Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Кваліфікаційна робота

освітній ступінь - бакалавр

на тему: «**ЗАСТОСУВАННЯ ПАТЕРНІВ ПРОЕКТУВАННЯ ДЛЯ ВИРІШЕННЯ ЗАДАЧ ЛІНІЙНОЇ АЛГЕБРИ/THE USAGE OF DESIGN PATTERN FOR LINEAR ALGEBRA SOFTWARE DEVELOPMENT**»

Виконав: студент 4-го року навчання,

Освітньої програми «Прикладна математика», 113

Зверьок Богдан Олександрович

Керівник Бублик В.В.,_________

кандидат наук, доцент

Рецензент ________________________

Київ 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ» Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри математики, проф., д.ф.-м.н.

Р. К. Чорней

(підпис)

„____”__________________________2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Зверьку Богдану Олександровичу факультету інформатики 4-го курсу

ТЕМА Застосування патернів проектування для вирішення задач лінійної алгебри/The usage of design pattern for linear algebra software development

Зміст ТЧ до кваліфікаційної роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Розділ 1. Огляд концепції шаблонів проектування
6. Розділ 2. Огляд та застосування структурних патернів GOF
7. Розділ 3. Огляд та застосування породжувальних патернів GOF
8. Розділ 4. Огляд та застосування поведінкових патернів GOF
9. Висновки
10. Список використаних джерел

Дата видачі «___» __________ 2024 р. Керівник _______________________

(підпис)

Завдання отримав ________________________

(підпис)

Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання	Примітка
1.	Отримання завдання на курсову роботу.	23.10.2023	
2.	Огляд технічної літератури за темою роботи.	13.11.2023	
3.	Аналіз основних патернів проектування.	16.12.2023	
4.	Реалізація математичної складової роботи мовою програмування С++.	05.02.2024	
5.	Аналіз отриманих результатів з керівником.	25.03.2024	
6.	Покращення архітектури з урахуванням зауважень керівника.	19.04.2024	
7.	Створення презентації.	23.05.2024	
8.	Захист роботи	03.06.2024	

Студент Зверьок Б.О.

Керівник Бублик В.В.

«____» ________________

ЗМІСТ

Анотація......6
ВСТУП...... 7
1. ОГЛЯД КОНЦЕПЦІЇ ШАБЛОНІВ ПРОЕКТУВАННЯ...... 10
2. ОГЛЯД ТА ЗАСТОСУВАННЯ СТРУКТУРНИХ ПАТЕРНІВ GOF...12
2.1 Огляд концепції структурних шаблонів......12
2.2 Огляд та застосування патерну “Замісник”......12
2.3 Огляд та застосування патерну “Компонувальник”...... 15
2.4 Огляд та застосування патерну “Адаптер”......18
3. ОГЛЯД ТА ЗАСТОСУВАННЯ ПОРОДЖУВАЛЬНИХ ПАТЕРНІВ GOF......22
3.1 Огляд концепції породжувальних шаблонів...... 22
3.2 Огляд та застосування патерну “Абстрактна фабрика”...... 22
3.3 Огляд та застосування патерну “Одинак”...... 27
4. ОГЛЯД ТА ЗАСТОСУВАННЯ ПОВЕДІНКОВИХ ПАТЕРНІВ GOF......34
4.1 Огляд концепції поведінкових шаблонів...... 34
4.2 Огляд та застосування патерну “Шаблон”...... 34
4.3 Огляд та застосування патерну “Стратегія”......44
ВИСНОВКИ......56
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ...... 57

Перелік прийнятих скорочень

GoF - Gang of Four, група авторів книги "Design Patterns: Elements of Reusable Object-Oriented Software", яка визначила основні шаблони проектування програмного забезпечення;

СЛАР - система лінійних алгебраїчних рівнянь

Анотація

Дана робота присвячена дослідженню застосування патернів проектування для вирішення задач лінійної алгебри в програмному забезпеченні. У роботі розглядаються основні типи патернів проектування, включаючи структурні, породжувальні та поведінкові патерни, запропоновані у книзі "Design Patterns: Elements of Reusable Object-Oriented Software" авторів Еріха Ґамми, Річарда Гелма, Ральфа Джонсона та Джона Вліссідеса.

У першому розділі наведено загальний огляд концепції шаблонів проектування та їх важливість у розробці програмного забезпечення. Другий розділ присвячений структурним патернам, таким як "Замісник", "Компонувальник" та "Адаптер", з описом їх застосування в задачах лінійної алгебри, а саме для операцій над матрицями, векторами та СЛАР. Третій розділ охоплює породжувальні патерни, включаючи "Абстрактну фабрику" та "Одинака", з прикладами їх використання для створення чисельних об'єктів у програмних системах. Четвертий розділ зосереджений на поведінкових патернах, таких як "Шаблон" і "Стратегія", які забезпечують гнучкість та розширюваність алгоритмів для знаходження власних чисел/векторів і розв’язування СЛАР. Робота демонструє практичне застосування кожного патерну через приклади та аналізує їхню ефективність у контексті задач лінійної алгебри.

Ключові слова: лінійна алгебра, матриці, вектори, системи лінійних алгебраїчних рівнянь, патерни проектування, GoF, програмне забезпечення.

ВСТУП

У сучасному світі обчислювальна техніка та програмне забезпечення відіграють ключову роль у вирішенні різноманітних завдань, включаючи задачі лінійної алгебри. Лінійна алгебра, будучи фундаментальною галуззю математики, має широкі застосування в різних сферах. Її застосування розповсюджене в областях від інженерії та фізики до комп'ютерних наук та штучного інтелекту. Розуміння та оптимізація лінійно алгебричної обробки даних має критичне значення для розв'язання складних проблем.

Розвиток програмних продуктів, спрямованих на розв'язання задач лінійної алгебри, вимагає високої якості і ефективності, тому часто стикається з численними викликами. Навіть при наявності ефективних математичних алгоритмів реалізація їх у вигляді програм може бути складною через потребу управління складністю коду, підтримкою та розширенням функціональності. Таким чином, використання патернів проектування може допомогти вирішити ці проблеми і підвищити якість програмного забезпечення.

Метою даної роботи є показати і довести дієвість застосування різних патернів проектування для створення програмного забезпечення, спрямованого на вирішення задач лінійної алгебри, а також визначити найкращі практики для застосування такого підходу. Для досягнення поставленої мети необхідно вирішити наступне завдання - побудувати програмні моделі для демонстрації ефективності вищеописаного прийому розробки програмного забезпечення.

Для досягнення поставлених завдань буде проведено дослідження концепцій та особливостей кожного типу патернів проектування з метою визначення їх відповідності вимогам, будуть створені прикладні програмні

рішення з використанням обраних патернів проектування для демонстрації їх застосування та ефективності, а також буде надана оцінка переваг кожного розглянутого патерну з точки зору їх ефективності та придатності до вирішення завдань лінійної алгебри.

Дослідження базується на використанні мови програмування C++, адже вона є однією з найпоширеніших мов програмування у сфері розробки програмного забезпечення, особливо в області високопродуктивного обчислення, а також вирішення задач наукового значення. Вона надає розширені можливості для реалізації патернів проектування, включаючи шаблони, управління пам'яттю та низькорівневий доступ до апаратного забезпечення.

Методологічна основа даної роботи базується на синтезі теорії патернів проектування та практичної реалізації їх у середовищі мови програмування C++.

Ця робота має на меті висвітлити важливість та можливості використання патернів проектування у розробці програмного забезпечення для лінійної алгебри та сприяти розумінню їх застосування в практичних завданнях. Вирішення поставлених завдань дозволить зрозуміти, як патерни проектування можуть бути використані для покращення якості та ефективності програмного забезпечення, спрямованого на вирішення задач лінійної алгебри.

Робота складається з трьох основних розділів. Перший розділ має на меті огляд та застосування структурних шаблонів. У цьому розділі надається огляд концепції структурних патернів проектування визначених "бандою чотирьох", таких як "Адаптер", "Замісник" та "Компонувальник". У другому розділі йдеться про застосування породжуючих шаблонів, проводиться огляд концепції породжуючих патернів проектування GoF, таких як "Абстрактна фабрика" та "Одинак". Третій розділ містить

застосування поведінкових шаблонів GoF, розглядається концепція поведінкових патернів проектування GoF, зокрема "Стратегія", та "Шаблон".

Кожен з розділів роботи має на меті не лише представити теоретичні аспекти патернів проектування, але й показати їх практичне застосування у вирішенні конкретних завдань у програмному забезпеченні для лінійної алгебри на мові програмування C++. У кожному розділі подано пояснення кожного шаблону та його застосування у вирішенні конкретних проблем у контексті лінійної алгебри, а також представлені приклади реалізації кожного з цих шаблонів.

