МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

КАФЕДРА ІНФОРМАТИКИ ФАКУЛЬТЕТУ ІНФОРМАТИКИ

Розробка системи оркестрування та управління процесами розподілених систем

Текстова частина до курсової роботи
за спеціальністю 121 «Інженерія програмного забезпечення»

Керівник курсової роботи
Андрощук М.В.

_____
(*підпис*)
“____” _____ 2024 р.

Виконав студент Григоренко С.С.

_____
(*підпис*)
“____” _____ 2024 р.

Київ – 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики

Доцент, к. ф-м. н. Гороховський С.С.

“___” _____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Григоренку Сергію Сергійовичу

факультету інформатики 3 курсу бакалаврської програми

ТЕМА: Розробка системи оркестрування та управління процесами розподілених систем

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Календарний план

Анотація

Вступ

Огляд існуючих рішень та формулювання вимог до системи

Процес розробки та архітектурна концепція: використання найкращих практик

Тестування та демонстрація функціоналу

Висновки

Використані джерела

Дата видачі “___” _____ 2023 р.

Керівник_____

(підпис)

Завдання отримав_____

(підпис)

Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Визначення теми курсової роботи	Жовтень 2023	
2.	Отримання завдання на курсову роботу	Жовтень 2023	
3.	Огляд літератури	Листопад - Грудень 2023	
4.	Написання теоретичної частини курсової роботи	Січень 2024	
5.	Написання програмної реалізації	Лютий 2024	
6.	Написання практичної частини курсової роботи	Березень - Квітень 2024	
7.	Захист курсової роботи	Травень 2024	

Студент _____
Керівник _____

“_____”_____ 2023

АНОТАЦІЯ

У даній курсовій роботі розглянуто процес розробки та впровадження фреймворку для управління та оркестрування процесів у мікросервісній архітектурі.

Фреймворк спрямований на створення та керування послідовністю кроків, що складають бізнес-процес, та надає зручний інтерфейс для конфігурування та виконання цих процесів.

У роботі детально описано імплементацію скінченних автоматів для опису процесів, використання принципів функціонального програмування та шаблонів проєктування для створення ефективної та масштабованої архітектури фреймворку.

Основні переваги розробленого фреймворку полягають у простоті конфігурації, гнучкості в налаштуванні користувацьких процесів, а також надійності та ефективності виконання завдань. Робота докладно розглядає використані технології, інструменти та підходи, такі як Java Concurrency API, Google Truth, JUnit Jupiter, Mockito, Gradle, які допомогли забезпечити успішний розвиток і реалізацію проекту.

Отже, курсова робота пропонує детальний огляд архітектури та реалізації фреймворку для оркестрації у мікросервісних застосунках. Було досліджено різноманітні технології, інструменти та підходи, які вплинули на створення системи. Робота слугує цінним джерелом інформації для тих, хто цікавиться розробкою архітектурно складних систем у сучасних умовах.

ЗМІСТ

ВСТУП

У цьому дослідженні мета полягає в розробці системи, спрямованої на моделювання та оркестрацію багатокрокових операцій у розподілених системах. Одним із ключових аспектів цього дослідження є застосування принципів програмної інженерії, таких як Domain Driven Design (DDD), багатопотоковість у мові програмування Java, використання шаблонів програмування, API дизайн та функційне програмування засобами Java.

У сучасному розробці програмного забезпечення, мікросервісна архітектура веб-додатків стає стандартом індустрії, що пропонує сепарацію функціоналу системи та ізоляцію програмних модулів як у коді, так і при розгортанні. Однак, разом із цим виникають нові виклики, зокрема, складність комунікації між сервісами, що може негативно впливати на швидкість розробки та призводити до несправностей програмного комплексу.

Одним із звичних розв'язків цього питання є асинхронний обмін повідомленнями через message-broker. Цей підхід пропонує абстракцію у вигляді черг або потоків подій, що доповнює ідею мікросервісів. Проте, його недолік полягає в складності представлення серії операцій у коді через відсутність контексту між етапами багатокрокового сценарію, що ускладнює зневадження програм та розуміння відповідальності певних ділянок коду.

З урахуванням цих викликів, головною метою цього дослідження є розробка системи, яка дозволить декларативно описувати складні операції за допомогою загальновживаної мови в межах домену, спрощуючи процес розробки та покращуючи зрозуміння відповідальності різних частин коду.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ФОРМУЛЮВАННЯ ВИМОГ ДО СИСТЕМИ

Цей розділ присвячено огляду існуючих рішень, що вже довго користуються довірою розробників та їх клієнтів в усьому світі. Найвдаліші практики та ідеї дизайну увійдуть до вимог власної системи оркестрації процесів.

