МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження використання кінцевих автоматів та їх об'єктно-орієнтованих моделей при проектуванні та реалізації поведінки програмних систем
(тема)

Виконав: студент 2 курсу, групи ІПЗм-19-2
Нікітін Д.М.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми Освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Керівник к.т.н, доц. Голян В.В.
(посада, прізвище)

Допускається до захисту

Зав. кафедри ______________ З.В. Дудар
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121-Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри _________
(підпис)

«26» березня 2021 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Нікітіна Дмитра Михайловича
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження використання кінцевих автоматів та їх об'єктно-орієнтованих моделей при проектуванні та реалізації поведінки програмних систем затверджена наказом університету від «26» березня 2021 р. № 385

2. Термін подання студентом роботи до екзаменаційної комісії «10» травня 2021 р.

3. Вихідні дані до роботи *теорія автоматів, методи дискретної математики, патерни проектування об'єктно-орієнтованих програм, об'єктно-орієнтоване програмування, пояснювальна записка*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз предметної галузі і постановка задачі, огляд патернів проектування програмних систем, використання кінцевих автоматів у моделі асинхронного програмування, модифікація існуючих алгоритмів та патернів, тестування розроблених патернів та алгоритмів*

5. Перелік графічного матеріалу із зазначенням кресленників, схем, слайдів, ілюстрацій *граф автомату, діаграма взаємодії компонентів системи, діаграма класів, рисунки програмного коду, результати порівняльного аналізу, інтерфейс компоненту для виміру продуктивності*

6. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	к.т.н, доц. Голян В. В.		02.04.2021

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	27.01.2021	
2	Огляд існуючих методів, патернів та алгоритмів	08.02.2021	
3	Модифікація та покращення патернів, алгоритмів та їх тестування: аналіз отриманих результатів	08.03.2021	
4	Підготовка пояснювальної записки	29.03.2021	
5	Спецчастина	08.04.2021	
6	Підготовка презентації та доповіді	20.04.2021	
7	Попередній захист роботи	04.05.2021	
8	Нормоконтроль, рецензування	05.05.2021	
9	Занесення диплома в електронний архів	07.05.2021	
10	Допуск до захисту у зав. кафедри	11.05.2021	

Дата видачі завдання «25» січня 2021 р.

Студент гр. ІПЗм-19-2______________ Нікітін Д.М.
(підпис)

Керівник роботи ______________ к.т.н, доц. Голян В.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи магістра містить: 76 стор., 19 рис., 21 джерело.

АБСТРАКЦІЯ, АРХІТЕКТУРА СИСТЕМИ, АСИНХРОННА МОДЕЛЬ, КІНЦЕВИЙ АВТОМАТ, ПАТЕРН ПРОЕКТУВАННЯ, СТАН, ФУНКЦІЯ ПЕРЕХОДУ, EVENT CONTEXT, EVENT MANAGER, ITERATOR, STATE TABLE

Об’єкт дослідження – використання кінцевих автоматів при проектуванні програмних систем.

Мета дослідження – аналіз існуючих програмних рішень та алгоритмів, які базуються на кінцевих автоматах, та їх модернізація для оптимізації проектування та реалізації програмних систем.

В результаті роботи було вивчено програмні реалізації алгоритмів з використання кінцевих автоматів та патерну «State» у асинхронному програмуванні; визначено можливості модернізації існуючих алгоритмів для покращення їх роботи; модернізовано патерн «State», який дозволяє об’єктам змінювати свою поведінку в залежності від їх стану, з усуненням основних недоліків; запропоновано метод модифікації вихідного коду програми для використання нового патерну з мінімальною кількістю змін, які можуть привести до появи помилок.

Практичне значення роботи полягає в тому, що отримані результати спрощують розробку, підтримку і супроводження програмних систем та можуть бути використані на практиці при написанні та проектуванні їх поведінки.

ABSTRACTION, ASYNCHRONOUS MODEL, DESIGN PATTERN, EVENT CONTEXT, EVENT MANAGER, FINITE-STATE MACHINE, ITERATOR, STATE, STATE TABLE, STATE-TRANSITION FUNCTION, SYSTEM ARCHITECTURE

Objective of study – the use of finite-state machines in the software design.

Purpose of study – analysis of existing software solutions and algorithms based on

finite-state machines, and their modernization to optimize the design and implementation of software systems.

As a result, the software implementations of algorithms that use finite-state machines and the "State" pattern in asynchronous programming were studied; opportunities to upgrade existing algorithms to improve their performance were identified; the "State" pattern was upgraded, which allows objects to change their behavior depending on their state, eliminating the main drawbacks; a method of modifying the source code of the program to use the new pattern with a minimum number of changes which can lead to errors was proposed.

The practical value of the work is that the results simplify the development and maintenance of software systems and can be used during implementing and designing their behavior.

Я, Нікітін Дмитро Михайлович, студент гр. ІПЗм-19-2, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження використання кінцевих автоматів та їх об'єктно-орієнтованих моделей при проектуванні та реалізації поведінки програмних систем», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

Вступ 8
1 Аналіз предметної галузі 11
1.1 Аналіз предметної галузі 11
1.2 Існуючі алгоритми та патерни, що базуються на кінцевих автоматах 13
1.2.1 Патерн «State» 14
1.2.2 Модель асинхронного програмування 16
1.3 Постановка задачі 20
2 Патерн проектування «State Engine» 22
2.1 Огляд 22
2.2 Структура 23
2.3 Застосування 26
2.4 Можливості розширення 27
2.5 Перехід від патерну «State» до патерну «State Engine» 28
2.6 Відносини з іншими патернами 29
2.7 Подальший розвиток та покращення 30
3 Модель кінцевого автомату в асинхронних операціях 32
3.1 Принципи реалізації асинхронності у .NET 33
3.2 Недоліки існуючого підходу до асинхронності у .NET 35
3.3 Модифікація алгоритму виконання асинхронних операцій 36
3.4 Недоліки нової версії алгоритму та способи її покращення 41
4 Тестування розроблених методів та аналіз отриманих результатів 43
4.1 Порівняння використання патернів «State» та «State Engine» 43
4.2 Проведення тестів продуктивності модифікованої версії алгоритму виконання асинхронних операцій 44
Висновки 48
Перелік джерел посилання 50
Додаток А Перелік джерел посилання за науковими напрямами керівника та

науковців кафедри програмної інженерії 52
Додаток Б Звіт результатів перевірки кваліфікаційної роботи на унікальність тексту 53
Додаток В Слайди презентації 54
Додаток Г Елементи програмного коду 71
Додаток Д Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008:2015 75

ВСТУП

У сучасному світі більшість програмних продуктів є високонавантаженими, розподіленими, масштабованими системами. Розробка будь-якої програми, чи то невеликої процедури з обробки інформації, чи комплексного програмного продукту, складається з декількох етапів, грамотна реалізація яких є обов'язковою умовою для досягнення успішного результату. Управління такими великими проектами вимагає ретельного аналізу існуючих підходів, методів та інструментів, що дозволить підлаштовуватися під виникаючі вимоги без зайвих проблем [1].

При розробці програмного забезпечення основоположним етапом є проектування – повне планування того, що доведеться розробляти, в які терміни, з якими вихідними даними і очікуваним результатом. У першу чергу це впливає на обрану модель життєвого циклу програмної системи, яка визначає етапи та процес створення програмного забезпечення [2].

В даний час актуальною є проблема розробки якісного програмного забезпечення. На етапі проектування важливою задачею є створення такої архітектури, яка буде надавати можливості для розширення або швидкої зміни модулів системи. На етапі розробки важливо створити максимально оптимізовані компоненти, які будуть легкими у розумінні іншими розробниками, гарантуватимуть легкість супроводу та зможуть бути використані у декількох місцях.

Особливістю сучасних програмних систем є використання під час розробки процесу рефакторингу [3] – регулярної перебудови вихідного коду програми (без зміни основного функціоналу) для полегшення підтримки змін та поліпшення розуміння коду.

На етапі розробки ПЗ багато уваги приділяється реалізації взаємодії з додатковими системи та сервісами для отримання та обробки даних (ввід-вивід): взаємодія з базою даних, файловою системою, передача інформації по мережі Інтернет – усе це повинно бути враховано ще на етапі проектування.

Досить часто є необхідність спроектувати модуль, який буде взаємодіяти з додатковими системи і сервісами для отримання та обробки даних, або компонент системи, що представляє собою об’єкт обраної доменної області, який може змінювати свої властивості або поведінку в залежності від поточного стану. В обох випадках має сенс використати алгоритм або патерн, який працює на основі кінцевого автомату або його об’єктно-орієнтованому представленні.

Для досягнення цілей, описаних вище, існує велика кількість методів та шаблонів для створення програмних модулів. Але деякі з них, окрім явних переваг, мають і недоліки, які можуть на досить пізньому етапі розробки призвести до появи неочікуваних помилок або завадити внесенню змін у систему.

Тому задача покращення способів проектування та реалізації програмних систем (у тому числі тих, що використовують кінцеві автомати) є актуальною і досі, що підтверджується великою кількістю створюваних методик та модернізацією існуючих.

Метою дослідження є аналіз існуючих програмних рішень та алгоритмів, які базуються на кінцевих автоматах, та їх модернізація для оптимізації проектування та реалізації програмних систем.

Об’єктом дослідження, обраним для вивчення, є використання кінцевих автоматів при проектуванні програмних систем.

Предметом дослідження є проектування та реалізація поведінки програмних систем з використанням кінцевих автоматів та їх об’єктно-орієнтованих моделей.

Під час виконання роботи було використано наступні методи дослідження:

– методи теорії автоматів (теоретичні дані про кінцеві автомати);

– методи дискретної математики (кінцеві автомати як модель дискретної системи);

– патерни проектування об'єктно-орієнтованих програм (у тому числі розглянутий патерн «Стан»);

– методи побудови і аналізу алгоритмів (для реалізації виконання асинхронних операцій);

– абстрагування (виділення тільки основних властивостей програмної

системи: абстрагування від апаратної взаємодії при виконанні операцій вводу-виводу; абстрагування від математичних властивостей кінцевого автомату при розгляді патерну проектування);

– формалізація (представлення кінцевого автомату у вигляді множини елементів).

Практичне значення роботи полягає в тому, що отримані результати спрощують розробку, підтримку і супроводження програмних систем та можуть бути використані на практиці при написанні та проектуванні їх поведінки.

Результати даної роботи було опубліковано у збірниках матеріалів 2 наукових конференцій:

– «V Міжнародна науково-практична конференція «PRIORITY DIRECTIONS OF SCIENCE AND TECHNOLOGY DEVELOPMENT», яка відбулась 24-26 січня 2021 року у м. Київ, Україна;

– «I Міжнародна науково-теоретична конференція «DÉBATS SCIENTIFIQUES ET ORIENTATIONS PROSPECTIVES DU DÉVELOPPEMENT SCIENTIFIQUE», яка відбулась 5 лютого 2021 року у м. Париж, Франція.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Традиційно в програмуванні використовують синхронну модель – послідовне виконання інструкцій з синхронними системними викликами, які повністю блокують потік виконання, поки системна операція, наприклад читання з диска, не завершиться.