1. ОГЛЯД КОНЦЕПЦІЇ ШАБЛОНІВ ПРОЕКТУВАННЯ

Патерн проектування представляє собою типовий метод розв'язання певної проблеми, що часто виникає під час розробки програмного забезпечення. Він є загальним принципом вирішення проблеми, який можна адаптувати та використовувати для різних програм. У відміну від готових функцій чи бібліотек, патерн не є конкретним кодом, але надає структурований підхід до розв'язання певної проблеми, що легко підлаштовується під різні ситуації і реалізації.

Аналогія з кулінарним рецептом та інженерним кресленням може бути корисною для розуміння патернів проектування. Якщо алгоритм подібний до кулінарного рецепту з чіткими кроками, то патерн — це інженерне креслення, яке надає високорівневий опис рішення, не вказуючи конкретних дій для його отримання.

Опис патернів проектування зазвичай містить наступні елементи: Проблема: Опис проблеми або ситуації, до якої застосовується патерн; Мотивація: Пояснення того, чому використання патерна є вигідним для вирішення даної проблеми; Структура класів: Опис структури та взаємозв'язку класів, які складають розв'язок патерна; Приклад: Наведення прикладу реалізації патерна на одній з мов програмування; Особливості реалізації: Вказівки щодо того, як можна адаптувати реалізацію патерна для різних контекстів та вимог; Зв'язки з іншими патернами: Опис взаємозв'язку даного патерна з іншими, що може бути корисним для розуміння та використання. Цей формат дозволяє стандартизувати та узагальнити знання про патерни проектування, що сприяє їх широкому застосуванню та розумінню.

Патерни проектування можна класифікувати за різними критеріями, такими як призначення (структурні, поведінкові, породжувальні), рівень

абстракції (класові, об'єктні) або складність впровадження. У цій роботі вони будуть розглядатися в першу чергу за критерієм призначення.

Крім позитивних прикладів патернів, важливо також розуміти таке явище як антипатерни - типові помилки чи недоліки у дизайні програм, що призводять до збільшення складності системи та зниження продуктивності розробки і яких можна уникнути або виправити за допомогою патернів. Вони можуть виникнути, наприклад, якщо розробники не розуміють повністю поставлену перед ними задачу або не аналізують достатньо уважно всі можливі варіанти рішення. Також варто згадати про таке явище як “Золотий молоток (Golden Hammer)”, суть якого полягає у тому, що розробники використовують один і той самий підхід або інструмент для вирішення всіх проблем, незалежно від того, чи є він найкращим варіантом у даному контексті, або “Cargo Cult Programming”, де діло у бездумному копіюванні коду або практик з інших проектів або бібліотек, навіть якщо розробник не розуміє їхньої суті або необхідності [1].

Якщо ми говоримо про якесь конкретне застосування таких прийомів, доречним буде продемонструвати це на прикладанні до лінійної алгебри. Це може бути корисним для створення ефективних та легко розширювальних програм, які працюють з математичними структурами та операціями. Лінійна алгебра включає в себе широкий спектр операцій, що буде доречним структурувати та розподілити їх між різними класами або компонентами програми. В даній галузі часто оперують різними типами даних, такими як цілі числа, дійсні числа або комплексні числа, тому вкрай необхідно створити рішення для розширення системи для підтримки нових типів даних без зміни попередньо зробленої роботи. Варто також згадати про патерни оптимізації, що прекрасно вирішують проблеми з навантаженням на систему, особливо у випадках, коли масштабність даних є проблемою.

2. ОГЛЯД ТА ЗАСТОСУВАННЯ СТРУКТУРНИХ ПАТЕРНІВ GOF

2.1 Огляд концепції структурних шаблонів

Структурні патерни є однією з категорій патернів проектування і використовуються для створення більш гнучких та ефективних програм шляхом композиції об'єктів та класів. Основна мета структурних патернів полягає в тому, щоб створити стійку до змін структуру програми, забезпечуючи високий рівень абстракції та розподілуючи обов'язки між різними частинами системи [2]. Шаблони цього виду допомагають розділити відповідальності між різними класами та об'єктами програми загалом зменшуючи зайву складність системи. Це дозволяє забезпечити простоту та зрозумілість коду, а також зменшує залежність між різними частинами системи. Багато структурних патернів базуються на ідеї композиції та агрегації об'єктів, що дозволяє створювати складні структури, які складаються з простіших елементів, та забезпечує гнучкість та легкість розширення програми. Вони також допомагають дотримуватися принципу окремості, розділяючи різні аспекти функціональності на окремі класи та компоненти, що звісно дає лише переваги.

2.2 Огляд та застосування патерну “Замісник”

Патерн "Замісник" - це структурний патерн проектування, який дозволяє створювати об'єкт-посередник для свого батьківського класу, який матиме змогу контролювати доступ до нього. Основна ідея полягає в тому, щоб створити об'єкт, який діє як замісник для іншого об'єкта, контролює доступ до нього та виконує додаткову функціональність до або після виконання операцій над ним [3].

Замісник використовується коли контролювати доступ до об'єкта (наприклад, перевірка прав доступу), коли потрібно виконати додаткові

операції до або після виконання основних операцій над об'єктом (наприклад, логування, кешування, відкладене завантаження даних тощо) або коли потрібно оптимізувати роботу з реальним об'єктом (наприклад, виконувати операції тільки при реальній необхідності) [4].

Використання вищеописаного прийому буде продемонстроване на операціях з матрицями. Нехай буде створений загальний абстрактний клас матриць (Matrix) і дочірній клас з конкретною реалізацією (ConcreteMatrix). Для агрегації різних матричних операцій в один результат буде створена клас MatrixManager і тоді для застосування патерну буде створено новий клас MatrixManager Proxy, що буде дочірнім до MatrixManager. При його створенні необхідно буде передати в конструктор екземпляр класу дочірнього до MatrixManager. Таким чином кожен екземпляр класу MatrixManagerProxy буде містити екземпляр іншого класу менеджера матриць, над чиїми операціями будуть проводитися маніпуляції. В даному випадку буде розглянуто знаходження і кешування результату оберненої матриці за допомогою матриці мінорів та матриці алгебраїчних доповнень.

Алгоритм дій наступний: Спочатку знаходиться визначник вихідної матриці і робиться перевірка, чи може взагалі ця матриця бути оберненою. Для цього її визначник не має дорівнювати нулю. Знаходиться матриця мінорів, де для кожного елемента a_{ij} вихідної матриці A, його мінор M_{ij} обчислюється як детермінант підматриці, яка утворюється видаленням рядка i та стовпця j з матриці A, тобто $M_{ij} = det(A_{ij})$. Після цього знаходиться матриця алгебраїчних доповнень, де для кожного елемента M_{ij} матриці мінорів, його алгебраїчне доповнення C_{ij} обчислюється як $(-1)^{i+j}$ помножене на відповідний мінор M_{ij}: $C_{ij} = (-1)^{i+j} * M_{ij}$. Далі

знаходиться транспонована матриця алгебраїчних доповнень і ділиться на визначник вихідної матриці: $A^{-1} = C_{ij}^{T} / det(A)$.

Таким чином, щоб не виконувати всі вищеописані оперції щоразу коли програмі потрібно знайти обернену матрицю до даної, клас замісник буде кешувати результат цієї дії. У класі MatrixManagerProxy буде перевизначено відповідний метод з додаванням додаткової логіки. Коли метод для знаходження викликається вперше, обернена матриця буде обрахована і збережена в полях класу, але в одному з полів також буде позначено, що результат уже був обрахований і наступного разу його ще раз обраховувати не потрібно. Таким чином, при послідуючих викликах цього методу буде повертатися вже готове значення, що було збережене раніше. Нижче можна побачити кодове представлення:

```
class MatrixManager {
    public:
      virtual Matrix *getInverseMatrix(const Matrix &matrix) const {

        if (!matrix.isMatrixInvertible()) {
          return nullptr;
        }

        vector<vector<int>> minors = matrix.calculateMatrixOfMinors();
        Matrix *inverseMatrix = matrix.calculateInverse(minors);
        return inverseMatrix;
      }
    };

class MatrixManagerProxy : public MatrixManager {
```

```
private:
  mutable Matrix *inverseMatrix;
public:
  mutable bool isInverseCached;

MatrixManagerProxy() : inverseMatrix(nullptr), isInverseCached(false)
{}

  Matrix *getInverseMatrix(const Matrix &matrix) const override {
    if (!isInverseCached) {
      inverseMatrix = getInverseMatrix(matrix);
      isInverseCached = true;
    }
    return inverseMatrix;
  }

  ~MatrixManagerProxy() {
    delete inverseMatrix;
  }
};
```

2.3 Огляд та застосування патерну “Компонувальник”

Патерн "Компонувальник" - це структурний патерн проектування, який дозволяє створювати ієрархічну структуру об’єктів, за допомогою їх об'єднання в деревоподібну структуру даних. Цей патерн дозволяє клієнтам обробляти окремі об'єкти та їхні групи однаковим чином. Перевага цього підходу в тому, що клієнтам не потрібно знати, чи вони мають справу з листком чи контейнером. Вони можуть взаємодіяти з усіма

елементами в ієрархії однаково, викликаючи спільні методи. Також він дозволяє створювати деревоподібні структури, що можуть бути рекурсивними, тобто контейнери можуть містити інші контейнери, створюючи складні ієрархії, з його допомогою можна легко обробляти складні структури, додавати нові елементи та видаляти існуючі, не змінюючи клієнтський код [5].