1.1 Огляд існуючих рішень

1.1.1 Apache Airflow

Airflow був розроблений в Airbnb і був випущений як відкрите програмне забезпечення в 2015 році. Apache Airflow є однією з найпопулярніших відкритих платформ для оркестрації розподілених систем. Вона використовує модель орієнтованих ациклічних графів, де кожен вузол представляє собою завдання, а ребра визначають залежності між завданнями [1]. Один із основних принципів дизайну Airflow – «код як конфігурація». Усі робочі процеси в системі визначаються Python кодом, що полегшує створення, керування та налогодження процесів. Airflow надає широкий набір вбудованих операторів для виконання різних завдань, а також має можливість розширення за рахунок створення власних операторів та додатків. Відмінною рисою Apache Airflow є його масштабованість та здатність працювати з розподіленими робочими процесами, що робить його ідеальним вибором для великих обчислювальних завдань.

1.1.2 Prefect

Prefect – проєкт з відкритим кодом, був запущений у 2018 році і за короткий час зарекомендував себе як потужний та простий у використанні інструмент для оркестрації робочих процесів. Prefect є сучасною платформою для оркестрації робочих процесів, яка надає Python-native інтерфейс для визначення робочих процесів [2]. Однією з основних переваг Prefect є його простота використання та гнучкість, оскільки він дозволяє розробникам визначати робочі процеси безпосередньо за допомогою коду Python. Prefect також надає інструменти для моніторингу та візуалізації робочих процесів, що полегшує відстеження їх виконання та виявлення проблем, що робить його хорошим вибором для великого спектру задач, від простих завдань до складних аналітичних обчислень.

1.1.3 Dagster

Dagster є іншою цікавою платформою для оркестрації робочих процесів. Проєкт був представлений в 2019 році і швидко набув популярності в спільноті даних та аналітики. Dagster використовує сильну систему типів та наголошує на створенні модульних робочих процесів, які можна перевикористовувати та комбінувати [3]. Бібліотека також має глибоку інтеграцію з інструментами машинного навчання, що робить його популярним вибором для створення та оркестрування робочих процесів у галузі аналізу даних та машинного навчання.

Кожна з розглянутих платформ для оркестрації робочих процесів має свої унікальні переваги та особливості. Apache Airflow відомий своєю масштабованістю та можливістю працювати з розподіленими робочими

процесами, Prefect відрізняється простотою використання та гнучкістю, а Dagster має сильну систему типів та найбільш підходить для обробки даних.

Усі платформи пропонують інструменти для визначення робочих процесів у вигляді орієнтованих ациклічних графів, де завдання представлені вузлами, а залежності між завданнями - ребрами. Цей підхід дозволяє декларативно описувати послідовність та логіку виконання робочих процесів. Крім того, всі три платформи надають засоби для моніторингу, керування та налагодження робочих процесів, що дозволяє користувачам ефективно керувати та відстежувати виконання завдань у реальному часі. Загальною характеристикою всіх цих платформ є їх спрямованість на підтримку складних робочих процесів та потоків даних шляхом надання інструментів для декларативного визначення, ефективного виконання та керування цими процесами.

1.2 Постановка вимог до системи

У ході аналізу предметної області та встановлення вимог до системи було сформульовано ряд ключових пунктів, які визначають специфіку розробки та функціональні можливості фреймворку.

1.2.1 Зручний API

Переважна увага була спрямована на створення зручного клієнтського API, оскільки його наявність вважається запорукою якісного програмного забезпечення [4]. Метою було створення API, що відповідає принципам простоти та лаконічності, що сприяє зручності його використання та уникненню помилок при взаємодії з користувачами.

1.2.2 Клієнтський код має легко тестуватись

Додатково, було відзначено важливість тестування користувацьких процесів програмними засобами, оскільки це є важливим кроком у забезпеченні якості програмного забезпечення та виявленні помилок на ранніх стадіях розробки.

1.2.3 Підтримка асихронних завдань

Крім того, важливими вимогами є можливість обробки результатів отриманих асинхронно, що відповідає вимогам екосистеми мікросервісів, та неблокуюче виконання процесів, що забезпечить оптимізацію часу виконання та підвищення продуктивності системи.

1.2.4 Наявність документації для розробників

Наявність вичерпної документації та прикладів застосування є важливими елементами для забезпечення доступності та якості фреймворку, що допоможе зрозуміти користувачам як використовувати систему та вирішувати потенційні проблеми.

РОЗДІЛ 2. ПРОЦЕС РОЗРОБКИ ТА АРХІТЕКТУРНА КОНЦЕПЦІЯ: ВИКОРИСТАННЯ НАЙКРАЩИХ ПРАКТИК ТА ПІДХОДІВ

2.1 Технологічний стек

Для створення фреймоворку було обрано мову програмування Java, як поширений інструмент для створення мікросервісних застосунків. Було обрано 17 версію, як одну з найбільш сучасних, що вже повноцінно підтримується хмарними сервісами.

Принциповим моментом стала відсутність залежностей у проекті, крім логування та засобів тестування. Цей вибір був зроблений для спрощення підтримки системи та зниження вхідного порогу до фреймворка.