Основним недоліком такого підходу є те, що потік повинен буде чекати, доки операція не завершиться. У високонавантажених системах найчастіше так і відбувається – майже весь час програма чогось чекає: диска, СУБД, мережі, якоїсь зовнішньої події. У мало навантажених системах це можна вирішити створенням нового потоку для кожної блокуючої дії. Поки один потік спить, інший працює.

Але що робити, коли користувачів дуже багато? Якщо створювати на кожного хоча б один потік, то продуктивність такого сервера різко впаде через те, що контекст виконання потоку постійно змінюється. Та і кількість потоків, які можуть повноцінно працювати паралельно, обмежена. Цю проблему може вирішити асинхронне програмування.

Асинхронність в програмуванні – виконання процесу в неблокуючому режимі системного виклику, що дозволяє потоку програми продовжити обробку. Тобто на час, який потрібен комп’ютеру для виконання I/O операції, потік не блокується, а виконує інші команди, повертаючись до попередніх інструкцій при зміні їх стану.

Одним з найбільш популярних та ефективних методів асинхронного програмування є реалізація з використанням алгоритму, побудованого на основі кінцевого автомату [4].

Якщо коротко, кінцевий автомат – це абстрактний автомат, число можливих внутрішніх станів якого кінечне. Тобто кінцевий автомат – це чорний ящик, в котрий можна щось передати і дещо звідти отримати. Це досить зручна абстракція, яка дозволяє інкапсулювати складний алгоритм, а також кінцеві автомати дуже

ефективні. При цьому для нього існує кінцева множина вхідних символів, що призводять до формування вихідних слів. Також важливим є те, що кожен вхідний символ здатен перевести автомат у новий стан.

У абстрактному вигляді кінцевий автомат є математичною схемою F, що характеризується 6-ма елементами [5]: кінцевою множиною X вхідних сигналів; кінцевою множиною Y вихідних сигналів; кінцевою множиною Z внутрішніх станів; початковим станом z_0; функцією переходів $j(z,x)$; функцією виходів $k(z,x)$.

Автомат, що задається схемою F: $F = |X, Y, Z, z_0, j, k|$, функціонує в дискретному часі, моментами якого є такти, кожному з яких відповідають постійні значення вхідного і вихідного сигналів і внутрішні стани [6].

Робота кінцевого автомату відбувається за такою схемою: на кожному i-му такті на вхід автомату, що знаходиться в стані z(i), подається деякий сигнал x(i), на який автомат реагує переходом на (i+1)-му такті в новий стан z(i+1) і видачою деякого вихідного сигналу y(i). Загалом роботу кінцевого автомату можна описати наступними формулами [7]:

$$\begin{cases} z(i+1) = j[z(i), x(i)], i = 0, 1, \ldots; \\ y(i) = k[z(i), x(i)], i = 0, 1, \ldots; \end{cases} \quad (1)$$

Приклад графу автомату зображено на рисунку 1.1.

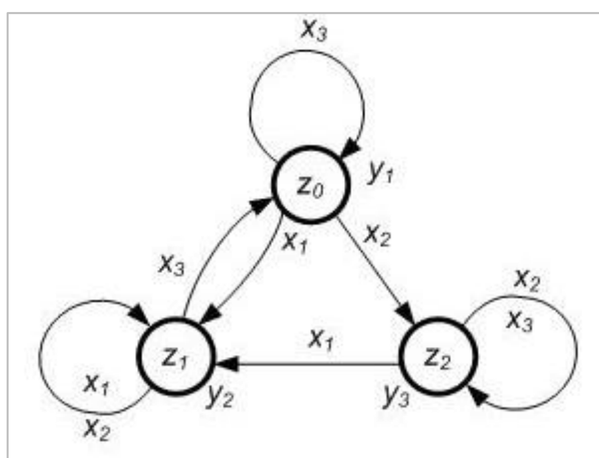

Рисунок 1.1 – Приклад графу автомату

Проекцією кінцевого автомату у світі об’єктно-орієнтованого проектування є патерн «Стан» («State») [8], який дозволяє об'єктам змінювати поведінку залежно від свого стану. Ззовні створюється враження, ніби змінився клас об'єкта. Саме цей патерн є об’єктно-орієнтованою реалізацією кінцевого автомату.

Патерн «Стан» використовується при проектуванні модулів (класів), які мають певний набір внутрішніх станів, та, в залежності від поточного стану, виконують різні операції (змінюють свою поведінку), надаючи при цьому один і той же інтерфейс взаємодії.

1.2 Існуючі алгоритми та патерни, що базуються на кінцевих автоматах

Патерн проектування – це типовий спосіб вирішення певної проблеми, що зазвичай використовується під час проектування архітектури програмних систем.

На відміну від розроблених бібліотек чи функцій, патерн не можна просто додати в програму. Патерн – це не якийсь конкретний код, а загально відомий принцип вирішення певної проблеми, що майже завжди потребує налаштування для потреб тієї чи іншої програми.

Описи патернів зазвичай дуже формальні та містять академічні терміни, й найчастіше складаються з таких пунктів [9]:

– проблема, яка вирішується використанням патерну;

– мотивація щодо вирішення проблеми методом, який пропонує патерн;

– структура класів описаного патерну;

– приклад однією з мов програмування;

– особливості реалізації в певних контекстах;

– переваги використання даного патерну;

– можливі проблеми використання даного патерну;

– зв’язок з іншими патернами.

1.2.1 Патерн «State»

Патерн «State» – це поведінковий патерн проектування, що дає змогу об’єктам змінювати поведінку в залежності від їхнього стану.

Патерн «State» необхідно розглядати у контексті концепції машини станів, також відомої як кінцевий автомат.

Машину станів досить часто реалізують за допомогою набору умовних операторів if (switch), які перевіряють поточний стан об’єкта та налаштовують необхідну поведінку для виконання.

Основна ідея полягає в тому, що компонент може знаходитися в одному з декількох станів, які постійно змінюють один одного. Набір цих станів, а також функцій переходів між ними, визначений наперед та кінцевий. Перебуваючи в різних станах, модуль може по-різному реагувати на одні й ті самі події, що з ним відбуваються.

Такий підхід можна застосовувати і до окремих програмних об’єктів. Наприклад, об’єкт «Document» може приймати три стани: «Draft», «Moderation» або «Published». У кожному з цих станів метод «publish» працюватиме по-різному:

– зі стану «Draft» він надішле документ до «Moderation»;

– з «Moderation» – в «Published», але за умови, що це зробив адміністратор;

– в стані «Published» метод не буде робити нічого.

За тією ж схемою буде працювати метод «revoke»:

– в стані «Published» метод поверне документ до «Moderation», але за умови, що це зробив адміністратор;

– з «Moderation» – в «Draft» для подальшого редагування та повторного надсилання до «Moderation»;

– в стані «Draft» метод не буде нічого робити.

Схематичне зображення даного прикладу відображено на рисунку 1.2.

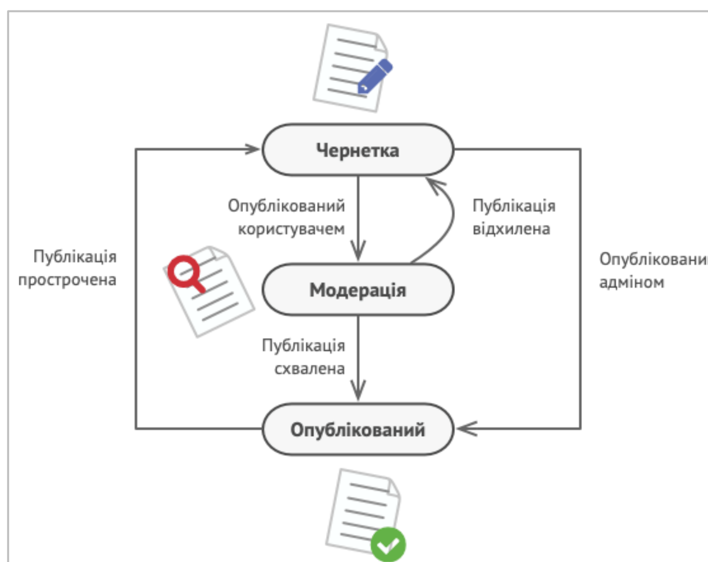

Рисунок 1.2 – Можливі стани компоненту «Document» та переходи між ними

Патерн «State» полягає у створенні окремих класів для кожного стану, в якому може перебувати компонент контексту, та винесенню до них поведінки, що відповідає описаним станам.

Початковий компонент, також відомий як контекст, замість того, щоб зберігати код всіх станів, містить посилання на один з об'єктів-станів і делегує йому роботу в залежності від стану.

Діаграму класів [10] патерну «Стан» зображено на рисунку 1.3.

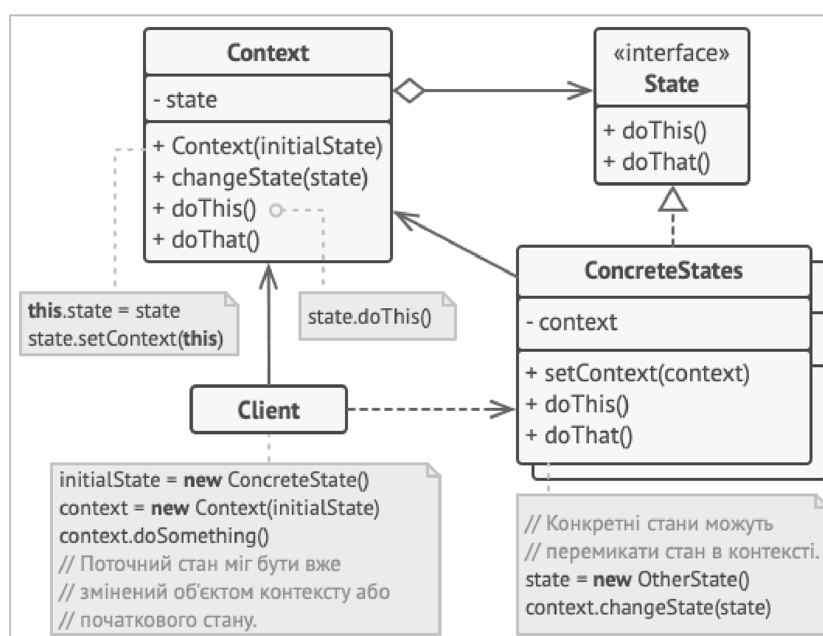

Рисунок 1.3 – Діаграма класів патерну «Стан»

Але цей патерн, як і будь-який інший, має певні недоліки [11]:

– поділ реалізації станів за різними класами призводить до розподілу логіки переходів по ним;

– не забезпечується незалежність класів, що реалізують стани, один від одного;

– створення ієрархії класів станів і їх повторне використання стає ускладненим.