Даний патерн буде продемонстровано на прикладі теми векторів. Нехай існує певний клас векторів Vector з базовим набором методів. Тоді буде створено клас-компонувальник CompositeVector, що унаслідує попередній клас і перевизначить всі його методи. Цей клас міститиме в полях масив векторів і інтерфейс для додавання елементів туди. Тоді коли будь-яка перевизначена операція з базового класу вектора при виклику буде застосовуватися до всіх векторів в масиві.

З допомогою цього класу можна гнучко складати складні комбінації векторів і працювати з ними з таким же інтерфейсом, який би був для одного вектора. Такий підхід дуже корисний для обробки, наприклад, складних графічних об’єктів, для моделювання складних 3D форм або для чисельних обчислень, де дані представлені у вигляді векторів. Кодове представлення можна побачити нижче:

```
class CompositeVector : public Vector {
    private:
      std::vector<Vector> vectors;

    public:

      void addVector(const Vector &vec) {
        vectors.push_back(vec);
```

```
    }

    void add(const double el) override {
        for (Vector &vec: vectors) {
            vec.add(el);
        }
    }

    void add(const Vector &other) override {
        for (Vector &vec: vectors) {
            vec.add(other);
        }
    }

    void multiply(double scalar) override {
        for (Vector &vec: vectors) {
            vec.multiply(scalar);
        }
    }

    void print() const override {
        for (const Vector &vec: vectors) {
            vec.print();
        }
    }
};
```

2.4 Огляд та застосування патерну “Адаптер”

Патерн "Адаптер" - це структурний патерн проектування, який дозволяє об'єднати два несумісні між собою інтерфейси і зробити їх взаємозамінними. Головна мета патерна адаптера - забезпечити спільну точку взаємодії між класами, які інакше не могли би працювати разом через різницю в їхніх інтерфейсах. Існує два типи адаптерів: класовий, який використовує множинне успадкування для адаптування інтерфейсів, і об’єктний адаптер, що використовує композицію, де адаптер містить екземпляр адаптованого класу. Другий тип дозволяє більшу гнучкість, оскільки він не потребує наслідування, але вимагає додаткового коду для делегування викликів. В цілому суть підходу у легкій і ефективній інтеграції різних частин програми або бібліотеки, забезпечуючи максимальну гнучкість та розширюваність програмного забезпечення [6].

Цей патерн буде продемонстровано на прикладі взаємодії матриць з векторами, чиї інтерфейси є несумісними. Буде використано об’єктний адаптер, клас унаслідує клас вектора і в конструкторі при створенні об’єкта буде приймати екземпляр класу матриці. Кожен метод батьківського класу буде перевизначеним, де логіка в них буде адаптована таким чином, щоб працювати з один стовпцем матриці, що була передана при створенні. Кодове представлення:

```
class MatrixAdapter : public LinearVector {
    private:
      Matrix &matrix;

    public:
      MatrixAdapter(Matrix &_matrix) : matrix(_matrix) {
         if (matrix.numCols() != 1) {
```

```
                throw std::invalid_argument("MatrixAdapter can only adapt matrices with one column.");
            }
        }

        virtual size_t size() const override {
            return matrix.numCols();
        }

        virtual void add(const LinearVector &other) override {
            if (size() != other.size()) {
                throw std::invalid_argument("Vectors must be of the same size for addition.");
            }
            Vector result;
            for (size_t i = 0; i < size(); ++i) {
                matrix.setValue(i, 0, matrix.getValue(i, 0) + other.getValue(i));
            }
        }

        virtual void multiply(double scalar) override {
            for (size_t i = 0; i < size(); ++i) {
                matrix.setValue(i, 0, matrix.getValue(i, 0) * scalar);
            }
        }

        virtual void print() const override {
            std::cout << "[";
```

```
        for (size_t i = 0; i < size(); ++i) {
            std::cout << matrix.getValue(i, 0);
            if (i != size() - 1) {
                std::cout << ", ";
            }
        }
        std::cout << "]" << std::endl;
    }
};
```

Даний адаптер перевизначає всі методи базового класу і змінює логіку відповідно до об’єкта, що передається йому в конструкторі. Інший спосіб використання цього підходу полягає в перетворенні вхідних даних на елементи створення базового класу.

Нехай суть задачі у створенні обчислювальної платформи, де, наприклад, потрібно розв’язати систему лінійних рівнянь, котру користувач буде задавати текстом. Таким чином нам потрібно якось підлаштувати цю стрічку під відповідний об’єкт для забезпечення зручного інтерфейсу проведення різних операцій. В такому випадку буде створено клас-адаптер, що унаслідує клас, який репрезентує систему лінійних рівнянь, і буде конвертувати стрічку до матриць коефіцієнтів і сталих, що необхідні для створення базового класу. Таким чином у класі-адаптері вся реалізація залишиться в базовому класі, окрім метода, що буде безпосередньо конвертувати стрічку в матриці і результат якого буде переданий в конструктор базового класу. Демонстрацію відповідного класу можна побачити нижче:

```
class LinearEquationSystem {
```

```
protected:
  std::vector<std::vector<double>> coefficientMatrix_;
  std::vector<std::vector<double>> constantMatrix_;

public:
  LinearEquationSystem(
                  const std::pair<std::vector<std::vector<double>>, std::vector<std::vector<double>>> coefficientConstantsPair)
                  : coefficientMatrix_(coefficientConstantsPair.first), constantMatrix_(coefficientConstantsPair.second) {}
};

class EquationParserAdapter : public LinearEquationSystem {
public:
  EquationParserAdapter(const std::string &equationsString)
  : LinearEquationSystem(parseEquations(equationsString)) {}

private:
                static      std::pair<std::vector<std::vector<double>>, std::vector<std::vector<double>>>
  parseEquations(const std::string &equationsString) {
  // Парсинг стрічки до пари матриць
  }
};
```

3. ОГЛЯД ТА ЗАСТОСУВАННЯ ПОРОДЖУВАЛЬНИХ ПАТЕРНІВ GOF

3.1 Огляд концепції породжувальних шаблонів

Породжувальні патерни проектування - це клас патернів, які вирішують проблеми, пов'язані з процесом створення об'єктів. Вони забезпечують механізми для створення об'єктів таким чином, щоб клієнти не повинні бути прив'язані до конкретних класів об'єктів, які вони створюють. Це сприяє зменшенню залежності між клієнтами та створюваними об'єктами, що робить систему більш гнучкою та легше змінюваною. Породжувальні патерни дозволяють створювати об'єкти з максимальною гнучкістю та незалежністю від конкретних класів, що сприяє полегшенню розширення та підтримки систем. Вони є важливою складовою частиною процесу проектування програмного забезпечення, дозволяючи створювати добре структуровані та легко змінювані системи. Основні ідеї за породжувальними патернами включають інкапсуляцію знань про конкретні класи та приховання деталей створення та комбінування об'єктів [7].

3.2 Огляд та застосування патерну “Абстрактна фабрика”

Абстрактна фабрика - це породжувальний патерн проектування, який дозволяє створювати різні сімейства пов'язаних між собою об'єктів без прив'язки до конкретних класів цих об'єктів. Він надає інтерфейс для створення сімейства об'єктів різних типів, причому ці об'єкти повинні взаємодіяти між собою. У цього підходу є багато переваг, бо гарантується сумісність створених об'єктів, оскільки всі вони створюються в межах однієї фабрики, а також він дозволяє легко змінювати сімейство створених об'єктів, змінюючи лише фабрику. Проте є й недоліки, як от те, що при

додаванні нових продуктів або фабрик потрібно внести зміни до всіх існуючих фабрик, що може призвести до значних модифікацій у коді. Також при додаванні нових сімейств об'єктів потрібно розширити інтерфейс абстрактної фабрики та всіх конкретних фабрик.