В якості інструменту для автоматизації збирання та управління залежностями проєкту було використано Gradle. Це гнучкий інструмент, який надає можливість створення багатомодульних додатків, написання власних сценаріїв збірки та розвинену систему плагінів.

2.2 Концепт системи

Концепт системи передбачає розподіл на три шари абстракцій, що відповідають різним аспектам функціонування програмного забезпечення. Система, оскільки використовується мова Java, має базуватись на найкращих практиках об’єктно-орієнтованого дизайну задля розподілу відповідальності між компонентами програми та уникнення сильного зв’язування між класами. Також система повинна користуватись головними перевагами функціонального програмування: незмінність об’єктів, експресивність коду, використання монад та функцій вищих порядків.

2.3 Функціональне програмування в Java

2.3.1 Функціональне програмування та його роль у сучасній розробці

Функційне програмування - це парадигма програмування, яка покладає основний акцент на використання функцій як основної будівельної одиниці програмного коду. Вона ставить перед собою завдання максимально уникати змінного стану та мутації даних, спрощуючи таким чином розробку та розуміння програм.

Роль функційного програмування в сучасних підходах до розробки полягає в тому, що воно сприяє покращенню читабельності, підтримці, тестуванню і рефакторингу коду. Відокремлення даних та логіки, яке притаманне функційному програмуванню, дозволяє створювати більш надійне та масштабоване програмне забезпечення.

2.3.2 Реалізація функціонального програмування в Java

У світі Java функційне програмування отримало значний розвиток з введенням лямбда-виразів у восьмій версії мови. Це дозволило розробникам використовувати більш елегантний та зрозумілий синтаксис при написанні коду, а також впроваджувати багатопоточні та паралельні операції з меншими зусиллями. Також, вбудовані функційні інтерфейси та API допомагають розробникам реалізовувати функціональні підходи до розв'язання завдань у Java.

2.4 Задача

Задача є найнижчим шаром абстракції, вона визначає атомарні дії, які можуть включати в себе обчислення, звернення до сервера або асинхронний виклик мікросервісу. У коді, задача визначена як Java інтерфейс з анотацією `@FunctionalInterface`, що позначає, що вона має лише один метод і може використовуватись у лямбда-виразах, які спрощують код та спонукають до дотримання принципу єдиної відповідальності. Ця анотація позначає, що інтерфейс має виключно один метод і може використовуватись у лямбда-виразах, що були введені до стандарту мови у восьмій версії.

```
@FunctionalInterface
public interface Task<P, V> {
   /**
    * Submits the task to the receiver and awaits its
completion.
    *
    * @param payload required argument to complete the task
    */
   Result<V> execute(P payload);
}
```

Лістинг інтерфейсу задачі

2.4.1 Результат

У процесі виконання, кожна задача продукує *результат*. Результат за дизайном є монадою з функційних мов програмування. Цей клас допомагає представити результат операції, що може містити як значення, отримане в ході виконання задачі, так і помилку, що описує причину виникнення виняткової ситуації.

2.4.2 Монада

Монада – це концепція функціонального програмування, яка дозволяє структурувати послідовно виконані операції, керуючи потенційними помилками та побічними ефектами [6]. У Java класичний приклад монади - це `Optional`. Він дозволяє працювати зі значеннями, які можуть бути відсутніми, та легко обробляти цю ситуацію без використання `null`. Саме за прикладом бібліотечного класу `Optional` та типом `Result` з мови програмування `Rust`, було розроблено власну реалізацію.

Такий підхід допомагає вирішити недолік мови Java в обробці винятків. Створюючи програми на Java, програміст має два шляхи для обробки: checked та runtime exceptions. Перший варіант є непрактичним, до того ж він стає частиною сигнатури методу, де-факто змінюючи тип, який повертається методом. Другий, у свою чергу, може спричинити зупинку програми. Винятки часу виконання важко перехопити, оскільки ніде немає оголошення про можливе виникнення помилки.

Тому варіант представлення результату за допомогою власного класу допомагає локалізувати виняток у контексті операції та опрацьовувати там, де потрібно отримати значення операції, де цього вимагає логіка програми.
Ще однією перевагою є можливість абстрагуватись від класів, запроваджених у Java SDK, для роботи з результатами багапотокових обчислень, про що піде мова далі.

2.5 Асинхронна задача

При розробці програм, особливо у сучасних веб-додатках, виникає потреба у виконанні асинхронних операцій, що зазвичай відбуваються на віддалених серверах або в мережі. Для цього використовуються концепції реактивного програмування, де результати обчислень можуть бути отримані асинхронно та оброблені у реальному часі. В Java, одним з ключових інструментів для роботи з асинхронними операціями є клас CompletableFuture.