1.2.2 Модель асинхронного програмування

Модель асинхронного програмування є досить популярною та реалізована у багатьох мовах програмування. У рамках даної роботи розглянемо реалізацію цієї моделі у мові C#.

Модель асинхронного програмування на основі об’єктів Task (TAP – Task-based Asynchronous Pattern) забезпечує абстракцію над асинхронним кодом [12]. Основна ідея цієї моделі полягає в тому, що розробник пише код як послідовність тверджень (інструкцій), не звертаючи уваги на те, синхронні вони чи асинхронні, а компілятор виконує ряд перетворень, оскільки деякі з цих операторів можуть почати асинхронну роботу.

Використання даної абстракції написання програмного коду і є метою цього синтаксису: дозволити писати код, який читається як послідовність операторів, але виконується у набагато складнішому порядку на основі розподілу зовнішніх ресурсів.

Ключові слова async та await у C# – це основа асинхронного програмування. Використовуючи ці два ключові слова, розробник має можливість використовувати ресурси в .NET Framework, щоб створити асинхронний метод. Методи, які визначаються за допомогою ключового слова async, називаються асинхронними методами.

Механізм async реалізований в компіляторі C # за підтримки бібліотек базових класів .NET. В саму виконуючу середу (CLR) не довелося вносити ніяких змін.

Розглянемо приклад коду, написаний з використанням асинхронних операцій та моделі асинхронного програмування у C# (рис. 1.4).

```
public async Task<List<Person>> GetPeopleAsync(int page)
{
    if (page < 0)
    {
        throw new ArgumentException("Page number must be greater than 0", nameof(page));
    }

    return await _db.GetPeopleAsync(page);
}
```

Рисунок 1.4 – Приклад асинхронного методу у C#

Даний метод позначено модифікатором async, що робить його асинхронним, а самі асинхронні дії позначено модифікаторами await, що вказує компілятору про їх специфічні вимоги щодо виконання.

Задача цього методу – повернути певну сторінку об’єктів типу «Person», які описують моделі людей. Спочатку перевіряється, що значення змінної «page» не менше нуля, оскільки від’ємне число є некоректним у даному випадку. Далі викликається метод «GetPeopleAsync» на об’єкті-абстракції бази даних, який відповідає за відправку запитів до БД. У якості значення, що повертається з методу, виступає список об’єктів типу «Person».

Даний фрагмент коду являє собою його версію до компіляції, тобто до заміни усіх допоміжних конструкцій автоматично згенерованими компонентами, які відповідають за правильний порядок виконання описаних операцій.

Розглянемо скомпільовану версію даного методу у вигляді набору інструкцій на мові C#, але вже після заміни програмних конструкцій і до компілювання у мову CIL (Common Intermediate Language), що являє собою проміжний артефакт етапу компіляції додатків для платформи .NET, за допомогою якого досягається кросплатформенність. Метод після компіляції зображено на рисунку 1.5.

```
[AsyncStateMachine(typeof(<GetPeopleAsync>d__3))]
[DebuggerStepThrough]
public Task<List<Person>> GetPeopleAsync(int page)
{
    <GetPeopleAsync>d__3 stateMachine = new <GetPeopleAsync>d__3();
    stateMachine.<>t__builder = AsyncTaskMethodBuilder<List<Person>>.Create();
    stateMachine.<>4__this = this;
    stateMachine.page = page;
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

Рисунок 1.5 – Код асинхронного методу, згенерований компілятором C#

У наведеному вище прикладі компілятор автоматично застосував до методу атрибут «AsyncStateMachine». Коли метод («GetPeopleAsync») має модифікатор async, компілятор генерує код, який включає структуру кінцевого автомату.

Код також містить метод «GetPeopleAsync», що викликається в кінцевому автоматі. Компілятор додає атрибут «AsyncStateMachine» до методу, щоб можна було ідентифікувати відповідний кінцевий автомат. Це необхідно для того, щоб створити об'єкт, здатний зберегти стан методу в момент, коли програма дійде до await. Адже код до цього ключового слова виконується в потоці, який його викликав, а потім при його досягненні зберігається інформація про місце методу, де перебувала програма, щоб продовжити виконання при поновленні.

Згенерований метод автоматично замінив той, який спочатку був написаний вручну. Тобто версія, отримана компілятором, повністю переробила початковий метод, замінивши основні конструкції, не впливаючи при цьому на функціональну частину модулю.

У даному випадку в якості вхідних символів кінцевого автомату виступатиме стан виконання асинхронної операції і вже на основі цього значення кінцевий автомат буде формувати вихідний стан і реакцію на виконання завдання. Даний підхід спрощує формування і управління виконанням асинхронних завдань.

Окрім наведених фрагментів компілятор генерує і найголовніший компонент асинхронної моделі – представлення кінцевого автомату, яке відповідає за перехід між станами асинхронних операцій та управління ними під час виконання програми.

Згенерований компонент реалізує інтерфейс «IAsyncStateMachine», який представляє кінцеві автомати, створені для асинхронних методів. У даній задачі він відповідає за початковий стан кінцевого автомата, перевіряє завершеність асинхронного завдання і переходить в потрібне місце методу. При цьому відбувається перехід в один з декількох станів автомата: зупинка методу в місці, де зустрічається await, або синхронне завершення. Частину коду згенерованого кінцевого автомату зображено на рисунку 1.6.

```
[CompilerGenerated]
private sealed class <GetPeopleAsync>d__3 : IAsyncStateMachine
{
    public int <>1__state;
    public AsyncTaskMethodBuilder<List<Person>> <>t__builder;
    public int page;
    private TaskAwaiter<List<Person>> awaiter;