Даний патерн можна прекрасно застосувати для породження екземплярів класів чисельних об’єктів. Нехай існує будь-який клас матриці та вектора. Кожен з них мусить мати певне поле, що зберігатиме чисельні значення. Тут постає питання, якого саме типу має бути це поле, адже існує безліч різних підходів, що мають переваги і недоліки в залежності від конкретної ситуації. Можна згадати два приклади, такі як vector зі стандартного простору імен C++ std і звичайний масив.

Використання векторів для реалізації класу матриць має свої переваги. Вектори є динамічними масивами та забезпечують автоматичне управління пам'яттю, що дозволяє уникнути проблем, пов'язаних з витіканням пам'яті або некоректною індексацією. Вони також забезпечують безпеку виконання та зручний інтерфейс для додавання, видалення та отримання елементів, спрощуючи процес роботи з ними. Однак, використання векторів може мати свої недоліки. Наприклад, вони можуть займати більше місця в пам'яті та мати невеликий наклад на продуктивність через додаткові операції копіювання та розширення. Також вектори не забезпечують такого швидкого доступу до елементів як звичайні масиви, що може вплинути на продуктивність додатка в цілому.

Використання звичайних масивів для реалізації класу матриць також має свої переваги. Звичайні масиви можуть бути більш ефективними за використання пам'яті та швидшими для доступу до елементів, оскільки вони не мають додаткових накладних витрат. Це особливо важливо, якщо ви працюєте з великими обсягами даних або якщо потрібна оптимізація продуктивності. Проте, використання звичайних масивів також може мати

свої недоліки. Наприклад, вони не забезпечують автоматичного управління пам'ятю, що може призвести до проблем, таких як витік пам'яті або сегментація. Розмір звичайного масиву повинен бути відомим на етапі компіляції, що може ускладнити роботу з динамічно змінюваними розмірами матриць та зробити їх менш гнучкими у використанні.

Підсумовуючи, можна сказати, що вектори дозволяють динамічно змінювати розмір матриці та мають зручний інтерфейс, але можуть бути менш ефективними за використання пам'яті. Звичайні масиви ефективніші за використання пам'яті та швидші для доступу до елементів, але вимагають відомого розміру матриці на етапі компіляції, тому підходи обираються відповідно до вимог конкретної системи чи її модуля.

У випадку якщо система потребує обох цих підходів для різних операцій чи використання у різних її модулях, доцільно буде застосувати вищеописаний патерн, а саме “Абстрактна фабрика”.

Буде створено загальний інтерфейс абстрактних фабрик AbstractAlgebraFactory, від якого наслідуються два дочірні класи, що будуть відповідати за один з двох вищеописаних підходів зберігання даних, а саме ArrayFactory та StdFactory. Кожен клас буде мати методи createMatrix та createVector, і буде повертати екземпляр класу відповідно до своїх вимог, іншими словами ArrayFactory буде породжувати об’єкти матриць та векторів, в основі яких структурою даних служить звичайний двовимірний масив, водночас як StdFactory повертатиме екземпляри класів, в основі яких буде vector з простору імен std. Оскільки при створенні об’єкту потрібно буде передати набір даних заздалегідь у якомусь визначеному форматі, нехай це буде двовимірний масив, тому у фабриках, породжувальні об’єкти яких працюють з іншими структурами даних, потрібно буде зробити перетворення даних до іншого формату.

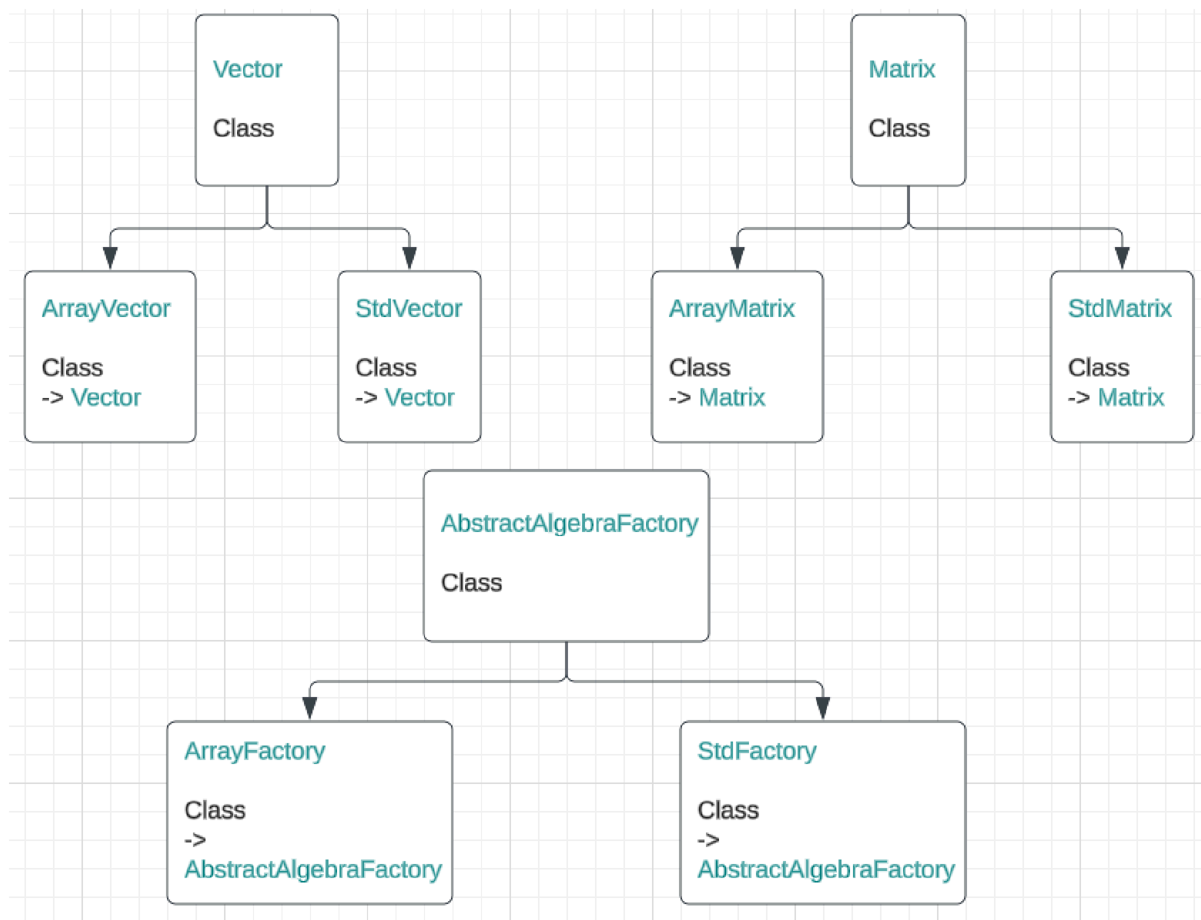

рис. 3.1 Діаграма класів використаних для патерну “Абстрактна фабрика”

Кодове представлення вищеописаного методу:

```
class AbstractAlgebraFactory {
        public:
          virtual Matrix createMatrix(int **values) = 0;

          virtual Vector createVector(int *values) = 0;

          virtual ~AbstractAlgebraFactory() {}
        };

        class ArrayFactory : public AbstractAlgebraFactory {
```

```
    public:
      Matrix createMatrix(int **values) override {
        return ArrayMatrix(values);
      }

      Vector createVector(int *values) override {
        return ArrayVector(values);
      }
    };

class StdFactory : public AbstractAlgebraFactory {
    private:
      std::vector<std::vector<int>> convertToIntVector(int **values) {
        std::vector<std::vector<int>> result;
        int rows = sizeof(values) / sizeof(values[0]);
        int cols = sizeof(values[0]) / sizeof(values[0][0]);

        for (int i = 0; i < rows; ++i) {
          std::vector<int> row;
          for (int j = 0; j < cols; ++j) {
            row.push_back(values[i][j]);
          }
          result.push_back(row);
        }
        return result;
      }

      std::vector<int> convertToIntVector(int *values) {
```

```
        std::vector<int> result;
        for (int i = 0; i < sizeof(values); ++i) {
            result.push_back(values[i]);
        }
        return result;
    }

public:

    Matrix createMatrix(int **values) override {
        return StdMatrix(convertToIntVector(values));
    }

    Vector createVector(int *values) override {
        return StdVector(convertToIntVector(values));
    }
};
```

3.3 Огляд та застосування патерну “Одинак”

Патерн "Одинак" - це породжувальний патерн проектування, суть якого у забезпеченні того, що клас, який реалізує цей підхід, матиме лише один екземпляр та забезпечуватиме глобальну точку доступу до цього екземпляра. Це корисно тоді, коли потрібно мати доступ до єдиного об'єкта для обробки деяких загальних ресурсів або функціональностей, наприклад, об'єкту, що забезпечує доступ до бази даних, сервісу журналізації, тощо. Або ж такий метод можна використати у випадку, коли клас є по суті статичним і не зберігає ніякого стану, проте існують подібні йому класи, що реалізують такий же інтерфейс як і він. Тому в таких

випадках теж застосовується цей підхід для того, щоб мати змогу використовувати ці класи під егідою одного інтерфейсу і при цьому не завантажувати оперативну пам'ять створенням багатьох об'єктів [8]. Однак цей патерн порушує принцип єдиної відповідальності (Single Responsibility Principle), оскільки клас відповідає не тільки за свої основні функції, але й за створення свого єдиного екземпляру, а також може призводити до складності у тестуванні, оскільки використання глобальних змінних може ускладнити ізоляцію класів для тестування.

