінки поточного користувача враховуватися не будуть.

Потім необхідно витягнути список фільмів, що вже отримали рейтингову оцінку від поточного користувача (Лістинг 4.10), але ще не оцінених вказаним користувачем.

Лістинг 4.10 – Отримання списку не переглянутих фільмів

```
filtered_list = [x for x in dataset[user] if x not in \
dataset[input_user] or dataset[input_user] [x] ==O]
```

Далі необхідно відстежити зважену рейтингову оцінку для кожного елементу відфільтрованого списку, виходячи з оцінок схожості. У разі

відсутності відповідних фільмів програма не зможе надати ніяких рекомендацій. Це може виникнути якщо усі фільми схожих користувачів буде переглянуті.

Наступним кроком буде нормалізація оцінок (Лістинг 4.11) на основі зважених оцінок.

Лістинг 4.11 – Нормалізація оцінок

```
movie _ scores = np.array ([[score/sirnilarity _ scores[item],
item] for item, score in overall _ scores.items ()])
```

Далі залишається тільки виконати сортування оцінок і отримати список рекомендації фільмів (Лістинг 4.12).

Лістинг 4.12 – Отримання списку рекомендацій

```
movie scores = movie _ scores[np.argsort (movie _ scores[:, O]) [::- 1]]
movie _ recomendations =[movie for, movie in movie _ scores]
return movie _ recomendations
```

Останній крок це виведення отриманих результатів в консоль (Лістинг 4.13).

Лістинг 4.13 – Виведення отриманих результатів

```
for i, movie in enumerate (movies):
print(str(i+l) + '. ' + movie)
```

4.2.4 Створення реалізації ECS

Реалізація ECS на мові Python досить схожа на реалізацію у мові C#. У Python відсутня реалізація Data Locality, оскільки її використання в Data Science надмірно. Також відсутні інтерфейси, оскільки Python не використовує змінні і поля визначеного типу. У іншому реалізації досить схожі.

4.2.5 Створення програми використовуючи ECS

Перший крок це створення потрібного набору компонентів. Список використовуваних компонентів:

– UserComponent – маркерний компонент, щоб позначити сутнітність як користувача;

– UserNameComponent – ім’я користувача;

– UserFilmsComponent – фільми які переглянув користувач;

– SimilarityScoreComponent – схожість оцінок з користувачем для якого складається список рекомендацій;

– SimilarFilmsComponent – схожі фільми з користувачем для якого складається список рекомендацій;

– TargetUserComponent – маркерний компонент, щоб позначити користувача для якого складається список рекомендацій;

– OverallScoresComponent – список зважених оцінок для кожного фільму, який потрапив в список схожих фільмів;

– AllSimilarityComponent – список оцінок схожості;

– MovieRecomendationComponent – список фільмів, які рекомендуються до перегляду.

Наступним кроком є реалізація систем, потрібних для поточної програми. Список створених систем:

– InitUsersEntitySystem – система, яка створює сутності користувачів і заповнює їх даними;

– SetTargetUserSystem – система, яка визначає для якого користувача складатиметься список рекомендацій;

– AddSimilaryScoreSystem – додає коефіцієнт схожості на усі сутності користувача, окрім тієї для якої складається список рекомендацій;

– AddSimilarFilmsToUser – визначення схожих фільмів для кожного користувача;

– CalculateScoresToTargetUser – розрахунок зважених оцінок, для кожного фільму, який потрапив в список схожих фільмів;

– GenerateMovieRecomendationSystem – розрахунок списку фільмів, рекомендованих до перегляду;

– PrintMovieRecomendationSystem – виведення списку фільмів в консоль;

– CleanTempComponentsSystem – очищення тимчасових компонентів;

– CheckFinishedSystem – завершення роботи програми, якщо уся поставлена робота виконана.

Після закінчення реалізації систем, залишається тільки викликати їх точок входу (Лістинг 4.14).

Лістинг 4.14 – Точка входу

```
systems = MovieRecomenderSystems (dataset [«Chris Duncan»])
systems.initialize()
while (not State.get _ instance ().isFinished) :
        systems.execute()
```

4.2.6 Тестування

Тестування буде виконано мануальним та автоматизованим способами. На вхід програм буде подано декілька різних імен користувачів, які є в базі і будуть перевірені результати. Результати обох реалізацій зображені на рисунках 4.7. В якості користувачів вибрані: Chris Duncan та Bill Duffy.