    private void MoveNext()
    {
        List<Person> result;
        try
        {
            if (<>1__state != 0)
            {
                if (page < 0)
                {
                    throw new ArgumentException("Page number must be greater than 0", "page");
                }
                awaiter = <>4__this._db.GetPeopleAsync(page).GetAwaiter();
                if (!awaiter.IsCompleted)
                {
                    <>1__state = 0;
                    <GetPeopleAsync>d__3 stateMachine = this;
                    <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
                    return;
                }
            }
            else
            {
                <>1__state = -1;
            }
            result = awaiter.GetResult();
        }
        catch (Exception exception)
        {
            <>1__state = -2;
            <>t__builder.SetException(exception);
            return;
        }
        <>1__state = -2;
        <>t__builder.SetResult(result);
    }
}
```

Рисунок 1.6 – Фрагмент коду згенерованого кінцевого автомату

Даний компонент містить змінні, які зберігають поточний стан автомату, асинхронну операцію, виконувану у даний момент, та метод «MoveNext». Цей метод переміщує кінцевий автомат до його наступного стану. Цей метод містить в собі оригінальний код і викликається як при першому виклику методу, так і після ключового слова await.

Розробники, які мають досвід у написанні програм на мові C#, можуть побачити схожість назви даного методу з методами «MoveNext», які генеруються блоками ітераторів. Ці блоки дозволяють реалізувати інтерфейс «IEnumerable» в одному методі за допомогою ключових слів yield return. Застосовуваний для цієї мети кінцевий автомат нагадує асинхронний автомат, тільки простіше.

1.3 Постановка задачі

Базуючись на аналізі обраної предметної галузі, можна виділити основні проблеми, вирішенням яких буде приділятись увага під час виконання дослідження:

– складність керування етапами виконання операцій у асинхронній моделі програмування;

– зменшення кількості внутрішніх станів та переходів між ними при виконанні асинхронних операцій;

– неможливість повторного використання коду станів в об’єктно-орієнтованих реалізаціях кінцевого автомату;

– децентралізація логіки переходів між станами об’єкту (процес зміни поточного стану);

– залежність об’єктно-орієнтованих класів станів один від одного (неможливість їх використання для опису інших моделей).

На основі описаних проблем можна виокремити задачі, вирішення яких є метою проведеного дослідження:

– вивчення програмних реалізацій алгоритмів з використання кінцевих автоматів у моделі асинхронного програмування;

– аналіз патерну проектування програмного забезпечення «Стан», який базується на об’єктно-орієнтованій абстракції кінцевого автомату;

– модернізація існуючих алгоритмів, які використовують моделі кінцевих автоматів, для керування виконанням асинхронних операцій у програмних системах;

– розробка патерну проектування, який дозволяє об’єктам змінювати свою поведінку в залежності від їх стану, з усуненням основних недоліків існуючого патерну «Стан»;

– розробка методу рефакторингу, який дозволить перейти від використання патерну «Стан» до використання розробленого патерну з мінімізацією шансів на виникнення помилок.

2 ПАТЕРН ПРОЕКТУВАННЯ «STATE ENGINE»

Найбільш відомою реалізацією об'єкта, що змінює поведінку в залежності від стану, є патерн «State». Недолік патерна «State» полягає в тому, що поділ реалізації станів за різними класами призводить до розподілу логіки переходів по ним, що ускладнює розуміння програми. При цьому не забезпечується незалежність класів, що реалізують стани, один від одного. Таким чином, створення ієрархії класів станів і їх повторне використання ускладнене.

У даній роботі пропонується новий патерн «State Engine», який поєднує переваги кінцевих автоматів та патерну «State».

2.1 Огляд

Для того, аби надати можливість використовувати класи станів повторно (для інших моделей домену або у контексті будь-якого іншого компоненту), даний патерн використовує механізм відправки подій, які використовуються та надсилаються класами станів автомату, сповіщаючи про операції, які були виконані. Об’єкт автомату, у свою чергу, реагує на ці події, змінюючи свій поточний стан, що може виконуватись як самим автоматом, так і окремим об’єктом, який буде за це відповідальний. Дана схема певним чином схожа з патернами «Observer» / «Publish-Subscribe», які використовуються для обміну повідомленнями між компонентами системи або між підсистемами (за рахунок брокерів повідомлень) [13].

Описані зміни дозволяють централізувати логіку переходу між станами автомату в одному місці, усунувши необхідність класів станів знати один про одного (що можна побачити у класичній реалізації патерну, де кожний конкретний стан відповідає за зміну стану автомату).

При цьому сама логіка переходів між станами автомату може бути реалізована різними методами. Наприклад, можна побудувати таблицю переходів між станами, яка буде вказувати, на який стан перейти при поточному стані та події, яку було від нього отримано. Іншим способом є використання стандартних засобів мови програмування, як то інструкції switch (if-else), які будуть для кожного варіанту пари стан-подія генерувати новий стан об'єкту кінцевого автомату [14].

2.2 Структура

Припустимо, що є необхідність спроектувати клас «Document», який було описано у попередньому розділі, який представляє собою документ, що може редагуватися, перевірятися та публікуватися.

Документ може мати 3 стани:

– «чернетка» – початковий стан, коли документ було створено, але не було перевірено чи опубліковано;

– «модерація» – документ перебуває у цьому стані після чернетки та наповнення документу вмістом. На даному етапі документ перевіряється модератором. Якщо під час модерації було виявлено помилки, файл відправляється у стан «чернетка»;

– «опублікований» – якщо після модерації не було виявлено жодних помилок, документ публікується адміністратором системи, що змінює його стан на «опублікований».

Перехід між станами документу відбувається за допомогою методів «render» та «publish». У стані «чернетка» метод «render» відображає документ для перегляду, а метод «publish» – відправляє його модератору для перевірки, змінюючи при цьому його стан на «модерація». Під час модерації метод «render» дозволяє модератору переглянути останню версію, відправлену автором, а метод «publish» – відправити документ до публікації.

Описана схема є досить простою, але при необхідності може бути розширена за допомогою додаткових класів станів (нових методів для кожного стану) або умов, необхідних для виконання певних дій (наприклад, документ може бути опублікований лише користувачами, які мають на це право, а після модерації документ потрапляє спочатку у стан «готовий до публікації»).

Таким чином, описані класи повинні мати вигляд, зображений на рисунку 2.1, що являє собою діаграму класів для описаної ситуації.

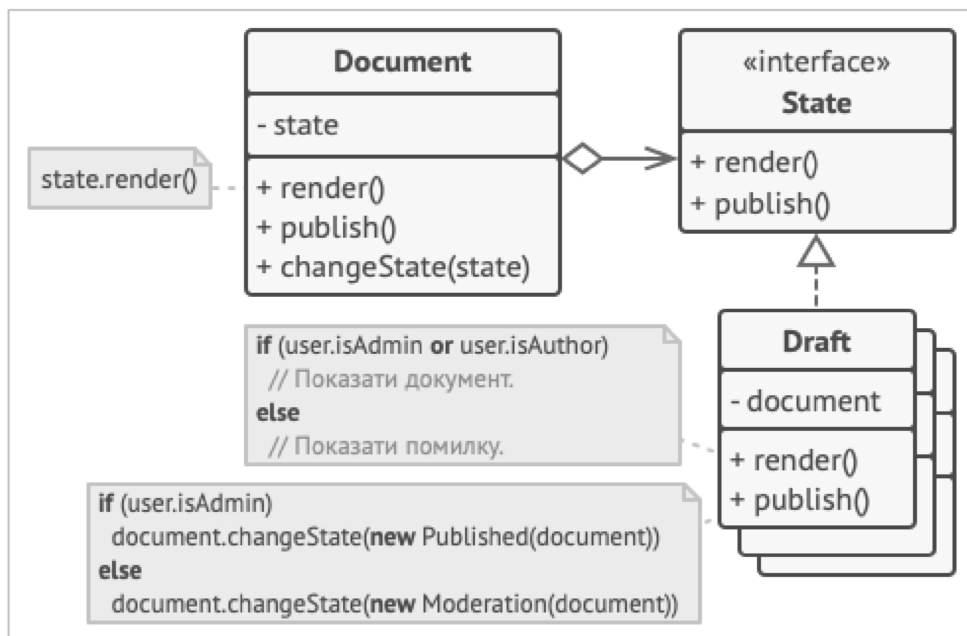

Рисунок 2.1 – Діаграма класів моделі документу

Поточний стан документа було відображено за допомогою зв'язку «has a» (композиції), що говорить про те, що об'єкт документу визначає період життя об'єкту стану (тобто стан не може існувати, якщо не існує документ).

Обмеження, описані у доменній моделі, повинні бути реалізовані у класах станів (на даній схемі це клас «Draft» та інші), що дозволить розподілити їх незалежно один від одного.

Основною відмінністю реалізації даної ситуації з використанням патерну «State Engine» є те, що описані стани документа не знають про існування інших станів, що дозволяє динамічно міняти набір станів без зміни існуючої логіки переходів між ними.

Структуру модернізованого патерну зображено на рисунку 2.2.

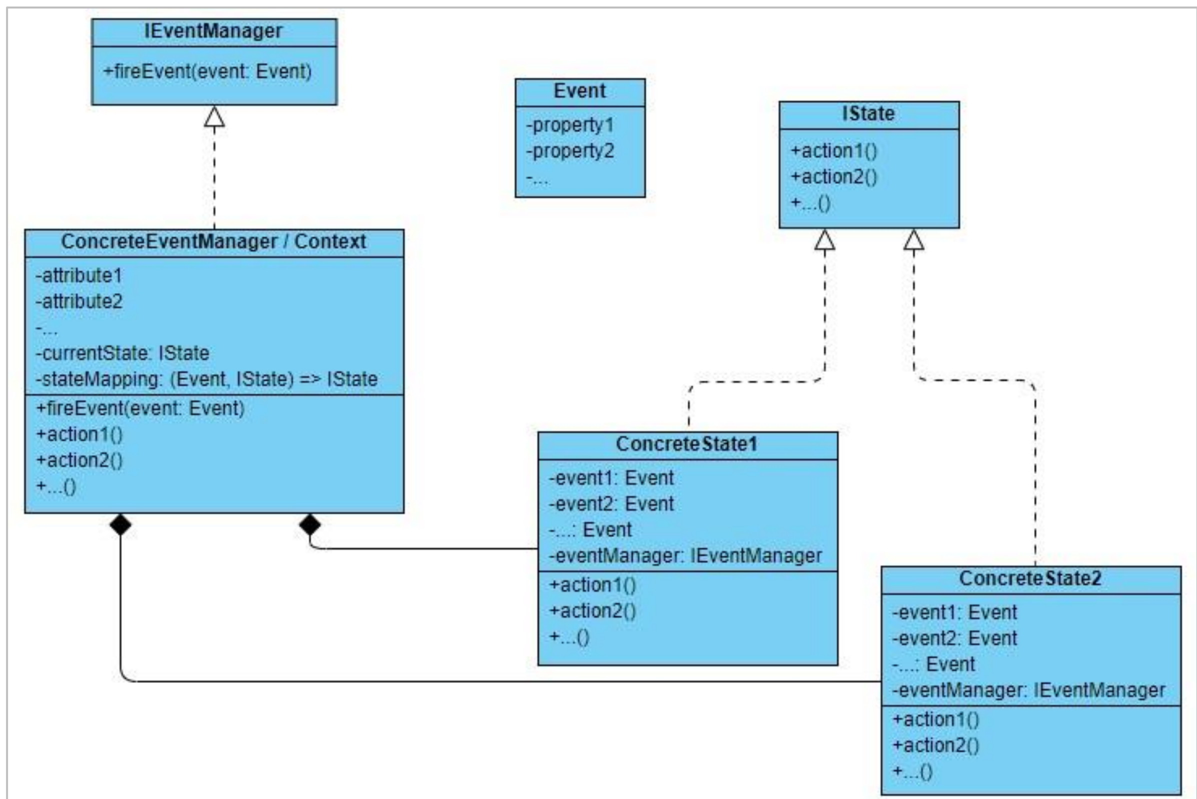

Рисунок 2.2 – Діаграма класів патерну «State Engine»

На даній діаграмі присутні наступні учасники:

– Event – базовий клас для події, яка була згенерована класами станів. Містить набір атрибутів «property1», «property2», що можуть бути використані класом, який отримує події;

– IState – базовий інтерфейс типів станів, який визначає набір операцій («action1», «action2»), які повинні підтримувати усі стани певного об'єкту;

– ConcreteState1, ConcreteState2 – конкретні класи станів, які реалізують інтерфейс «IState». Саме ці класи відповідають за створення подій, що описуються класом «Event»;

– IEventManager – базовий інтерфейс компоненту, який може приймати події від конкретних станів, та який передається станам при їх створенні. Саме цей інтерфейс є єдиним контрактом, що дозволяє взаємодіяти класам станів та кінцевому автомату;

– ConcreteEventManager/Context – конкретний клас абстракції кінцевого автомату, який має поточний стан, що може бути змінений упродовж життєвого циклу, а також реалізує інтерфейс «IEventManager» для реагування на події, що сталися, та зміни свого стану.

При створенні об’єкту абстракції кінцевого автомату (класу «ConcreteEventManager»), відбувається ініціалізація усіх існуючих станів компоненту та побудова графу переходів між станами при виникненні певних події. Усі створені стани отримують об’єкт для повідомлення про події та об’єкт контексту, що у даному випадку є один типом «ConcreteEventManager», але може бути розділений на різні класи за потреби.

Під час роботи об’єкт автомату делегує виконання операцій поточному стану, який може модифікувати збережений контекст або відправити подію, що призведе до зміни поточного стану автомату на новий, визначений у таблиці переходів. Тобто рішення про зміну стану приймає сам автомат, а не його стани.

2.3 Застосування

Патерн «State Engine» може бути використаний у наступних випадках:

– якщо є компонент, який кардинально змінює свою поведінку в залежності від свого внутрішнього стану, причому таких видів станів багато (і вони можуть змінюватися), а їхній код часто змінюється;

– якщо програмний код модулю містить багато складних, схожих один на одного умовних операторів, які визначають поведінку в залежності від поточних значень властивостей модулю;

– якщо використовується таблична машина станів, побудована на умовних операторах;

– якщо є необхідність повторно використати класи станів для реалізації кінцевого автомату іншого типу, який не має нічого спільного з початковим класом.

Патерн «State Engine» дозволяє досить легко реалізувати абстракцію кінцевого автомату для опису поведінки моделі певного домену. Далі, при необхідності, є можливість коригувати та модифікувати створений компонент, аби задовільнити нові вимоги.

2.4 Можливості розширення

Представлений у роботі патерн є досить гнучким та динамічним, що проявляється у можливості змінювати та налаштовувати його у залежності від потреб тієї чи іншої програмної системи. Патерн «State Engine» дозволяє проводити наступні модифікації:

– зберігання моделі даних. Контекст можна реалізувати таким чином, що модель даних, яка передається класам станів для використання під час виконання операцій, буде відокремлена від менеджеру подій і буде винесена в окремий клас. При такому підході ускладняється реалізація патерну, але зменшується залежність між класами станів та класом контексту;