Даний підхід буде продемонстровано на прикладі розв'язування системи лінійних рівнянь за допомогою множення матриць різними способами, а саме методом прямого множення та методом Штрассена. Щоб розв'язати систему лінійних рівнянь за допомогою множення матриць, спершу потрібно представити систему у вигляді матричного рівняння AX=B, де A - матриця коефіцієнтів системи, X - матриця невідомих змінних, а B - матриця вільних членів. Обчислюється обернена матриця A^{-1}, та множиться на матрицю вільних членів B, в результаті чого знаходиться матриця невідомих змінних X. Але варто зазначити, що система має розв'язок, коли A є невиродженою, тобто її визначник не дорівнює нулю.

Як вже було описано раніше, даний приклад буде показано на прикладі двох методів множення матриць. Перший метод прямого множення є найпоширенішим. Основна ідея полягає в тому, щоб для кожного елемента результуючої матриці обчислити суму добутків відповідних елементів відповідних рядків першої матриці та стовпців другої матриці. Для кожного елементу C_{ij} результуючої матриці, де i - номер рядка, а j - номер стовпця, буде обчислено суму добутків відповідних елементів A_{ik} та B_{kj} для всіх k від 1 до кількості стовпців (або

рядків) у відповідних матрицях. Цей алгоритм має часову складність $O(n^3)$, де n - розмір матриць. Це означає, що час виконання зростає кубічно з розміром матриць, що робить його неефективним для великих матриць.

Водночас метод Штрассена є методом, що дозволяє зменшити кількість операцій порівняно з класичним алгоритмом множення матриць, особливо для великих матриць. Цей метод ґрунтується на ідеї розбиття матриць на менші підматриці, обчисленням окремих частин та комбінуванням результатів для отримання кінцевого продукту. Кожна вихідна матриця розбивається на підматриці рівного розміру, обчислюються проміжні матриці, які складаються з підсумків деяких лінійних комбінацій підматриць вихідних матриць, потім за допомогою проміжних матриць обчислюються кінцеві підматриці, що потім комбінуються для отримання результату. Основна ідея полягає в тому, що у методі Штрассена кількість операцій зменшується до $O(n^{log_2 7})$, що призводить до значного зменшення обчислювальних витрат порівняно з класичним методом множення матриць, особливо для великих матриць. Однак втім, метод Штрассена вимагає більше пам'яті та обчислювальних ресурсів для зберігання і обчислення проміжних матриць, тому в деяких випадках він може бути менш ефективним [9].

Програмно весь вищеописаний алгоритм буде реалізовано наступним чином. Буде створено загальний інтерфейс MatrixMultiplier, який реалізує два класи, що відповідатимуть за відповідні методи множення матриці: MatrixMultiplierSimple та MatrixMultiplierStrassen. Далі буде створено функцію, що прийматиме вже обернену матрицю A та матрицю B, результатом якої буде розв’язок системи лінійних рівнянь. Всередині даної функції буде викликано метод, що проаналізує матрицю A і повертатиме відповідний метод множення матриць. У випадку якщо A буде мати розмір більший за, нехай, 10000, повертатиметься метод

Штрассена, інакше класичний метод. Таким чином ресурси системи будуть витрачатися ефективно при проведенні даної операції.

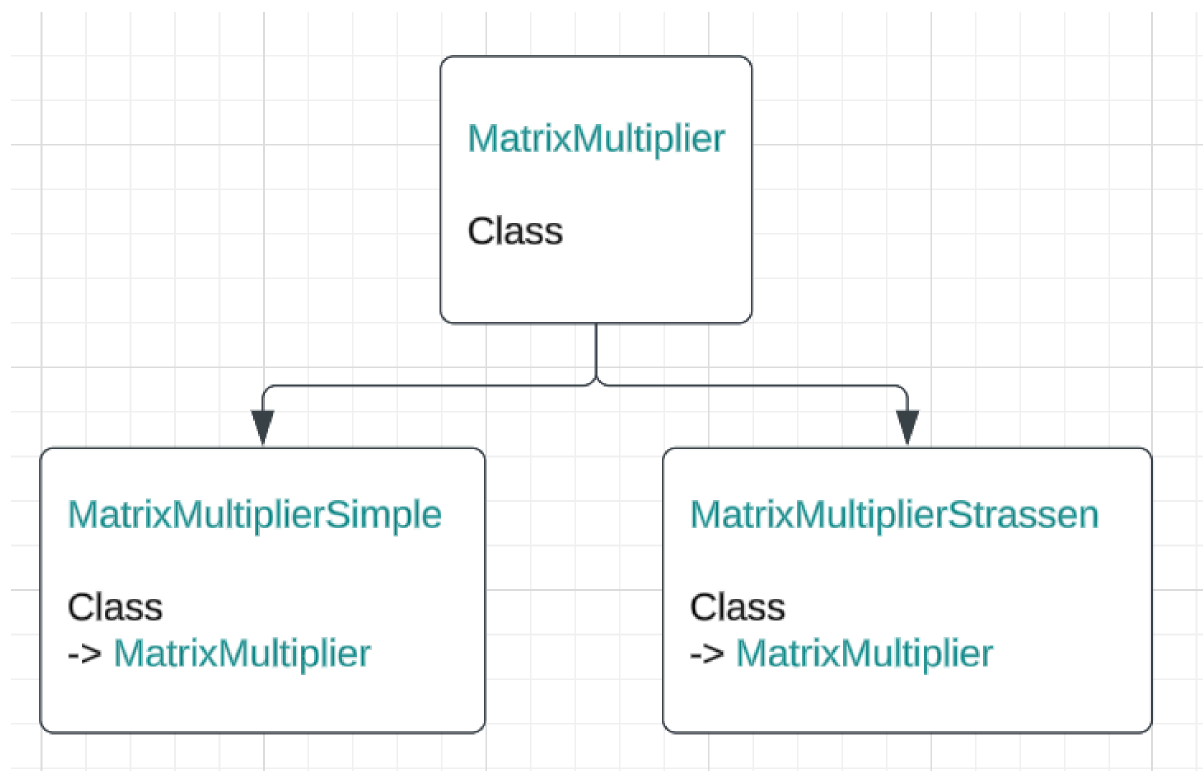

рис. 3.2 Діаграма класів використаних для патерну “Одинак”

Кодове представлення матиме наступний вигляд:

```
class MatrixMultiplier {
        public:
          virtual std::vector<std::vector<int>>
                multiply(const std::vector<std::vector<int>> &matrix1, const std::vector<std::vector<int>> &matrix2) = 0;
        };

class MatrixMultiplierSimple : public MatrixMultiplier {
        private:
```

```
    MatrixMultiplierSimple() {}
  public:
    static MatrixMultiplierSimple* &getInstance() {
      static MatrixMultiplierSimple* instance;
      return instance;
    }

    std::vector<std::vector<int>>
        multiply(const std::vector<std::vector<int>> &matrix1, const
std::vector<std::vector<int>> &matrix2) override {
                      std::vector<std::vector<int>> result(matrix1.size(),
std::vector<int>(matrix2[0].size(), 0));
      for (size_t i = 0; i < matrix1.size(); ++i) {
        for (size_t j = 0; j < matrix2[0].size(); ++j) {
          for (size_t k = 0; k < matrix1[0].size(); ++k) {
            result[i][j] += matrix1[i][k] * matrix2[k][j];
          }
        }
      }
      return result;
    }
  };

class MatrixMultiplierStrassen : public MatrixMultiplier {
  private:
    MatrixMultiplierStrassen() {}

  public:
```

```
        static MatrixMultiplierStrassen* &getInstance() {
            static MatrixMultiplierStrassen* instance;
            return instance;
        }

        std::vector<std::vector<int>>
            multiply(const std::vector<std::vector<int>> &matrix1, const
std::vector<std::vector<int>> &matrix2) override {
            // Реалізація множення матриць методом Штрассена
        }
    };

    std::vector<std::vector<int>>
    multiplyMatrices(const std::vector<std::vector<int>> &matrix1, const
std::vector<std::vector<int>> &matrix2,
                    MatrixMultiplier *matrixMultiplier) {
        return matrixMultiplier->multiply(matrix1, matrix2);
    }

    MatrixMultiplier*                getMatrixMultiplier(const
std::vector<std::vector<int>> &matrix1, const std::vector<std::vector<int>>
&matrix2) {
        if(matrix1.size() > 100000)
            return MatrixMultiplierStrassen::getInstance();

        return MatrixMultiplierSimple::getInstance();
    }
```

```
vector<int> solveLinearEquations(const vector<vector<int>> &inversedA, const vector<int> &B) {
    auto multiplier = getMatrixMultiplier(inversedA, {B});
    vector<vector<int>> AB = multiplyMatrices(inversedA, {B}, multiplier);

    vector<int> result;
    for (const auto &row: AB) {
        result.push_back(row[0]);
    }

    return result;
}
```