Цей клас є частиною пакету `java.util.concurrent` та є важливим компонентом для роботи з багатопотоковістю та асинхронним програмуванням в Java. `CompletableFuture` дозволяє виконувати обчислення асинхронно, уникаючи блокування потоку виконання. Він також надає можливості для комбінування, обробки та обробки результатів операцій.

2.5.1 CompletableFuture API

`CompletableFuture` дозволяє запускати обчислення асинхронно, тобто операції можуть виконуватись паралельно без блокування основного потоку виконання програми [7]. При ініціації асинхронної операції, об’єкт CompletableFuture повертається, що дозволяє отримувати результати обчислень у реальному часі, коли обчислення завершаються. Об’єкт, що повертається після запуску обчислень, надає широкий набір методів для комбінування, обробки та обробки результатів операцій, що дозволяє ефективно керувати асинхронними операціями.

Таким чином задача спростилась до створення класу, який є обгорткою для `CompletableFuture` (за принципом шаблону проєктування «Фасад») та синхронного очікування результату з `CompletableFuture`.

2.5.2 Шаблон проєктування «Фасад»

Фасад – це структурний патерн, що надає єдиний інтерфейс взаємодії до набору інтерфейсів у підсистемах. Цей шаблон дозволяє зменшити складність взаємодії зі складною підсистемою шляхом надання простого, більш високорівневого, інтерфейсу та ізолювати клієнтський код від деталей реалізації підсистеми. Це дозволяє зберігати код більш зрозумілим та підтримуваним [5, с. 185].

Даний патерн, спонукає програміста розділяти складний функціонал на підсистеми. Це покращує структуру проєктів, дозволяє повторно використовувати цілі модулі в майбутньому та моделювати бізнес-логіку в окремому контексті, а Фасад визначає спосіб комунікації з підсистемою.

Завдяки Фасаду, робота з асинхронністю у фреймворку звелась до передачі в задачу генератора `CompletableFuture`, а задача, при виконанні, запускає асинхронну операцію, очікує на результат. У випадку аварійного завершення роботи потоку, в якому відбувались обчислення, задача обгортає цю помилку в результат та повертає в процес.

2.6 Крок

2.6.1 Визначення та роль у системі

Крок, як проміжний шар абстракції, відіграє ключову роль у процесі оркестрації робочих процесів. Він інкапсулює в собі задачу та відповідає за передачу даних з загального контексту процесу до самої задачі.

У загальному сенсі, крок можна розглядати як один етап або ділянку робочого процесу, яка виконується певним чином із врахуванням вхідних даних та поточного стану. Кожен крок може бути найпростішою атомарною операцією або послідовністю складніших дій, які виконуються під час виконання процесу.

```
public abstract class WorkflowStep<S extends WorkflowState<S>>
implements WorkflowStepOrBuilder<S>, Task<S, S> {
    public static final String DEFAULT_NAME = "Unnamed";

    private final String name;

    protected WorkflowStep(String name) {
        this.name = Optional.ofNullable(name)
        .orElse(DEFAULT_NAME);
    }

    protected WorkflowStep() {
        this.name = DEFAULT_NAME;
    }

    @Override
    public WorkflowStep<S> toStep() {
        return this;
    }
}
```

Лістинг класу кроку

Сигнатура класу `WorkflowStep<S extends WorkflowState<S>>` визначає крок у системі оркестрації робочих процесів. Вона містить шаблонний параметр типу `<S>`, що пов'язує екземпляр кроку зі станом типу `S`, де `S` - це тип, який реалізує інтерфейс `WorkflowState<S>.`

Це визначення вийшло доволі природним, оскільки крок у своїй суті – це задача в контексті ланцюжка операцій. Єдиними відмінностями кроку від задачі полягають у попередній визначеності типу результату та призначенні. Крок як приймає, так і повертає стан. Він слугує адаптером (від однойменного шаблону проєктування «Адаптер») між задачею, яку визначив користувач системи та *процесом* виконання.

2.6.2 Шаблон проєктування «Адаптер»

Шаблон «Адаптер» є одним із структурних патернів проєктування, який дозволяє забезпечити взаємодію між двома інтерфейсами, які відрізняються або несумісні між собою. Основна ідея полягає в тому, щоб створити проміжний клас-адаптер, який конвертує інтерфейс одного класу у вигляд іншого, який очікує клієнт [5, с. 139].

Цей шаблон використовується тоді, коли потрібно взаємодіяти з існуючим класом, який має невідповідний інтерфейс з тим, що очікує клієнт або інша частина системи. Замість того, щоб змінювати код існуючого класу або створювати новий клас з таким самим інтерфейсом, шаблон «Адаптер» дозволяє створити проміжний клас, який перетворює запити від клієнта у формат, зрозумілий для цього класу.