– stateless і stateful класи станів. У патерні «State Engine» контекст і класи станів містять посилання на модель даних і менеджер подій. Таким чином, класи станів є stateful (зберігають свій внутрішній стан протягом своєї роботи). Такий підхід не завжди прийнятний, оскільки в деяких ситуаціях витрата пам'яті є критичною. В цьому випадку патерн можна змінити так, щоб класи станів були stateless. При цьому класам станів при кожному виклику методу доведеться передавати параметри на модель даних і менеджера подій. Окрім економії пам’яті це дозволить використовувати об’єкт стану для різних моделей даних, оскільки вони не будуть прив’язані один до одного;

– визначення переходів між станами. У початковій версії переходи між станами задаються в контексті (менеджері подій) за допомогою графу переходів або спеціальних конструкцій мови програмування. Але для певних випадків є

можливість реалізувати дану логіку в окремому класі, що дозволить класу контексту не залежати від конкретної таблиці переходів і мати змогу змінити її у разі необхідності. Це можу бути корисно, якщо суміжні стани можуть змінюватися по-різному в залежності від певних умов.

2.5 Перехід від патерну «State» до патерну «State Engine»

Для програмного забезпечення, яке містить модулі системи, що реалізують патерн «State», дуже важливим критерієм при переході до патерну «State Engine» є можливість провести модернізацію з мінімальною кількістю дій та без значних змін у існуючій кодовій базі, аби зменшити ризики виникнення помилок [15].

Під час виконання даної роботи було розроблено інструкцію, що містить послідовність етапів, необхідних для рефакторингу компонентів системи, які планується модернізувати. Послідовність виконуваних дій є наступною:

– визначьтеся з класом, який відіграватиме роль контексту (менеджеру подій). Це може бути як існуючий клас, який вже залежить від стану, так і новий клас, якщо код станів рознесений по декількох класах;

– створіть загальний контракт (інтерфейс) станів (при можливості використовуйте існуючий). Він повинен описувати методи, які присутні у всіх станах, що мають пряме відношення до контексту. Важливим моментом є те, що не всю поведінку контексту потрібно розміщати у нових класах, а тільки ту, яка є залежною від станів;

– створіть клас (при можливості використовуйте існуючі) для кожного фактичного стану, який реалізує контракт стану. Реалізуйте конкретні класи необхідних станів кодом описаної поведінки компоненту. Після проведених операцій усі методи інтерфейсу стану повинні бути реалізовані в створених класах станів. Також необхідно параметром передати класам станів об’єкт менеджеру подій, аби мати можливість надіслати сповіщення про виникнення події;

– при виділенні поведінки з контексту є вірогідність зіткнутися з тим, що ця поведінка залежить від приватних властивостей або методів контексту, до яких немає доступу з об’єкта стану. Є кілька способів, щоб обійти цю проблему. Найпростіший – залишити поведінку всередині контексту, викликаючи його з об’єкта стану. Іншим способом є реалізація класів станів у вигляді вкладених компонентів до класу контексту, що дозволить їм отримати доступ до всіх приватних членів контексту. Однак цей спосіб підтримується лише деякими мовами програмування (наприклад, Java, C#) та призводить до підвищення зв’язаності класів станів;

– створіть базовий клас події, що буде виникати у класах станів при виконанні певних операцій. Додайте необхідні поля до класу аби передати дані від класу стану до менеджера подій;

– реалізуйте логіку виникнення подій у класах станів, звертаючи увагу на те, що виникнення події може призвести до зміни стану контексту. Кожне виникнення події повинно супроводжуватись відправкою відповідного повідомлення менеджеру подій;

– створіть в контексті поле для зберігання станів, а також таблицю переходів між станами, яка за парою подія-стан буде переводити контекст у новий стан;

– видаліть код, який перемикає стан контексту, з класів конкретних станів, оскільки тепер за це буде відповідати менеджер станів.

Виконання описаних вище дій дозволить гнучко налаштовувати компоненти системи, що мають свій стан, та повторно використовувати класи станів між об’єктами різних типів.

2.6 Відносини з іншими патернами

Першочергова задача патернів – вирішити якусь проблему, яка зустрічається досить часто та при неправильному підході до рішення може призвести до

виникнення нових проблем або унеможливити масштабування модулю. Тобто патерни – це, окрім способу побудови програмного коду певним чином, ще й засіб опису проблем, які призводять до такого рішення [16].

Оскільки проблеми досить часто мають спільні риси, то і патерни, які покликані їх вирішити, мають дещо спільне. Патерн проектування «State Engine» перекликається з наступними шаблонами:

– «Bridge», «Strategy» та «State Engine» (а також трохи «Adapter») мають схожі структури класів – усі вони у своїй основі мають принцип «композиції», тобто делегування роботи іншим компонентам, період життя яких визначається головним (від якого залежать) модулем. Їх відрізняє те, що вони використовуються для вирішення різних проблем;

– «State Engine» можна розглядати як надбудову над патерном «Strategy». Композиція використовується в обох патернах, що дозволяє змінити поведінку основного компоненту шляхом делегування роботи вкладеним допоміжним об’єктам. Проте в «Strategy» ці об’єкти не знають один про одного і жодним чином не пов’язані. У патерні «State» конкретні стани самостійно можуть перемикати контекст головного об’єкта, визначаючи його майбутній стан.

2.7 Подальший розвиток та покращення

Незважаючи на всі переваги патерну «State Engine», він має і певні недоліки, які можуть бути виправлені у майбутніх версіях та після проведення додаткових досліджень та тестувань. Серед основних недоліків, які властиві патерну «State Engine», є наступні:

– необхідність виконувати делегування виконання операцій поточного об'єкта автомату інтерфейсу стану вручну. Мається на увазі необхідність викликати методи поточного стану для кожного такого ж методу класу контексту. Це обмеження можна обійти, використовуючи автоматичну генерацію коду контексту

(що позбавить необхідності самостійно створювати код для кожного контексту) або мови програмування, що підтримує динамічне делегування виконання операцій внутрішнім об’єктам;

– можливе порушення принципу LSP (Liskov Substitution Principle) у випадку, якщо класи станів об’єкту, які реалізують один і той самий абстрактний клас, послаблюють післяумови виконання операцій, тобто гарантують менше (не підтримують певний метод або призводять до виключної ситуації) або посилюють передумови, тобто вимагають більше (умови виклику методів або використання внутрішнього стану більш обмежене, ніж у батьківському класі). У такому випадку можна вирішити проблему заміною абстрактних модулів на конкретні базові класи (методи без реалізації яких будуть генерувати помилку за замовчуванням), від яких будуть наслідувати класи станів.

3 МОДЕЛЬ КІНЦЕВОГО АВТОМАТУ В АСИНХРОННИХ ОПЕРАЦІЯХ

Комп'ютерні програми не рідко обробляють дані з використанням тривалих процесів. Наприклад, отримують дані з бази або проводять складні обчислення. За час виконання однієї операції, ще декілька могли б бути завершені. А простій в роботі призводить до зниження продуктивності і збитків. Асинхронне програмування збільшує ефективність роботи системи, тому що не дозволяє блокувати основний потік виконання.

У синхронному коді кожна операція очікує на завершення попередньої. Тому весь програмний процес може перестати відповідати на виникаючі події, якщо одна з команд виконується дуже довго.

Асинхронний код прибирає блокуючу операцію з основного потоку програми, так що вона залишиться активною, але десь в іншому місці (буде оброблятись іншим потоком або процесом), а обробник може йти далі. Простіше кажучи, головний процес ставить завдання і передає його іншому незалежному процесу.

Асинхронне програмування успішно вирішує безліч завдань. Одне з найважливіших – доступність інтерфейсу користувача.

Як приклад візьмемо додаток, який підбирає фільм за зазначеними користувачем критеріями. Після того, як користувач вибрав параметри, система відправляє запит на сервер, де відбувається підбір відповідних картин.

Обробка може тривати досить довго. Синхронна робота додатку означає, що користувач не зможе взаємодіяти зі сторінкою, поки не прийде результат.

Асинхронний код дозволяє приховати від користувача ці неприємні ефекти і зберегти динамічність сторінки. Після того як дані завантажаться, програма виведе їх на екран.

В цьому випадку головний потік виконання розгалужується, що дозволяє одній частині продовжити займатися інтерфейсом, а іншій виконувати запит.

Тут виникає проблема. Коли запит завершиться в додатковій гілці, як про це

дізнається головна? Як повернути отримане значення в основний потік, якщо це необхідно? Для цього існують події та механізм зворотного виклику.

Якщо запит виконується асинхронно, то він може оповістити всіх охочих про своє закінчення. Програма підписується на це повідомлення і реєструє для нього обробник. Коли прийде час, запит створить подію і повідомить компоненти, що на неї підписались.

Оброблювач продовжує виконувати наступний код, поки не отримає повідомлення. Тоді він перерветься і обробить його.

3.1 Принципи реалізації асинхронності у .NET

Мова програмування C# має асинхронну модель програмування на мовному рівні, яка дозволяє легко писати асинхронний код без необхідності жонглювати зворотними викликами або використовувати бібліотеку, яка підтримує асинхронність. Ця модель базується на патерні асинхронного виконання на основі задач (TAP – Task-based Asynchronous Pattern) [17].

Ядром асинхронного програмування є об'єкти Task та Task<T>, які моделюють асинхронні операції. Вони підтримуються ключовими словами async та await. Модель досить проста в більшості випадків:

– для коду, пов'язаного з введенням/виведенням даних (I/O-based), відбувається очікування на завершення операції, яка повертає Task або Task<T> всередині async методу;

– для коду з операціями, що виконуються на процесорі (CPU-based), відбувається очікування операції, яка запускається у фоновому потоці методом Task.Run.

Ключове слово await надає контроль компоненту системи, який викликав метод з await, і дозволяє інтерфейсу користувача реагувати на події або сервісу бути еластичним.

Прикладом операції з введенням/виведенням є завантаження даних з веб-служби. У цій ситуації вам може знадобитися завантажити деякі дані з веб-служби при натисканні кнопки, але ви не хочете блокувати потік інтерфейсу користувача. Приклад коду для даної ситуації зображено на рисунку 3.1.

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // Цей рядок передасть керування інтерфейсом користувача
    // під час здійснення запиту від веб-служби.
    //
    // Потік інтерфейсу користувача тепер може виконувати інші роботи.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

Рисунок 3.1 – Приклад асинхронної операції вводу/виводу

Прикладом додатку, який має асинхронні операції, що виконуються на процесорі, є мобільна гра, де натискання кнопки може завдати шкоди багатьом ворогам на екрані. Виконання обчислення шкоди може бути дорогим, і якщо це зробити в потоці інтерфейсу користувача, гра може перестати відповідати на дії користувача, коли обчислення виконується.

Найкращий спосіб вирішити це – запустити фоновий потік, який виконує роботу з використанням методу Task.Run, і чекати його результату за допомогою await. Це дозволяє користувальницькому інтерфейсу відповідати без збоїв під час роботи. Приклад коду для даної ситуації зображено на рисунку 3.2.