4. ОГЛЯД ТА ЗАСТОСУВАННЯ ПОВЕДІНКОВИХ ПАТЕРНІВ GOF

4.1 Огляд концепції поведінкових шаблонів

Поведінкові патерни проектування визначають способи взаємодії об'єктів один з одним та вирішення завдань з розподілення їх обов'язків. Ці патерни спрощують розвиток програмних систем, сприяючи збереженню гнучкості і розширюваності коду. Основна мета поведінкових патернів - полегшити спільну роботу об'єктів та забезпечити ефективний обмін інформацією між ними, що в цілому допомагає вирішувати різноманітні завдання в програмуванні, полегшуючи розробку, розширення і підтримку програмного забезпечення. [10].

4.2 Огляд та застосування патерну “Шаблон”

Патерн “Шаблон” (ще називають “Шаблонний метод”)- це поведінковий патерн проектування, який визначає скелет алгоритму у базовому класі і дозволяє дочірнім класам перевизначати деякі кроки алгоритму без зміни його структури. Такий підхід є потужним інструментом для створення гнучких та розширюваних алгоритмів, які можуть легко пристосовуватися до різних вимог застосування. Однак використання шаблонного методу може призвести до складності розуміння коду через необхідність вивчення його структури та взаємодії між класами. Деякі аспекти алгоритму можуть бути закритими для змін у віддільних підкласах, що обмежує його гнучкість. Також важливо уникати перевищення обсягу базового класу та забезпечити збалансований розподіл відповідальностей між базовим класом та його підкласами [11].

Дуже доцільним буде продемонструвати роботу цього шаблону на прикладі розв’язування систем лінійних рівнянь різними методами, в даному випадку розглянуться методи Гауса та Якобі.

Метод Гауса базується на послідовному застосуванні елементарних перетворень до матриці системи, з метою приведення її до трикутного або ступеневого вигляду, де потім можна легко знайти значення змінних. Він відомий своєю універсальністю, оскільки він застосовний для будь-якого типу систем лінійних рівнянь, незалежно від їх розмірності. Починаючи з першого рядка матриці, на кожному кроці виконуються елементарні операції над рядками, щоб зробити всі елементи під головною діагоналлю рівними нулю. Цей процес продовжується до тих пір, поки матриця не стане трикутною. Після прямого ходу здійснюється зворотній хід, під час якого значення змінних визначаються від останнього рядка до першого. Цей процес полягає у використанні обернених елементарних операцій для підведення змінних до значень. Цей метод може бути досить ефективним, якщо кількість операцій виявиться достатньою для конкретної системи. Однак метод Гауса може стати чутливим до помилок округлення, що може вплинути на точність результатів, особливо при великих розмірностях матриць або при обробці чисел з високою точністю. Крім того, він може вимагати значних обсягів пам'яті для збереження матриці, що може стати проблемою для систем з великою кількістю рівнянь та змінних. Метод Гауса не завжди ефективний для розріджених матриць, оскільки вони можуть вимагати значних обчислювальних ресурсів [12].

Метод Якобі є ітераційним, що базується на послідовному покращенні наближення до точного розв'язку шляхом ітераційного оновлення. Основна ідея полягає в тому, щоб в кожній ітерації застосовувати нові значення змінних, засновані на попередніх значеннях та на самій системі рівнянь. Спочатку вибирається початкове наближення до розв'язку системи лінійних рівнянь, далі на кожній ітерації для кожної змінної обчислюються нові значення, засновані на попередніх значеннях та на самій системі рівнянь. Після кожної ітерації перевіряється умова

збіжності, яка вказує, чи досягнуто достатньо близького наближення до точного розв'язку. Якщо так, процес завершується, інакше ітерації продовжуються. Процес продовжується до тих пір, поки не буде досягнуто достатньо точного наближення до точного розв'язку. Метод Якобі відомий своєю простотою реалізації, оскільки він використовує прості математичні операції та ітераційний підхід. Крім того, він забезпечує широку застосовність і може бути ефективним для розв'язання різноманітних систем лінійних рівнянь, включаючи великі та розріджені матриці. Однак метод Якобі може мати обмежену швидкість збіжності, особливо у випадку складних систем або погано умовлених матриць, що може вимагати більшої кількості ітерацій для досягнення точного розв'язку. Також, існує ризик того, що метод Якобі може не збігатися для деяких систем лінійних рівнянь, що може виникати у випадку погано умовлених матриць або складних структур системи [13].

Оскільки ми маємо 2 підходи, що використовуються в різних ситуаціях, доцільно буде реалізувати їх програмно за допомогою шаблонного методу для забезпечення зручного їх використання. Спочатку буде створено клас-шаблон LinearSystemSolver. Метод для вирішення системи лінійних рівнянь прийматиме матрицю коефіцієнтів та матрицю констант, а всередині будуть описані кроки розв’язку, такі як: перевірка передумов (чи можливо застосувати цей метод до вхідних даних), підготовка даних, обчислення, та виведення результату. Останній крок буде мати стандартну реалізацію в базовому класу, а решта чотири будуть перевизначені в дочірніх класах відповідно до методу, що там застосовується. Таким чином буде створено клас GaussSolver, у якого буде перевизначено лише метод з обчисленнями, адже обраний метод можна застосувати до будь-яких даних і ніякі підготовки заздалегідь не потрібні. Проте в класі JacobiSolver, що теж успадкує клас LinearSystemSolver, окрім

обчислень буде перевизначений метод з перевіркою передумов, бо його варто застосовувати до діагонально панівних матриць. Така умова дозволяє забезпечити збіжність методу Якобі, тобто гарантує, що ітераційний процес збігається до розв'язку системи лінійних рівнянь. Хоча існують випадки, коли метод Якобі може працювати навіть для матриць, які не є діагонально домінантними, але в цих випадках збіжність методу може бути непередбачуваною або навіть відсутньою. Таким чином, вимога діагональної домінантності є не тільки рекомендацією, але і важливою умовою для забезпечення коректної та ефективної роботи методу Якобі. Окрім того, у класі JacobiSolver буде перевизначено ще й метод підготовки даних, де буде задано допустиму похибку.

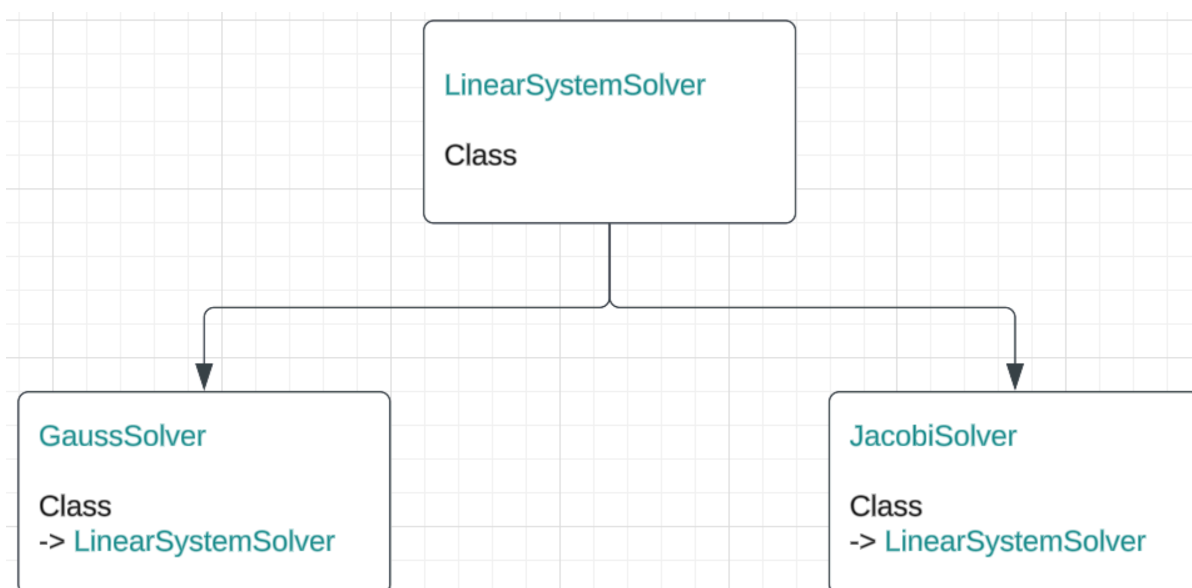

рис. 4.1 Діаграма класів використаних для патерну “Шаблон”