Наприклад, якщо ми маємо клас, який працює з деякою бібліотекою або стороннім API зі своїм власним інтерфейсом, і нам потрібно використовувати цей клас у нашій системі, де очікується інший тип

інтерфейсу, ми можемо використати шаблон «Адаптер» для створення класу, який перетворює запити з одного інтерфейсу в інший. Це дозволяє нам забезпечити сумісність між цими двома інтерфейсами без необхідності внесення змін у вихідний код або змінювати поведінку існуючих класів.

Отже, адаптер допомагає вирішити проблему несумісності інтерфейсів шляхом створення проміжного адаптуючого класу, який дозволяє взаємодіяти з іншими класами або системами, не змінюючи їхнього вихідного коду.

Ця архітектура дозволяє розділити відповідальності між компонентами системи оркестрації робочих процесів та дозволяє забезпечити модульність та гнучкість в процесі визначення та виконання робочих процесів.

2.7 Стан процесу

Для будь-якої складної системи, що може містити багато понять, поглядів на ті самі речі, ми можемо визначити спільний обмежений контекст.

У контексті *Domain Driven Design* це концепція, яка визначає межі чітко визначеного домену, де терміни, правила та моделі мають специфічне значення і застосування. Обмежені контексти допомагають уникнути плутанини та конфліктів між моделями домену, які виникають внаслідок різних тлумачень та потреб бізнесу в різних частинах системи. Кожен обмежений контекст має свою власну мову, яка відображає термінологію, правила та поняття, що характерні для конкретної сфери діяльності.

Надихнувшись цією концепцією, було створено компонент, який допомагає пов’язати задачі та їх результати в одному місці. Цей компонент називається *стан*. У його межах зберігається інформація про набуті дані, метаінформація процесу, тощо. Він визначає межі відповідальності процесу та пов’язує кроки з відповідним процесом.

За задумом це має бути незмінний (*immutable*) об’єкт, для якого визначена операція переходу. Імутабельність є важливою концепцією в світі функціонального програмування. Вона допомагає уникнути багатьох проблем, пов'язаних зі змінюваними об'єктами, таких як недетермінований стан, зміна стану в непередбачуваних місцях та конфлікти при багатопотоковому доступі, оскільки вони гарантовано не змінюються.

На відміну від незмінних об’єктів, змінювані об'єкти можуть бути причиною виникнення невизначеної поведінки. Такі ситуації дуже складно визначити та відлагодити через недетермінованість стану. Імутабельні об'єкти полегшують відлагодження, оскільки їхній стан завжди залишається незмінним.

Операція переходу визначається для всіх екземплярів стану з допомогою операції копіювання. Вона є операцією вищого порядку, оскільки приймає параметром функцію, що вносить зміни до копії поточно стану.

```
public interface WorkflowState<S> {
   /** Returns a copy of the state. */
   S copy();

   /**
    * Returns a new state that is the result of applying the
reducer to the current state.
    */
   static <S extends WorkflowState<S>> S reduce(S state,
Consumer<S> reducer) {
       S copy = state.copy();
       reducer.accept(copy);
       return copy;
   }
}
```

Лістинг інтерфейсу стану

2.8 Процес

2.8.1 Визначення та структура процесів

У ієрархії системи найвищим рівнем є процес, який можна розглядати як автомат з обмеженою кількістю станів, відомий також як *Finite State Machine*. Основна функція процесу полягає в обході графу усіх кроків та виконанні задач, призначених кожному кроку. Кожен процес може мати свою власну імплементацію, що визначає спосіб виконання кроків.

Кожен процес визначається кроками та послідовністю їх виконання. Послідовність виконання кроків встановлюється з допомогою класів-будівельників (від однойменного шаблону проєктування «Будівельник») [5, с. 97].

Для спрощення конфігурації процесів, було створено підсистему абстракцій для створення об’єктів кроків та процесів. За дизайном вона нагадує спосіб творення повідомлень у Protobuf, де кожен згенерований клас містить відповідний клас-будівельник. Це дозволяє використовувати inline синтакс при створенні екземплярів вище перелічених класів, що сприяє читабельності коду.

2.8.2 Шаблон проєктування «Спостерігач»

Взаємодія з процесом реалізується за допомогою механізму спостерігачів (від однойменного патерну проєктування «Спостерігач»).

«Спостерігач» – це поведінковий патерн, що визначає залежність один-до-багатьох та дозволяє повідомляти залежні об’єкти про оновлення даних у об’єкти, за яким спостерігають [5, с .293].

Типовим сценарієм його використання є необхідність сепарувати операції побічної дії від об’єкту, який тримає у собі стан, тим самим

уникнути сильного зв’язування між класами, оскільки сильне зв’язування згубно впливає на здатність повторно використовувати класи, що невідворотно призводить до стрімкого погіршення якості кодової бази з часом.