```
private DamageResult CalculateDamageDone()
{
    // Код пропущено:
    //
    // Здійснює розрахунок і повертає результат цього обчислення.
}

calculateButton.Clicked += async (o, e) =>
{
    // Цей рядок передасть керування інтерфейсом користувача, поки CalculateDamageDone()
    // виконує свою роботу. Потік інтерфейсу користувача може виконувати інші роботи.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

Рисунок 3.2 – Приклад асинхронної операції, виконуваної процесором

Цей код чітко окреслює операцію, що виконується при натисканні кнопки, він не вимагає керування фоновим потоком вручну, і робить це неблокуючим способом.

3.2 Недоліки існуючого підходу до асинхронності у .NET

У платформі .NET для кожного компоненту, який має метод, що складається з асинхронних операцій, компілятором автоматично генерується реалізація кінцевого автомату, що відповідає за такі операції:

– зміна стану компонента – перехід між асинхронними операціями;

– відстеження поточного стану компонента – яка з асинхронних операцій виконується у даний момент;

– збереження стану компонента для можливості його використання у подальших операціях;

– виконання проміжних операцій (синхронних);

– передача об’єкта, який представляє статус виконання асинхронних операцій, модулю, що викликав асинхронний метод, що дозволяє визначити, коли асинхронні операції було завершено та результат їх виконання.

Основним недоліком даного підходу є те, що компілятор, генеруючи кінцевий автомат, нічого не знає про контекст виконання модулів системи, що у певних ситуаціях може призвести до генерації неоптимального коду, що може позначитися на продуктивності даного компонента.

Із основних фрагментів, які найбільше впливають на швидкість виконання коду модулю, можна виділити наступні:

– перехід між станами автомату у випадках, коли його можна повністю уникнути – дане припущення може бути зроблене розробником системи, який відповідає за аналіз послідовності виконання операцій, проте компілятор на даному етапі не має таких можливостей;

– генерація зайвого коду – у певних ситуаціях можна обійтись вбудовуванням певних мовних конструкцій у місця виконання та уникнути певних перевірок змінних методу, знаючи контекст його виконання;

– збереження зайвого стану модулю – компілятор зберігає усі змінні, створенні на початку модулю, у реалізації кінцевого автомату, для подальшого використання під час виконання програмного коду. Але зберігання деяких даних можна було б уникнути якщо вони, наприклад, не впливають на кінцевий результат або використовуються лише один раз. Це призвело би до зменшення кількості виконуваних операцій.

3.3 Модифікація алгоритму виконання асинхронних операцій

Усі недоліки існуючого підходу зумовлені тим, що компілятор використовує один і той самий підхід для побудови кінцевого автомату в усіх ситуаціях, що, з однієї сторони, полегшує використання даного підходу у різних контекстах, а з іншої – призводить до перевантаження модулю зайвими конструкціями, на виконання яких витрачається процесорний час.

У програмних системах, де час виконання операцій є критичним, даний спосіб не є найефективнішим, що зумовлює пошук іншого методу вирішення задачі.

У даній роботі пропонується модифікувати існуючий алгоритм, залишивши основні будівельні блоки, змінивши при цьому процес створення контексту виконання операцій. Тобто замість генерації кінцевого автомату компілятором, створити свою реалізацію, яка буде використовувати ті ж самі компоненти, але більш ефективно у рамках певної задачі.

У якості ситуації, на якій продемонструємо роботу модифікованого алгоритму, оберемо задачу відправки http запиту на веб-сервер для завантаження списку користувачів ресурсу, отримання відповіді та збереження результату до

файлу у форматі JSON з поверненням кількості записаних байтів. Ця ситуація є досить синтетичною, оскільки у реальних проектах зустрічається досить рідко, так як порушує деякі з існуючих принципів проектування та розробки програмного забезпечення, але вона досить добре відображає саму суть виконання асинхронних операцій та побудову коду для її підтримки. Дана задача може бути представлена у вигляді послідовності операцій програмного коду, як зображено на рисунку 3.3 (використовуючи існуючий підхід до асинхронності).

```
/// <summary>
/// Завантажує список користувачів та збзерігає їх до файлу
/// </summary>
/// <param name="url">Адрес ресурсу для завантаження користувачів</param>
/// <param name="filePath">Шлях до файлу, куди буде збережена відповідь веб-сервера</param>
/// <param name="append">Параметр, який вказує чи потрібно дописувати дані до файлу, або переписати існуючі</param>
/// <returns></returns>
1 reference | 0 exceptions
public async Task<long> GetUsersAndSaveToFileAsync(string url, string filePath, bool append)
{
    using var httpClient = new HttpClient();

    var response = await httpClient.GetAsync(url);
    var usersStream = await response.Content.ReadAsStreamAsync();

    await using var fileStream = new FileStream(filePath, append ? FileMode.Append : FileMode.Create, FileAccess.Write);
    usersStream.CopyTo(fileStream);

    return usersStream.Length;
}
```

Рисунок 3.3 – Послідовність операцій програмного коду для описаної задачі

Продемонстрований код має структуру, яка використовує існуючий підхід до написання асинхронного коду у .NET [18]. Під час компіляції коду компілятор автоматично згенерує реалізацію кінцевого автомату, фрагменти якого зображено на рисунках 3.4-3.6.

```
[AsyncStateMachine(typeof(<GetUsersAndSaveToFileAsync>d__0))]
[DebuggerStepThrough]
public Task<long> GetUsersAndSaveToFileAsync(string url, string filePath, bool append)
{
    <GetUsersAndSaveToFileAsync>d__0 stateMachine = new <GetUsersAndSaveToFileAsync>d__0();
    stateMachine.<>t__builder = AsyncTaskMethodBuilder<long>.Create();
    stateMachine.<>4__this = this;
    stateMachine.url = url;
    stateMachine.filePath = filePath;
    stateMachine.append = append;
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

Рисунок 3.4 – Основний метод після заміни інструкцій компілятором

```
[CompilerGenerated]
private sealed class <GetUsersAndSaveToFileAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;

    public AsyncTaskMethodBuilder<long> <>t__builder;

    public string url;

    public string filePath;

    public bool append;

    public P <>4__this;

    private HttpClient <httpClient>5__1;

    private HttpResponseMessage <response>5__2;

    private Stream <usersStream>5__3;

    private FileStream <fileStream>5__4;

    private HttpResponseMessage <>s__5;

    private Stream <>s__6;

    private object <>s__7;

    private int <>s__8;

    private long <>s__9;

    private TaskAwaiter<HttpResponseMessage> <>u__1;

    private TaskAwaiter<Stream> <>u__2;
```

Рисунок 3.5 – Клас абстракції кінцевого автомату та поля зі станом

```
switch (num)
{
    default:
        awaiter3 = <httpClient>5__1.GetAsync(url).GetAwaiter();
        if (!awaiter3.IsCompleted)
        {
            num = (<>1__state = 0);
            <>u__1 = awaiter3;
            <GetUsersAndSaveToFileAsync>d__0 stateMachine = this;
            <>t__builder.AwaitUnsafeOnCompleted(ref awaiter3, ref stateMachine);
            return;
        }
        goto IL_009f;
    case 0:
        awaiter3 = <>u__1;
        <>u__1 = default(TaskAwaiter<HttpResponseMessage>);
        num = (<>1__state = -1);
        goto IL_009f;
    case 1:
        awaiter2 = <>u__2;
        <>u__2 = default(TaskAwaiter<Stream>);
        num = (<>1__state = -1);
        goto IL_0124;
    case 2:
        {
            awaiter = <>u__3;
            <>u__3 = default(ValueTaskAwaiter);
            num = (<>1__state = -1);
            goto IL_0216;
        }
        IL_0124:
        <>s__6 = awaiter2.GetResult();
```

Рисунок 3.6 – Фрагмент логіки переходу між станами кінцевого автомату

Проаналізувавши наведені фрагменти коду можна помітити, що компілятор створив велику кількість змінних для зберігання стану кінцевого автомату та початкового методу при виконанні коду. Окрім власне стану автомату, були створені поля для локальних змінних методу, результатів виконання методів, тимчасових змінних, необхідних для роботи логіки переходу між станами.

Згенерований код зміни станів кінцевого автомату (що зумовлює та визначає виконувану операцію у певний момент часу) виглядає досить громіздким, так як містить багато інструкцій розгалуження коду.

Модифікована версія алгоритму полягає у видаленні непотрібних (зайвих) конструкцій програмного коду для зменшення об'єму використовуваної пам'яті та часу виконання методу [19]. Даний процес у певній мірі схожий на процес рефакторингу, але у даному випадку ми проводимо його над автоматично згенерованим кодом.

Спрощення логіки переходу між станами кінцевого автомату та набору полів, що визначають дані контексту виконання, дозволяє пришвидшити процес виконання інструкцій програмного коду.

Модифіковану версію реалізації кінцевого автомату зображено на рисунках 3.7-3.8.

```
private sealed class AsyncStateMachine : IAsyncStateMachine
{
    public int state;

    public AsyncTaskMethodBuilder<long> builder;

    public string url;

    public string filePath;

    public bool append;

    private HttpClient httpClient;

    private long result = default(long);

    private TaskAwaiter<HttpResponseMessage> awaiterResponse;

    private TaskAwaiter<Stream> awaiterStream;

    private ValueTaskAwaiter awaiterValueTask;
```

Рисунок 3.7 – Поля зі станом та клас реалізації кінцевого автомату модифікованої версії алгоритму