Кодове представлення:

```
class LinearSystemSolver {
        public:
```

```
    void solve(const std::vector<std::vector<double>> &coefficients, const std::vector<double> &constants) {
        // Крок 1: Перевірка передумов
        if (!preconditionsMet(coefficients, constants)) {
            std::cerr << "Помилка: Передумови для розв'язку системи не виконані.\n";
            return;
        }

        // Крок 2: Підготовка даних
        prepareData(coefficients, constants);

        // Крок 3: Обчислення
        computeSolution(coefficients, constants);

        // Крок 4: Виведення результатів
        displaySolution();
    }

protected:
    virtual bool
        preconditionsMet(const std::vector<std::vector<double>> &coefficients, const std::vector<double> &constants) = 0;

    virtual void
        prepareData(const std::vector<std::vector<double>> &coefficients, const std::vector<double> &constants) = 0;
```

```
        virtual void
         computeSolution(const std::vector<std::vector<double>> coefficients,
const std::vector<double> constants) = 0;

        virtual void displaySolution() const {
            std::cout << "Отриманий розв'язок:\n";
            for (size_t i = 0; i < solution_.size(); ++i) {
                std::cout << "x" << i + 1 << " = " << solution_[i] << std::endl;
            }
        };

        std::vector<double> solution_;
    };

class GaussSolver : public LinearSystemSolver {
    private:
                void    gaussianElimination(std::vector<std::vector<double>>
&coefficients, std::vector<double> &constants) {
            int n = coefficients.size();

            for (int i = 0; i < n; ++i) {
                int max_row = i;
                for (int k = i + 1; k < n; ++k) {
                                        if   (std::abs(coefficients[k][i])   >
std::abs(coefficients[max_row][i])) {
                        max_row = k;
                    }
                }
```

```
            if (max_row != i) {
                std::swap(coefficients[i], coefficients[max_row]);
                std::swap(constants[i], constants[max_row]);
            }

            for (int k = i + 1; k < n; ++k) {
                double factor = coefficients[k][i] / coefficients[i][i];
                constants[k] -= factor * constants[i];
                for (int j = i; j < n; ++j) {
                    coefficients[k][j] -= factor * coefficients[i][j];
                }
            }
        }

        solution_.resize(n);
        for (int i = n - 1; i >= 0; --i) {
            double sum = 0.0;
            for (int j = i + 1; j < n; ++j) {
                sum += coefficients[i][j] * solution_[j];
            }
            solution_[i] = (constants[i] - sum) / coefficients[i][i];
        }
    }

protected:
    // Перевірка передумов: у Гаусса завжди виконується
```

```
        bool preconditionsMet(const std::vector<std::vector<double>>
&coefficients,
                              const std::vector<double> &constants) override {
            return true;
        }

        // Підготовка даних: для Гаусса нічого не потрібно робити
        void
            prepareData(const std::vector<std::vector<double>> &coefficients,
const std::vector<double> &constants) override {}

        void
          computeSolution(const std::vector<std::vector<double>> coefficients,
const std::vector<double> constants) override {
            std::cout << "Розв'язання лінійної системи за методом Гаусса.\n";
            std::vector<std::vector<double>> tempCoefficients = coefficients;
            std::vector<double> tempConstants = constants;
            gaussianElimination(tempCoefficients, tempConstants);
            solution_ = tempConstants;
        }
    };

class JacobiSolver : public LinearSystemSolver {
    private:
        double tolerance_;

        std::vector<double>
```

```
        computeNewValues(const std::vector<std::vector<double>> &coefficients, const std::vector<double> &constants,
                         const std::vector<double> &oldValues) {
            int n = coefficients.size();
            std::vector<double> newValues(n, 0.0);
            for (int i = 0; i < n; ++i) {
                double sum = 0.0;
                for (int j = 0; j < n; ++j) {
                    if (j != i) {
                        sum += coefficients[i][j] * oldValues[j];
                    }
                }
                newValues[i] = (constants[i] - sum) / coefficients[i][i];
            }
            return newValues;
        }

    protected:
        // Перевірка передумов: метод Якобі збігається тільки для діагонально домінантних матриць
        bool preconditionsMet(const std::vector<std::vector<double>> &coefficients,
                              const std::vector<double> &constants) override {
            int n = coefficients.size();
            for (int i = 0; i < n; ++i) {
                double diagonalElement = std::abs(coefficients[i][i]);
                double sum = 0.0;
                for (int j = 0; j < n; ++j) {
```

```
                if (j != i) {
                    sum += std::abs(coefficients[i][j]);
                }
            }
            if (diagonalElement <= sum) {
                return false;
            }
        }
        return true;
    }

    void
        prepareData(const std::vector<std::vector<double>> &coefficients,
const std::vector<double> &constants) override {
        tolerance_ = 1e-6; // Задаємо допустиму похибку
    }

    void
        computeSolution(const std::vector<std::vector<double>> coefficients,
const std::vector<double> constants) override {
        std::cout << "Розв'язання лінійної системи за методом Якобі.\n";

        std::vector<double> currentValues = constants;
        std::vector<double> previousValues(constants.size(), 0.0);

        while (true) {
            previousValues = currentValues;
```

```
                currentValues = computeNewValues(coefficients, constants,
previousValues);
            double maxDiff = 0.0;
            for (size_t i = 0; i < currentValues.size(); ++i) {
                maxDiff = std::max(maxDiff, std::abs(currentValues[i] -
previousValues[i]));
            }
            if (maxDiff < tolerance_) {
                break;
            }
        }

        solution_ = currentValues;
    }
};
```

4.3 Огляд та застосування патерну “Стратегія”

Патерн “Стратегія” є поведінковим патерном проектування, суть якого у визначенні сімейства алгоритмів, успадковуванні кожного з них і забезпеченні їх взаємозамінності. Таким чином відкривається можливість змінювати алгоритм або поведінку об'єкта віддільно від самого об'єкта, що використовує цей алгоритм. Основна ідея полягає в тому, щоб виділити часто змінюваний аспект програми в окремий клас, інкапсулювати його, а потім зробити об'єкти цього класу взаємозамінними. Таким чином, можна легко змінювати алгоритм без зміни класу, що його використовує.

Даний підхід добре розкривається на прикладі реалізації алгоритмів знаходження власних чисел та векторів матриці. Буде розглянуто два способи: степеневий метод та метод Якобі.

Степеневий метод є найбільш поширеним, його алгоритм наступний: Нехай маємо матрицю А. Обирається початковий вектор x_0 (зазвичай випадковий), і нормалізується, щоб його довжина була рівна одиниці. Далі поки x не буде збігатися, обчислюється новий вектор $x = Ax$ і нормалізується, щоб його довжина була рівна одиниці. Після збіжності власне число можна отримати, обчисливши $\lambda = \frac{A_x * x}{x * x}$, а вектор x після збіжності буде наближеним до власного вектора, якому відповідає найбільше власне число. Важливо враховувати, що метод степенів збігається тільки до найбільшого за модулем власного числа [14].

Водночас ідея методу Якобі полягає в тому, щоб поступово діагоналізувати матрицю шляхом послідовного застосування матричних обертань до неї до тих пір, поки всі елементи поза головною діагоналлю не стануть дуже малими. Нехай є початкова матриця А, тоді Р - матриця обертання. Шукається недіагональний елемент a_{ij}, де $|a_{ij}|$ є найбільшим серед всіх недіагональних елементів. Після цього обчислюється кут обертання θ так, щоб елемент a_{ij} занулився, створюється матриця обертання Р, що містить косинуси і синуси кута обертання. Обчислюється нова матрицю $A = P^T AP$, де P^T - є транспонованою матрицею Р. Цей процес повторюється доти, доки всі елементи поза головною діагоналлю матриці А не стануть дуже малими. Після закінчення цих ітерацій процесу матриця А буде приблизно діагональною, власні числа матимуть значення на діагоналі, а власні вектори будуть утворювати стовпці матриці Р [15].