Цей патерн описує яким чином повинен бути встановлений зв’язок між об’єктами, за яким контрактом вони будуть комунікувати. Таким чином, утворюються дві ролі: суб’єкт та спостерігачі. Суб’єкт може мати довільну кількість спостерігачів і сповіщати їх про оновлений стан. Коли стан оновлюється, всі спостерігачі по черзі реагують на зміни. Це породжує слабку зв’язаність, оскільки суб’єкт не знає хто є спостерігачем.

У системі оркестрації роль суб’єкту виконує процес. Саме він містить стан, який оновлюється кроками. Тому на кожне оновлення стану, сповіщаються спостерігачі, які можуть стрімити дані в режимі реального часу, писати логи і тому подібне.

```
public abstract class Workflow<S extends WorkflowState<S>>
extends WorkflowStep<S> {
   private final List<WorkflowObserver<S>> observers;
   protected Workflow(String name, List<WorkflowObserver<S>>
observers) {
       super(name);
       this.observers = observers;
   }

   protected Workflow(List<WorkflowObserver<S>> observers) {
       this.observers = observers;
   }

   protected final void notifyObservers(S state) {
       observers.forEach(observer ->
observer.observe(state.copy()));
   }
}
```

Лістинг класу процесу

2.8.3 Зв’язок процесу з кроком, вкладені процеси

Процес також є *кроком*, що дозволяє створювати нові процеси за допомогою композиції та сприяє повторному використанню коду. Це визначено сигнатурою абстрактного класу `abstract class Workflow<S extends WorkflowState<S>> extends WorkflowStep<S>`.

Ідея полягає в тому, що процеси можуть бути вкладеними. Вкладені процеси виконуються, як кроки, та керуються батьківським процесом. Кожен вкладений процес має власний *стан*, може мати своїх *спостерігачів*. Це дозволяє описувати весь функціонал системи, як незалежні компоненти і в подальшому комбінувати їх для досягнення необхідного результату.

2.8.4 Імплементації процесів

У рамках проекту було реалізовано три види процесів. Компонуючи та комбінуючи їх, можна виконати завдання будь-якої складності.

- Послідовний процес

Послідовний процес є базовою імплементацією абстрактного процесу. Він гарантує послідовність виконання кроків за принципом черги (First In First Out). Також цей тип процесу забезпечує синхронність виконання. Тобто наступний крок не почне виконуватись, доки не закінчиться попередній.

Імплементація базується на списках, де зберігаються всі кроки, які необхідно виконати. Коли викликається послідовний процес, починається ітерація по всіх доступних кроках та виконання кожного з них. Коли крок успішного завершується, стан процесу оновлюється, спостерігачі повідомляються про оновлення. А в разі виникнення помилки, процес завершується з помилкою, що повернулась від проваленого кроку.

- Паралельний процес

Паралельний процес також базується на списках. Його головною особливістю є паралельне виконання усіх кроків. При виконанні процесу, кожен крок запускається в окремому потоці з параметрами, отриманими з початкового стану.

У звичайному сценарії, процес очікує завершення всіх кроків. Коли крок успішно завершується, отриманий результат записується в стан процесу. Для того щоб не виникало race condition, доступ до стану процесу обмежений з допомогою `ReentrantLock`. Таким чином, стан блокується на читання та запис при кожному звертанні до нього. Це дозволяє гарантувати актуальність стану у будь-який момент часу. Паралельність виконання кроків, у свою чергу, реалізована за допомогою `CompletableFuture`, який використовує керований JVM `ForkJoinPool`. Цей механізм підходить для нетривалих обчислень та виділяє потоки з короткою тривалістю життя.

- Процес визначений графом

Цей вид процесу сильно відрізняться від обох попередніх. Він базується на ациклічному направленому графі. Такий граф моделює послідовності виконання кроків процесу, де кожен крок може мати залежності від інших, але не може привести до зациклення, коли процес повертається до вже відвіданої вершини.

У проекті було реалізовано власний клас графу. Імплементація базується на списку суміжності вершин, що має тип `Map<WorkflowStep<S>, List<WorkflowStep<S>>>`. У даній структурі визначаються самі вершини та зв’язки між ними. У класі реалізовано API для наповнення графу та представлення кроків у послідовному порядку з допомогою алгоритму

топологічного сортування. Ця особливість дозволила зробити систему ще гнучкішою, відкривши можливість визначати послідовні процеси з допомогою *DAG*.

За задумом, цей процес повинен мати високу часову ефективність у всіх сценаріях використання. Він виконує всі задачі, починаючи з вершини, що містить незалежну, першу, задачу. Рекурсивно спускаючись по цьому дереву, процес може оптимізувати виконання, паралельно виконуючи задачі, для яких було задоволено умови запуску.

Паралелізм досягається завдяки тому самому механізму, що й у паралельному процесі. У разі виникнення помилки чи непередбачуваної ситуації, виконання усіх піддерев кроків припиняється, процес повертає результат з описом винятку.