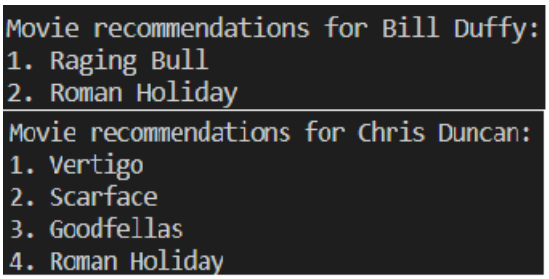

Рисунок 4.7 – Результат роботи програми

Наступний крок тестування це подання імені неіснуючого користувача. Результат цього етапу зображений на рисунку 4.8.

Рисунок 4.8 – Помилка при введені імені

Останнім кроком буде виконано автоматизоване тестування. Для цього буде написано два тест-кейсу. Перший буде перевіряти чи правильно завантажилися дані для конкретного юзера. Другий буде перевіряти, шо програма сгенерувала результат для правильного користувача.

Тести будуть виконані в форматі додаткового модуля і їх можна ввімкнути і вимкнути коли завгодно. Кожний тест буде системою, я буде міститись в TestSystems.

Перший тест(Лістинг 4.15) буде знаходити потрібного користувача і порівнювати список фільмів.

Лістинг 4.15 – Тест на перевірку завантаженого списку фільмів

```
class TestUserFilmList(ExecuteSystem):
    def __init__(self, userName, filmList):
        self.group = Matcher.get_instance().get_group([UserNameComponent, UserFilmsComponent])
        self.testedUserName = userName
```

```
            self.filmList = filmList

        def execute(self):
            for entity in self.group.entities:
                if entity.get_component(UserNameComponent).value ==
self.testedUserName:
                    for film in self.filmList:
                        if not film in entity.get_component(UserFilmsComponent).value:
                            print('TestUserFilmList is failed not right film list for ' +
self.testedUserName)
                            return
            print(«TestUserFilmList is success»)
```

В якості вхідних даних буде подано два набори, один правильний та один неправильний. Результати тесту можна побачити на рисунках 4.9 та 4.10.

```
TestUserFilmList is success
```

Рисунок 4.9 – Результат тесту при подачі правильного набору

```
TestUserFilmList is failed not right film list for Chris Duncan
```

Рисунок 4.10 – Результати тесту при подачі неправильного набору даних

Другий тест(Лістинг 4.16) буде визначати чи є серед користувачів для яких створено список рекомендацій потрібний користувач.

Лістинг 4.16 – Тест на перевірку наявності списку рекомендацій для потрібного користувача

```
class TestTargetUserHaveMovieRecomdation(ExecuteSystem):
```

```
        def __init__(self, targetUserName):
            self.group = Matcher.get_instance().get_group([UserNameComponent,
MovieRecomendationComponent])
            self.testedUserName = targetUserName

        def execute(self):
            for entity in self.group.entities:
                if entity.get_component(UserNameComponent).value ==
self.testedUserName:
                    print(«TestTargetUserHaveMovieRecomdation is success»)
                    return

            print('TestTargetUserHaveMovieRecomdation is failed' + 'Not have
results for' + self.testedUserName)
```

В якості вхідних наборів буде подано два значення. Одне з них буде правильно, а друге з них буде помилкове.

```
TestTargetUserHaveMovieRecomdation is success
```

Рисунок 4.11 – Результат тесту при подачі правильного набору

```
TestTargetUserHaveMovieRecomdation is failed Not have results forAdam Cohen
```

Рисунок 4.12 – Результати тесту при подачі неправильного набору даних

5 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

5.1 Аналіз використання ECS для C#

Аналіз архітектури програмного продукту буде проводитись по таким параметрам:

– швидкість роботи;

– гнучкість архітектури;

– можливість автоматизації тестування;

– час розробки.

Для початку треба проаналізувати час за який програма виконує один повний цикл. Для цього був використаній стандартний таймер.

Таблиця 5.1 – Порівняння часу одного повного циклу програми

Кількість літаків	ООП реалізація(в секундах)	ECS реалізація(в секундах)
10	0	0
100	0,5	0,3
1000	3	1
10000	15	2

Виходячи з результату дослідження часу одного повного циклу випливає що ECS реалізація працює значно швидше. Причому при збільшенні вхідних даних, швидкість зростає в геометричній прогресії. Цей результат досягнуто через використання Data Locality. Оскільки, всі данні знаходять в кеші. Результати можуть доволі сильно відрізнятись на різних комп’ютерах. Це пов’язано з тим, що в більш ранніх процесорах кеш працює повільніше, або відсутній взагалі.