```
void IAsyncStateMachine.MoveNext()
{
    try
    {
        if ((uint)state > 2u)
        {
            httpClient = new HttpClient();
        }
        switch (state)
        {
            default:
                awaiterResponse = httpClient.GetAsync(url).GetAwaiter();
                if (!awaiterResponse.IsCompleted)
                {
                    state = 0;
                    AsyncStateMachine stateMachine = this;
                    builder.AwaitUnsafeOnCompleted(ref awaiterResponse, ref stateMachine);
                    return;
                }
                goto IL_009f;
            case 0:
            IL_009f:
                var responseMessage = awaiterResponse.GetResult();
                awaiterStream = responseMessage.Content.ReadAsStreamAsync().GetAwaiter();
                if (!awaiterStream.IsCompleted)
                {
                    state = 1;
                    AsyncStateMachine stateMachine = this;
                    builder.AwaitUnsafeOnCompleted(ref awaiterStream, ref stateMachine);
                    return;
                }
                goto IL_0124;
            case 1:
            IL_0124:
                var usersStream = awaiterStream.GetResult();
                var fileStream = new FileStream(filePath, append ? FileMode.Append : FileMode.Create, FileAccess.Write);
                usersStream.CopyTo(fileStream);
                result = usersStream.Length;
                awaiterValueTask = ((IAsyncDisposable)fileStream).DisposeAsync().GetAwaiter();
```

Рисунок 3.8 – Фрагмент логіки переходу між станами кінцевого автомату у модифікованій версії алгоритму

Провівши аналіз отриманого коду, можна зробити висновки, що його версія набагато менша за оригінальну та містить тільки необхідні конструкції та властивості з даними, що використовуються під час виконання програмного коду.

Серед основних змін та покращень структури коду та процесу виконання у даній версії можна виділити наступні:

– кількість змінних, що використовуються для зберігання додаткового стану та даних модулю, зменшилась на 8 (з 18 до 10). Було видалено копії існуючих змінних, які використовувались лише у одній операції. У певних місцях замість збереження даних до глобальної змінної класу було створено локальну змінну, що має набагато менший час життя та створюється лише за необхідності;

– було зменшено кількість інструкцій програмного коду, що виконуються для кожного конкретного стану кінцевого автомату. Серед основної кількості

видалених команд були ті, що повторно ініціюють змінні або ініціюють глобальну змінну через локальну, яка більше ніде не використовується;

– кількість конструкцій try/catch було зменшено з 2 до 1 шляхом поєднання інструкцій з обох логічних блоків;

– загальний розмір програмного коду було зменшено з 8933 байтів до 4603 байтів (з 225 рядків коду до 120), що майже у 2 рази менше початкової версії. Дане покращення призводить до зменшення розміру скомпільованого файлу, швидкості компіляції та часу виконання програмного коду, оскільки кількість команд, які необхідно опрацювати, стала меншою;

– загальна структура моделі кінцевого автомату була значно спрощена та логічно перебудована (без зміни функціоналу), що дозволяє швидко зрозуміти основне призначення даного програмного блоку та легко вносити зміни до модулю системи у разі необхідності.

3.4 Недоліки нової версії алгоритму та способи її покращення

Модифікована версія алгоритму виконання асинхронних операцій на основі кінцевого автомату, запропонована у даній роботі, значно покращує оригінальний алгоритм, що автоматично генерується компілятором платформи .NET (мова програмування C#). Але оновлена версія має свої певні недоліки та можливості для покращення, що дозволить зробити алгоритм ще більш швидким, зменшити кількість пам’яті, необхідної для виконання програмного коду системи, та полегшити процес побудови модулю для виконання асинхронних операцій у рамках певної задачі.

Серед основних потенційних змін, що дозволять покращити описаний алгоритм, є наступні:

– автоматична генерація оновленого алгоритму для кожної конкретної задачі. На даному етапі для роботи алгоритму його необхідно кожного разу вручну

структурувати (перебудовувати початковий, автоматично згенерований), що потребує певну кількість часу та може призвести до появи помилок. Важливим оновленням буде створення підсистеми, яка дозволить автоматично створювати нові версії алгоритму для певної задачі;

– зменшення внутрішнього стану кінцевого автомату. У даному пункті мова йдеться не про стан самого автомату, а про внутрішній стан усього модулю системи, тобто змінні та поля з даними. Одним з можливих покращень може бути винесення усього стану в окрему структуру даних, що зменшить його дублювання (оскільки у даній версії копія стану передається з основного методу компоненту до абстракції кінцевого автомату);

– зменшення кількості станів кінцевого автомату у певних ситуаціях. Наприклад, можна об’єднати дві або більше схожих за способом обробки асинхронні операції, почавши їх одночасно та змінивши стан кінцевого автомату лише коли усі з них завершаться.

4 ТЕСТУВАННЯ РОЗРОБЛЕНИХ МЕТОДІВ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Результати, отримані під час виконання даної роботи, дозволяють значно пришвидшити процес виконання асинхронних операцій у програмному коді, який використовує реалізацію кінцевого автомату для керування набором команд, та полегшити процес створення об'єктно-орієнтованої моделі компоненту, який може міняти свою поведінку в залежності від свого поточного стану.

Задля визначення більш конкретного значення приросту продуктивності роботи модулів системи при використанні розроблених алгоритмів та методик, необхідно провести ретельне тестування описаних результатів на реальних даних та у реальних ситуаціях.

4.1 Порівняння використання патернів «State» та «State Engine»

Для аналізу використання патерну «State Engine» кількісні характеристики не можуть бути застосовані, оскільки у даному випадку користувач патерну отримує вигоду не у вигляді зменшеного об'єму необхідної пам'яті або меншого часу виконання, а у вигляді дизайну системи, який може бути легко масштабований, змінений, повторно використаний та є легким у підтримці.

У розробленому патерні «State Engine» було усунено основні недоліки та складності використання патерну «State», а саме: ускладнене повторне використання класів станів, оскільки вони знають один про одного та сильно залежні між собою, що робить неможливим їх використання у ситуації, де порядок переходу між станами інший при тих самих станах; управління станами класом контексту, оскільки вони самі визначають наступний стан основної моделі та явно його вказують, в залежності від умов бізнес логіки, що розкриває деталі реалізації

контексту класам станів; розподіл логіки переходу між станами контексту по конкретним класам станів, що ускладнює логіку зміни стану та розділяє її по різним модулям системи.

Отримана модифікована версія патерну централізує логіку переходу між станами у класі контексту (або окремому класі, що відповідає за побудову графу переходів та шляхом композиції асоціюється з класом контексту), що дає краще розуміння системи та можливість внесення необхідних змін у одному місці.

У новій версії патерну класи станів більше не залежать та не знають про інші стани, що дозволяє використовувати їх повторно для проектування нових компонентів, що можуть мати такі ж самі стани, але перехід між якими може відбуватись у абсолютно іншому порядку. Це досягається шляхом виділення окремого інтерфейсу EventManager (менеджер подій), який дозволяє класам станів передати інформацію про подію, яка виникла, що призведе до зміни стану контексту в залежності від його поточного стану. Реалізація EventManager може бути як у самому класі контексту, так і винесена в окремий клас (як у випадку з реалізацією графу переходів між станами).

Таким чином, використання отриманої версії патерну дозволяє полегшити процес проектування та реалізації компонентів системи, поведінка яких визначається їх поточним станом. Дана модифікація дає можливість приділяти більше уваги процесу побудови логічної структури програмної системи, а не замислюватись над проблемами повторного використання отриманої архітектури модулів у майбутньому.

4.2 Проведення тестів продуктивності модифікованої версії алгоритму виконання асинхронних операцій

Для тестування отриманої версії алгоритму виконання асинхронних операцій необхідно визначити показники продуктивності по декількох аспектах:

– розмір пам’яті, необхідної для виконання програмного коду;

– загальний час виконання інструкцій програмного коду системи.

Для проведення замірів описаних показників продуктивності будемо використовувати пакет BenchmarkDotNet для платформи .NET, який розповсюджується за ліцензією MIT [20].

Для визначення показників було проведено 50 виконань кожної версії алгоритму, результати яких продемонстровано на рисунку 4.1.

Method	Mean	StdDev	Min	Median	Max	Ratio	Allocated