2.8.5 Подібність процесів

Оскільки кожен процес по своїй сутності є скінченним автоматом, кожен з них повинен мати початковий стан, який би конфігурував сценарій виконання. Ця можливість забезпечується інтерфейсом `WorkflowStep`, де стан є параметром необхідним для виконання кроку. Саме це значення ми розглядаємо як наш початковий стан.

Всі процеси мають забезпечувати консистентність стану виконання. Це означає, що ні в якому разі не повинно виникати стану гонитви, який би призводив до непередбачуваної поведінки чи помилкового результату.

РОЗДІЛ 3. ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ ФУНКЦІОНАЛУ

3.1 Тетстування фреймворку

Тестування є важливою складовою процесу розробки програмного забезпечення, тому чималу увагу було звернено на випробування написаного функціоналу. Завдяки вдалій архітектурі фреймворку, вдалось протестувати всі компоненти системи.

Для написання юніт-тестів використовувались наступні засоби:

3.1.2. Google Truth

Google Truth є бібліотекою для тестування в Java, яка забезпечує більш експресивний та зрозумілий синтаксис для написання тестів порівняння. Вона дозволяє легко виконувати перевірку різних умов та стверджень, роблячи тести більш зрозумілими та легкими для читання.

3.1.3 JUnit Jupiter

Jupiter є новітньою версією JUnit, основного фреймворку для тестування в Java. Він надає потужні можливості для написання та організації тестів, включаючи анотації для позначення тестових методів, можливість параметризації тестів та багато іншого. JUnit Jupiter дозволяє легко створювати розширені тестові набори (test suites) та виконувати різні види тестів.

3.1.4 Mockito

Mockito є бібліотекою для макетування об'єктів у тестах. Вона дозволяє створювати підроблені (mock) об'єкти для імітації реальних об'єктів у вашому коді, що допомагає ізолювати тести від залежностей та забезпечує більшу гнучкість та контроль при написанні тестів.

Загальне покриття тестами за рядками коду склало 75%.

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	94.1% (16/17)	78.6% (77/98)	75.2% (228/303)

Рисунок 1. Покриття вихідного коду юніт-тестами

3.2 Приклад застосування фреймворку

3.2.1 Визначення стану

Розглянемо примітивний приклад використання фреймворку. У даному прикладі виконуються задачі, що повертають дані про певну особистість, стан складається з двох полів: ім’я та вік.

```
static class Person implements WorkflowState<Person> {
    private String name;
    private int age;
    private Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public Person copy() {
        return new Person(name, age);
    }

    // Getters & Setters
}
```

Він реалізує метод інтерфейсу, який копіює об’єкт, це необхідна властивість стану.

3.2.2 Визначення кроків

Наступним етапом у створенні процесу є визначення кроків.

```
var firstStep = Step.<Person, String, Integer>
         forTask(name -> Result.ok(name.length()))
        .thatAccepts(Person::name)
        .reducesState(Person::setAge);
```

У даному фрагменті коду визначається крок, що відноситься до контексту `Person`, передає в задачу String та повертає Integer, виконує задачу з отримання нового віку персони. Ця задача записана у вигляді лямбда-

функції, що обраховує вік як довжину імені в символах. Після виконання задачі, кроки застосовує функцію, передану в метод `reducesState`, а саме встановлює нове значення віку.

Аналогічно повторюємо для інших необхідних кроків:

```
var secondStep = Step.<Person, Void, String>
         forTask(unused -> Result.ok(“Jane”))
        .thatAccepts(noPayload())
        .reducesState(Person::setName);
var thirdStep = Step.<Person, Person, Person>
         forTask(payload -> Result.ok(new Person("Agnis", 16)))
        .thatAccepts(workflowState())
        .reducesState((state, value) -> {
            state.setName(value.name());
            state.setAge(value.age());
        })
        .onSuccess(person -> System.out.println("Received a person!
" + person));
```

Особливість даного фрагменту полягає в наявності обробника успішного завершення задачі для третього кроку. Це ще один приклад використання шаблону Спостерігач, де крок є суб’єктом, що сповіщає про значення отримане від задачі. У випадку третього кроку, ми додаємо сторонній ефект до успішного сценарію виконання задачі, виводячи інформацію про новий стан у консоль.

3.2.3 Побудова процесу

Далі настає етап побудови процесу:

```
Workflow<Person> workflow = Workflows.<Person>builder()
        .addStep(firstStep)
        .addStep(thirdStep, dependsOn(firstStep, secondStep))
        .addStep(secondStep, dependsOn(firstStep))
        .build();
```

```
Person initialState = new Person("Amber", 34);
Person result = workflow.execute(initialState).unwrap();
// Result: Person{name='Agnis', age=16}
```

Творення процесу починається з ініціалізації об’єкта-будівельника з параметром класу стану. Після чого до образу процесу додаються кроки, всього є три способи це зробити, розглянемо кожен з них.