Другим етапом аналізу, буде перевірка гнучкості архітектури. Припустимо, що до технічного завдання додалась вимога щоб данні про

місцезнаходження літаків, не рахувались в програмі, а надходили з зовнішнього сервера.

При ООП реалізації треба внести корективи в два основних класи. Це AirplaneManagerEmulator та Airplane. Це є прямим порушенням Open/Closed Principe. Для великої програми це може бути фатальним. Також логіка роботи програми дещо зміниться. І потрібно буде редагувати багато класів, що може зайняти багато часу і зусиль.

При ECS реалізації потрібно лише замінити систему MoveAirplanesSystem на GetArplanesPositionFromServerSystem. Це не є порушенням жодного з жодних SOLID принципів. Також, ця зміна ніяк не змінить іншу логіку програми.

Наступним кроком аналіз буде перевірка програми на автоматизацію тестування. Припустимо, треба перевірити що по проходженню певного часу всі літаки прибули в точку призначення.

При ООП підході потрібно отримати поточне и кінцеве місцезнаходження літака и порівняти їх. Але при розробці програми було виконано правило інкапсуляції. І якщо відкрити данні ці данні для інших користувачів, то це може призвести до різного роду наслідків.

При ECS підході можна реалізувати систему, яка порівнює значення поточного и кінцевого місцезнаходження и додавати на літак компонент FinishedComponent. Окрім, простого тестування, також цей компонент можна використати, якщо при оновленні технічного завдання він знадобиться. Отримати значення цих компонентів можна за допомогою GetComponent методу.

Останнім кроком аналізу буде аналіз часу розробки. Сама реалізація архітектурного патерну ECS в загальний час включена не буде, оскільки це треба зробити лише один раз, або взагалі узяти готове рішення.

ECS реалізація займає більше часу за рахунок того що програма будується з великої кількості систем і тому програма містить в собі більше коду. Але при використанні ECS набагато швидше імплементуються нові

механізми. Також якщо використовувати JetBrains Rider, то можна досягти практично ідентичної швидкості до ООП підходу. И ще один плюс ECS підходу, це те що якщо розробку програми буде продовжувати інший розробник, він значно швидше розбереться з архітектурою програми.

5.2 Аналіз використання ECS для Python

Аналіз архітектури програмного продукту буде проводитись по таким параметрам:

– гнучкість архітектури;

– можливість автоматизації тестування;

– час розробки;

– модульність архітектури.

Перший етан аналізу це перевірка гнучкості архітектури. При функціональному підході гнучкість архітектури доволі мінімальна. Оскільки, програма має одну основну функцію. Щоб замінити якийсь елемент розрахунків, потрібно вносити зміни до цієї функції, а це є порушенням Open/Closed Principe. В ECS реалізаціїї всі дії поділені на маленькі системи. Припустимо, що треба відсіяти користувачів, які переглянули менше ніж 5 фільмів. В функціональній реалізації треба змінити частину основної логіки, що не є гарним рішенням. В ECS же лише треба додати нову систему, яка буде відсіювати користувачів, які не задовольняють умовам.

Наступним етапом є перевірка можливості автоматичного тестування. В обох реалізація автоматизація тестування можлива. Але при функціональній реалізації можливо лише протестувати лише кінцевий результат. А в ECS підході можна також протестувати проміжні результати.

Час розробки при функціональному підході в галузі штучного інтелекту значно швидше. Оскільки, вся основна логіка виконуються в одній чи декількох функціях і не потрібно задуватись про зберігання та отримання даних.

Модульність архітектури ECS є великою перевагою перед функціональним стилем. Оскільки дуже важко створювати різні модулі при функціональному підході. А в ECS кожна система виконує одну одиницю логіки. И потрібний набір систем можна використати в іншій програмі.

Що стосується швидкості роботи програмного продукту, то в мові програмування Python вона не є важливою. Оскільки, в цій галузі важливий результат роботи продукту, а не швидкість його виконання. Також в мові програмування Python не є можливим реалізація Data Locality.

ВИСНОВКИ

Основною метою проекту дослідження моделей застосування архітектурного патерну Entity Component System. Під час дослідження було виявлено, що Entity Component System підходить не для всіх мов програмування. Також цей патерн в деяких випадках може буде надмірним.