Modified	156.1 ms	4.27 ms	148.4 ms	155.4 ms	166.6 ms	0.78	79.26 KB
Original	199.8 ms	4.41 ms	186.3 ms	201.1 ms	203.7 ms	1.00	89.82 KB

Рисунок 4.1 – Результати заміру показників продуктивності

На даному рисунку рядок, у якому значення стовпця «Method» є «Modified», відображає дані модифікованої версії алгоритму, а рядок зі значенням «Original» – результати оригінальної версії.

На даному рисунку відображені наступні показники результату заміру продуктивності:

– «Mean» – середнє арифметичне значення часу виконання серед усіх замірів для певного методу;

– «StdDev» – середньоквадратичне відхилення серед усіх замірів;

– «Min» – мінімальне значення часу виконання для певного методу;

– «Median» – значення, яке розділяє більшу половину усіх замірів (50% перцентиль);

– «Max» – максимальне значення часу виконання для певного методу;

– «Ratio» – середнє значення відношень значень відповідних вимірів кожного методу до оригінального методу;

– «Allocated» – середній розмір виділеної пам’яті для певного методу.

Виходячи з отриманих даних, можна побачити, що виграш в часі виконання

модифікованої версії алгоритму складає 22% (стовпець «Ratio»), порівняно з оригінальною версією. Тобто при часі виконання оригінального алгоритму в 199.8 мс, модифікована версія виконується за 156.1 мс.

Для більш наочної демонстрації використання виділеної пам'яті для кожного методу, використаємо інструменти діагностики, вбудовані в середу розробки Visual Studio 2019, які дозволяють отримати знімок об'єктів, створених платформою на певний момент часу [21].

Спочатку зробимо знімок пам'яті з об'єктами в системній купі до виконання методу з необхідним алгоритмом, а потім – після, щоб була можливість визначити об'єкти (їх кількість та розмір пам'яті), які були створені саме під час роботи алгоритму.

Знімки купи з об'єктами оригінального та модифікованого методів представлені на рисунках 4.2-4.3.

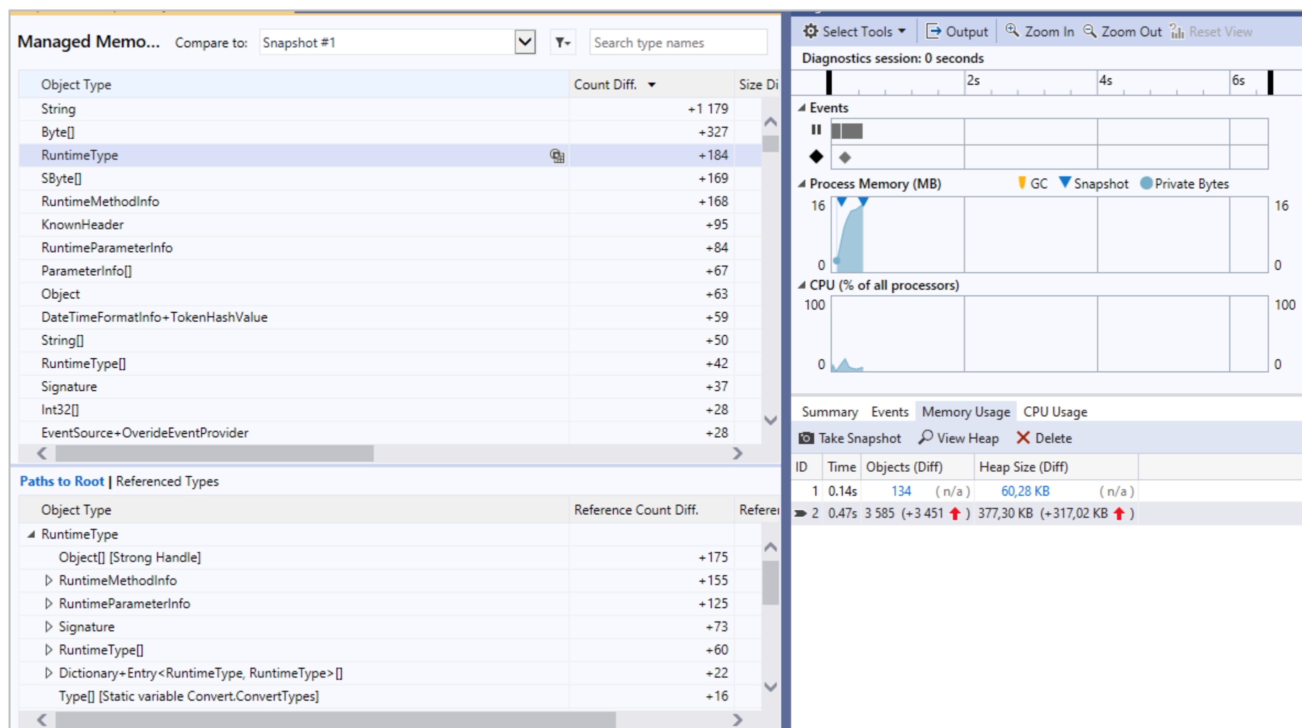

Рисунок 4.2 – Використання пам'яті модифікованою версією алгоритму

На даному рисунку показано, що під час виконання модифікованої версії алгоритму було створено 3451 нових об'єктів (окрім об'єктів, які були явно

створені в програмному коді , включає в себе також об’єкти, створені системними модулями платформи .NET) загальним розміром 317,02 КБ.

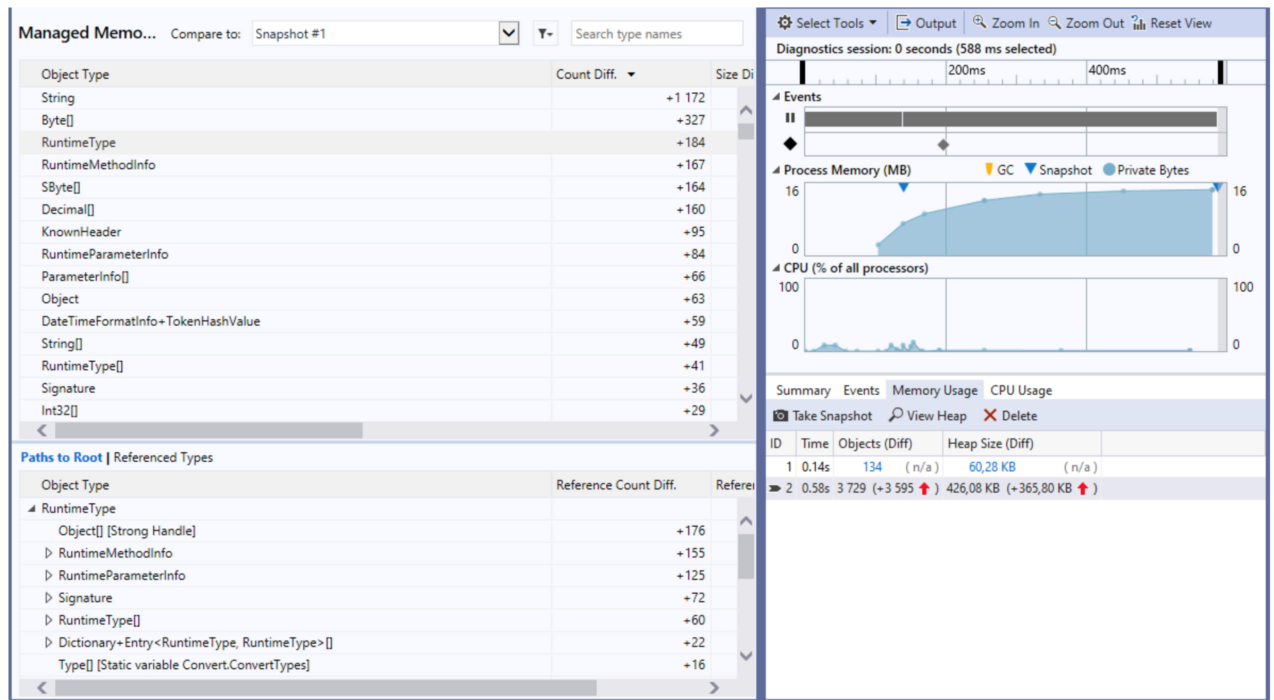

Рисунок 4.3 – Використання пам’яті оригінальною версією алгоритму

На даному рисунку показано, що під час виконання оригінальної версії алгоритму було створено 3595 нових об’єктів, що на 144 об’єкти більше, ніж у модифікованій версії, загальним розміром 365,8 КБ (що на 48,78 КБ більше).

Отримані дані можуть відрізнятись від продемонстрованих на рисунках при послідовних запусках, оскільки існує велика кількість факторів, що впливають на отримані результати (час отримання відповіді на http запит, очікування виконання I/O операцій під час роботи з файлом, виконання збірки сміття та інше). Але все одно можна зробити висновок, що отримані результати для модифікованої версії є кращими та більш продуктивними, ніж для оригінальної версії алгоритму, що може бути досить критичним для певного виду програмних систем.

ВИСНОВКИ

Метою атестаційної роботи є дослідження використання кінцевих автоматів та їх об'єктно-орієнтованих моделей при проектуванні та реалізації поведінки програмних систем. В першу чергу було проведене дослідження використання та модернізації патерну проектування «Стан», який базується на кінцевих автоматах, та конструкцій виконання асинхронних операцій, які побудовані на основі кінцевого автомату.

У результаті роботи було проведено аналіз предметної галузі та виявлені існуючі проблеми використання кінцевих автоматів та їх об’єктно-орієнтованих реалізацій (абстракцій) при створенні програмного забезпечення.

У роботі було виконано усі поставлені задачі, що підтверджується наступними отриманими результатами:

– розроблено патерн проектування, який дозволяє об’єктам змінювати свою поведінку в залежності від їх стану, з усуненням основних недоліків існуючого патерну «Стан»;

– запропоновано метод модифікації вихідного коду програми для використання нового патерну з мінімальною кількістю змін, які можуть привести до появи помилок;

– модифіковано алгоритм виконання асинхронних операцій з використанням моделі кінцевого автомату, що дозволяє зменшити кількість переходів між станами автомату.

Значення продуктивності отриманих результатів були порівняні з оригінальними версіями, що дозволило зробити висновок про покращення значень кількісних та якісних характеристик щодо використання модифікованих версій алгоритмів та патернів.

Практичне значення роботи полягає в тому, що отримані результати спрощують розробку, підтримку і супроводження програмних систем, покращують час виконання програмного коду модулю системи та можуть бути використані на

практиці при написанні та проектуванні їх поведінки.

Запропонований алгоритм дозволяє підвищити швидкість виконання процесу управління набором асинхронних операцій (зменшивши час виконання та об'єм необхідної пам'яті), що у разі високонавантажених систем має дуже велике значення.

Запропонований патерн дозволяє проектувати ті модулі системи, поведінка яких залежить (визначається) поточним станом, та які можуть його змінювати під час роботи, забезпечуючи при цьому повторне використання класів станів, незалежність між ними (слабку зв'язаність) та централізацію логіки переходів між станами (менеджер подій).

Запропонований метод рефакторингу коду програмної системи дозволяє легко перейти від використання патерну «State» до патерну «State Engine», розробленого у даній роботі, мінімізуючи шанси появи помилок.

Для оцінки отриманих результатів було проведено обширне, повноцінне тестування, результати якого було порівняно з існуючими версіями алгоритмів та патернів. Їх аналіз доводить, що використання патерну проектування «State Engine» покращує та полегшує процес проектування програмних систем, а модифікований алгоритм виконання асинхронних операцій виконується швидше, що досягається за рахунок зменшення кількості виконуваних операцій та набору створюваних об'єктів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Голян В. В., Бітюкова Є. І. Дослідження технологій та інструментів управління проектами// Монографія: SCIENCE, RESEARCH, DEVELOPMENT#27/ NECHNICS AND TECHNOLOGY – 2020. – 33 с.

2. Голян В. В., Кравченко О. К. Порівняння моделей життєвих циклів програмного забезпечення з метою виявлення найефективнішого// Збірник наукових праць ХНУ ПС № 2 (157) – 2019. – 6 с.

3. Фаулер М., Бек К., Брант Д., Опдайк У., Робертс Д. Рефакторинг: улучшение проекта существующего кода. – М.: «Діалектика», 2019. – 448 с.

4. Finite-State Machines: Theory and Implementation. URL: https://gamedevelopment.tutsplus.com/tutorials/finite- state- machines- theory- and- im plementation--gamedev-11867 (дата звернення: 20.02.2021).

5. Кузнєцов О. П., Адельсон-Вельський Г. М. Автоматы. Дискретная математика для инженера. – М.: «Энергия», 1980. – 344 с.

6. Математичні схеми моделювання систем. URL: https://studfile.net/preview/10069599/page:12/ (дата звернення: 07.03.2021).

7. Серебряков В. А., Галочкін М. П., Гончар Д. Р., Фуругян М. Г. Теория и реализация языков программирования. – М.: «МЗ-Пресс», 2006. – 431 с.

8. Гамма Е., Хелм Р., Джонсон Р., Вліссідес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования – М.: «Вільямс», 2015. – 368 с.

9. What is a design pattern. URL: https://refactoring.guru/design-patterns/what-is-pattern (дата звернення: 16.03.2021).

10. Ларман К. Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование.: Пер. з англ. – М.: «Вільямс», 2006. – 736 с.

11. Jordan K. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. – NY: «Williams», 2006. – 736 с.

12. Басс Л. Архитектура программного обеспечения на практике. – СПб.: «Питер», 2006. – 576 с.

13. V. Apukhtin, M. Shirokopetleva, V. Skovorodnikova. The Relevance of Using Message Brokers in Robust Enterprise Applications// IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 2019. – P. 305-309.

14. Мейер Б. Объектно-ориентированное конструирование программных систем. – М.: «Русская Редакция», 2005. – 1204 с.

15. Мартін Р. Чистый код. Создание, анализ и рефакторинг. – СПб.: «Питер», 2010. – 464 с.

16. Макконнелл С. Совершенный код. Мастер-класс. – М.: «Русская редакция», 2010. – 896 с.

17. Asynchronous programming. URL: https://docs.microsoft.com/en-us/dotnet/csharp/async (дата звернення: 24.03.2021).

18. Davies A. Async in C# 5.0: Unleash the Power of Async. – B: «O'Reilly Media, Inc.», 2012. – 108 c.

19. Bob Wescott. The Every Computer Performance Book. – C: «Createspace Independent Pub», 2013. – 222 c.

20. Замеряем производительность с помощью BenchmarkDotNet. URL: https://habr.com/ru/post/277177/ (дата звернення: 17.04.2021).

21. Analyze memory usage by using the .NET Object Allocation tool. URL: https://docs.microsoft.com/en-us/visualstudio/profiling/dotnet-alloc-tool?view=vs-2019 (дата звернення: 24.04.2021).