1. Явно додаємо крок, неявно вказуємо залежність

Цей варіант відображений у рядку `addStep(firstStep)`. На момент додавання кроку до графу, останній ще не містить вершин. Цю вершину неможливо пов’язати з іншим кроком, тому ми позначаємо цей крок, як первинний, тобто той що не містить залежностей.

У разі якщо граф вже містить вершини, крок, доданий через `addStep` без параметру, буде залежати від останнього доданого кроку. Наприклад:

```
Workflow<TestState> workflow = Workflows.<TestState>builder()
        .addStep(firstStep)
        .addStep(secondStep)
        .addStep(thirdStep)
```

У цьому прикладі відбувається наступне:

- Додається `firstStep`
- Додається `secondStep`
- Встановлюється залежнітсь між `secondStep` та `firstStep`
- Додається `thirdStep`
- Встановлюється залежність між `thirdStep` та `secondStep`

1. Явно додаємо крок, чітко вказуємо залежність

Даний приклад додавання кроку присутній у прикладі нижче.

```
Workflow<Person> workflow = Workflows.<Person>builder()
        .addStep(firstStep)
        .addStep(secondStep)
        .addStep(thirdStep, dependsOn(firstStep, secondStep))
```

У рядку `addStep(thirdStep, dependsOn(firstStep, secondStep))` явно додається третій крок і вказується його залежність від першого та другого кроків. Встановлення залежності відбувається з допомогою *модифікатора залежності*. За замовчуванням використовується модифікатор `dependsOnLastStep`, дію якого було продемонстровано раніше. У цьому ж випадку, модифікатор `dependsOn` потрібно передати всі кроки, від яких залежить перший аргумент.

2. Неявно додаємо крок

   Також наявний альтернативний варіант визначення залежностей, який частково реалізований з допомогою модифікатора `dependsOn`.

   Розглянемо приклад:

```
Workflow<Person> workflow = Workflows.<Person>builder()
        .addStep(thirdStep, dependsOn(firstStep, secondStep))
        .addStep(secondStep, dependsOn(firstStep))
```

   У цьому прикладі ми неявно додаємо другий крок у першому рядку. Це працює наступним чином:

   - Додається вершина `thirdStep`
   - До графу додаються вершини `firstStep`, `secondStep`
   - Встановлюється зв’язок між кроками для пар `firstStep`, `thridStep` та `secondStep`, `thirdStep`

   Процес, сконфігурований таким чином, за функціоналом відповідає двом попереднім.

Підсумовуючи, можна сказати, що всі три способи є взаємозамінними. Їх можна комбінувати за потреби, аби передати намір та задум у найбільш чіткий спосіб.

Створивши процес, далі необхідно ініціалізувати початковий стан, з яким він буде працювати. Після чого ми можемо запустити процес, отримавши у відповідь результат, коли процес завершить виконання.

ВИСНОВКИ

На закінчення, розробка системи оркестрування процесів розподілених систем – це комплексна та складна задача, що вимагає системного підходу. У процесі розробки були використані сучасні технології та випробувано багато підходів та технік програмування.

У ході роботи над курсовою роботою, було створено повноцінний фреймоворк, який задовольняє усі вимоги, поставлені на початку розробки цього програмного забезпечення: зручний API, вдала архітектура фреймворку, яка дозволяє тестувати клієнтський код підтримка асинхронних операцій.

Результатом роботи над проєктом став реліз бібліотеки на хмарний сервіс контролю версій GitHub з ліцензією MIT. Після чого бібліотека була випробувана в комерційному проєкті, де сприяла покращенню організації коду. Одним з наслідків впровадження впровадженням фреймворку став рефакторинг, який допоміг навести лад у коді та свого роду «задокументувати» процеси через декларативну природу цього проєкту.

Однією з головних переваг розробленого фреймворку є його простота в конфігурації та гнучкість у налаштуванні процесів. Використання принципів функціонального програмування та шаблонів проєктування дозволило створити ефективну та масштабовану архітектуру фреймворку, яка забезпечує стійкість та ефективність виконання завдань.

Отже, ця курсова робота пропонує новий погляд на процес оркестрації процесів та даних у розподілених системах та надає підґрунтя для подальших досліджень у цій області.

ДЖЕРЕЛА

1. Documentation. Apache Airflow. URL: https://airflow.apache.org/docs/

2. Documentation. Prefect. URL: https://docs.prefect.io/latest/.

3. Documentation. Dagster. URL: https://docs.dagster.io/getting-started.

4. Bloch J. Bumper-Sticker API Design. *InfoQ*.
URL: https://www.infoq.com/articles/API-Design-Joshua-Bloch/.

5. Design Patterns Elements of Reusable Object-Oriented Software / R. Helm et al. Pearson Education, 2000.

6. Function Programming. Loyola Marymount University.
URL: https://cs.lmu.edu/~ray/notes/functionalprogramming/.

7. CompletableFuture (Java Platform SE 8). Oracle Documentation.
URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html.