Для перевірки ефективності застосування Entity Component System було розроблено два програмних продукти на різних мовах програмування.

Перший це система для слідкування за літаками, які знаходять в повітрі. Мовою програмування цього продукту була обрана C#. Цей продукт був розроблений двома архітектурними підходами, об’єктно-орієнтованим та компонентно-орієнтованим з застосуванням власної ефективної реалізації ECS. Був проведений аналіз ефективності використання різних підходів. Також був створений автоматичний тест для перевірки роботи програмного продукту.

Другий програмний продуктом була обрана рекомендаційна система, яка складає список фільмів рекомендованих до перегляду, на підставі даних, наданих у файлі ratings.json. Цей продукт відноситься до галузі штучного інтелекту та був розроблений використовуючи мову програмування Python. Для розробки було використано два архітектурних підходи, функціональний та компонентно-орієнтований з застосуванням власної реалізації ECS. Також було виконане тестування мануальним та автоматизованим способами. Був проведений аналіз різних архітектурних підходів.

В ході аналізу програмних продуктів було виявлено, що Entity Component System можливе лише в мовах, які мають інструменти для об’єктно-орієнтованого програмування. Також було доведено, що Entity Component System значно спрощує створення модульної та гнучкої архітектури.

В ході магістерської роботи було проаналізовано існуючи архітектурні підходи та різні мови програмування. Були виявлені їх сильні та слабкі сторони.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Джоши, Искусственный интеллект с примерами на Python [Текст] / Джоши. – К. : Издательство Диалектика, 2019. – 448 с.

2. Robert N. Game Programming Patterns [Текст] / R. Nystrom. – L. : GB, 2014. – 354 с.

3. Себеста Р. Основные концепции языков программирования [Текст] / Р. Себеста. – С. Питерб. : Питер, 2003 . – 672 с.

4. Шаблон проектирования Entity-Component-System [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/post/343778/ – 2014. – Загл. с экрана.

5. Как и почему мы написали свой ECS [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/company/pixonic/blog/413729/ – 2018. – Загл. с экрана.

6. Создание архитектуры программы или как проектировать табуретку [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/post/276593/ – 2016. – Загл. с экрана.

7. Троелсен Э. Язык программирования C#5.0 и платформа .NET 4.5 [Текст] / Э. Троелсен. – М. : Диалектика, 2013. – 1312 с.

8. Barry P. Head First Python [Текст] / P. Barry. – B. : O'Reilly Media, 2016. – 624 с.

9. Bruce M. Software Engineering: A Practitioner's Approach [Текст] / M. Bruce. – N. : McGraw-Hill Education, 2014. – 976 с.

10. Bass L. Software Architecture in Practice [Текст] / L. Bass, P. Clements, R. Kazman; B. : Addison-Wesley Professional., 2014. – 624 с.

11. Design Patterns [Текст] / Э.Гамма, Д. Влисидис, Р. Хелм, Р. Джонсон. – B. : Addison-Wasley, 1994. – 368 с.

12. Шевчук А. Design Patterns via C# [Текст] / А. Шевчук, О. Денис; – С. Питерб : Питер, 2015. – 288 с.

13. Mitchell T. Machine Learning [Текст] / T. Mitchell. – N. : McGraw-Hill Education, 2017. – 432 с.

14. Fabian R. Data-Oriented Design [Текст] / R. Fabian. – L. : Richard Fabian, 2018. – 653 с.

15. Martin R. Чистая архитектура. Искусство разработки программного обеспечения [Текст] / R. Martin. – С. Питерб : Питер , 2012. – 458 с.

16. «Забытые» парадигмы программирования [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/post/223253/ – 2014. – Загл. с экрана.

17. Gabbrielli M. Programming Languages: Principles and Paradigms [Текст] / Maurizio Gabbrielli. – B. : Springer, 2010. – 460 с.

18. Fowler M. Patterns of Enterprise Application Architecture [Текст] / M. Fowler. . – B. : Addison-Wasley, 2012. – 558 с.

19. Martin R. Чистый код. Создание, анализ и рефакторинг [Текст] / R. Martin. – С. Питерб. : Питер, 2019. – 448 с.

20. Нейман Д. Вычислительная машина и мозг [Текст] / Д. Нейман – М. : АСТ, 2018. – 528 с.

21. Макконнелл С. Совершенный код [Текст] / С. Макконнелл – М. : Русская редакция, 2016. – 359 с.