З точки же зору програми, буде створено загальний інтерфейс стратегії EigenSolverStrategy з методами для знаходження власних векторів і чисел. Тоді буде створено дочірні класи з реалізацією вищеописаних підходів у вигляд стратегії, а саме JacobiEigenSolverStrategy та PowerIterationEigenSolverStrategy. Після цього ці стратегії вже можна

використовувати у різних класах, наприклад буде створено EigenSolver, що прийматиме EigenSolverStrategy при створенні екземпляру і матиме змогу оперувати страгетією всередині своїх методів.

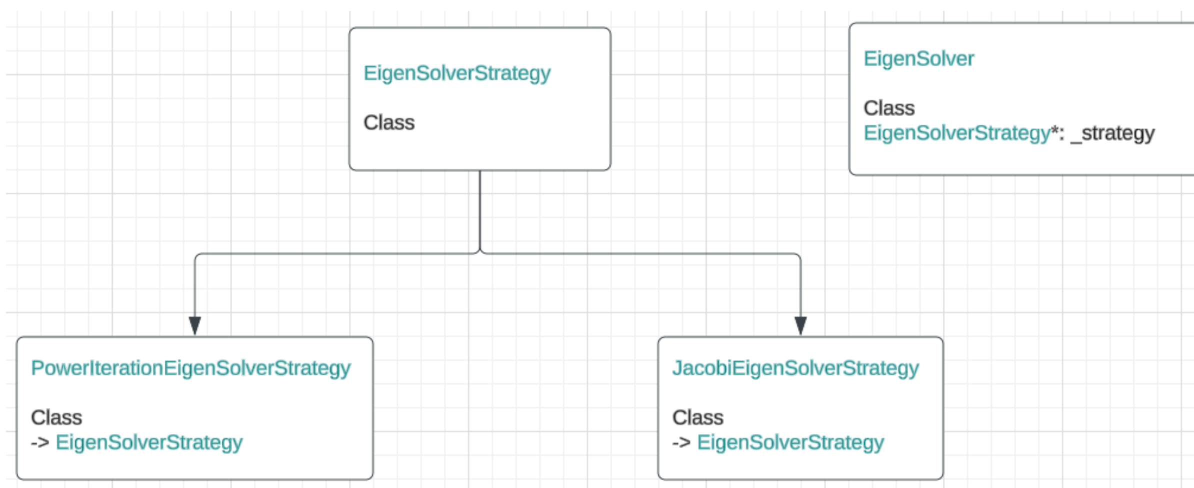

рис. 4.2 Діаграма класів використаних для патерну “Стратегія”

Програмні моделі матимуть наступний вигляд:

```
class EigenSolverStrategy {
    public:
      virtual ~EigenSolverStrategy() {}

      virtual std::vector<double> computeEigenvalues(const std::vector<std::vector<double>> &matrix) const = 0;

      virtual std::vector<std::vector<double>>
      computeEigenvectors(const std::vector<std::vector<double>> &matrix) const = 0;
    };
```

```
class JacobiEigenSolverStrategy : public EigenSolverStrategy {
    private:
      std::vector<double> diagonal_;

      bool checkAccuracy(const std::vector<std::vector<double>> &matrix,
double epsilon) const {
        int n = matrix.size();
        for (int i = 0; i < n; ++i) {
          for (int j = 0; j < n; ++j) {
            if (i != j && std::abs(matrix[i][j]) > epsilon) {
              return false;
            }
          }
        }
        return true;
      }

      std::vector<std::vector<double>> computeRotationMatrix(int p, int q,
double theta, int n) const {
                  std::vector<std::vector<double>>   rotationMatrix(n,
std::vector<double>(n, 0.0));
        for (int i = 0; i < n; ++i) {
          rotationMatrix[i][i] = 1.0;
        }
        rotationMatrix[p][p] = rotationMatrix[q][q] = std::cos(theta);
        rotationMatrix[p][q] = -std::sin(theta);
        rotationMatrix[q][p] = std::sin(theta);
        return rotationMatrix;
```

```
    }

    std::vector<std::vector<double>>
        computeFinalEigenvectors(const std::vector<std::vector<double>>
&initialEigenvectors) const {
        int n = initialEigenvectors.size();
                std::vector<std::vector<double>> finalEigenvectors(n,
std::vector<double>(n, 0.0));
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                for (int k = 0; k < n; ++k) {
                    finalEigenvectors[i][j] += initialEigenvectors[i][k] *
initialEigenvectors[k][j];
                }
            }
        }
        return finalEigenvectors;
    }

public:
                std::vector<double>    computeEigenvalues(const
std::vector<std::vector<double>> &matrix) const override {
        std::cout << "Обчислення власних чисел за методом Якобі.\n";
        int n = matrix.size();
        std::vector<std::vector<double>> A = matrix;
        double epsilon = 1e-6;
        while (!checkAccuracy(A, epsilon)) {
            for (int p = 0; p < n; ++p) {
```

```
                for (int q = p + 1; q < n; ++q) {
                    if (std::abs(A[p][q]) > epsilon) {
                        double phi = 0.5 * std::atan2(2 * A[p][q], A[q][q] - A[p][p]);
                        std::vector<std::vector<double>> R = computeRotationMatrix(p, q, phi, n);
                        A = MatrixOperations::matrixMultiplication(MatrixOperations::matrixTranspose(R), A);
                        A = MatrixOperations::matrixMultiplication(A, R);
                    }
                }
            }
        }

        std::vector<double> eigenvalues(n);
        for (int i = 0; i < n; ++i) {
            eigenvalues[i] = A[i][i];
        }
        return eigenvalues;
    }

    std::vector<std::vector<double>>
    computeEigenvectors(const std::vector<std::vector<double>> &matrix) const override {
        std::cout << "Обчислення власних векторів за методом Якобі.\n";
        int n = matrix.size();
        std::vector<std::vector<double>> A = matrix;
```

```
                        std::vector<std::vector<double>>    initialEigenvectors(n,
std::vector<double>(n, 0.0));
            for (int i = 0; i < n; ++i) {
                initialEigenvectors[i][i] = 1.0;
            }
            double epsilon = 1e-6; // Точність
            while (!checkAccuracy(A, epsilon)) {
                for (int p = 0; p < n; ++p) {
                    for (int q = p + 1; q < n; ++q) {
                        if (std::abs(A[p][q]) > epsilon) {
                            double phi = 0.5 * std::atan2(2 * A[p][q], A[q][q] -
A[p][p]);
                            std::vector<std::vector<double>>  R  =
computeRotationMatrix(p, q, phi, n);
                            A    =
MatrixOperations::matrixMultiplication(MatrixOperations::matrixTranspose(R)
, A);
                            A = MatrixOperations::matrixMultiplication(A, R);
                            initialEigenvectors    =
MatrixOperations::matrixMultiplication(initialEigenvectors, R);
                        }
                    }
                }
            }

            return computeFinalEigenvectors(initialEigenvectors);
        }
    };
```

```
class PowerIterationEigenSolverStrategy : public EigenSolverStrategy {
    private:
        void normalize(std::vector<double> &vec) const {
            double norm = 0.0;
            for (double val: vec) {
                norm += val * val;
            }
            norm = std::sqrt(norm);
            for (double &val: vec) {
                val /= norm;
            }
        }

        std::vector<double>
            matrixVectorMultiplication(const  std::vector<std::vector<double>>
&matrix, const std::vector<double> &vec) const {
            int n = matrix.size();
            std::vector<double> result(n, 0.0);
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    result[i] += matrix[i][j] * vec[j];
                }
            }
            return result;
        }
```

```
        double dotProduct(const std::vector<double> &vec1, const std::vector<double> &vec2) const {
            int n = vec1.size();
            double result = 0.0;
            for (int i = 0; i < n; ++i) {
                result += vec1[i] * vec2[i];
            }
            return result;
        }

    public:
        std::vector<double> computeEigenvalues(const std::vector<std::vector<double>> &matrix) const override {
            std::cout << "Обчислення власних чисел за методом степенів.\n";
            int n = matrix.size();
            std::vector<std::vector<double>> copiedMatrix(matrix);

            std::vector<double> eigenvalues(n, 0.0);
            double epsilon = 1e-6;
            for (int i = 0; i < n; ++i) {
                std::vector<double> x(n, 1.0);
                double lambda = 0.0;
                double lambda_prev = 1.0;
                int max_iterations = 1000;
                int iter = 0;
                while (std::abs(lambda - lambda_prev) > epsilon && iter < max_iterations) {
                    lambda_prev = lambda;
```

```
                std::vector<double> Ax = matrixVectorMultiplication(matrix,
x);
                lambda = dotProduct(Ax, x);
                normalize(x);
                ++iter;
            }
            eigenvalues[i] = lambda;
            for (int j = 0; j < n; ++j) {
                copiedMatrix[j][j] -= lambda;
            }
        }
        return eigenvalues;
    }

    std::vector<std::vector<double>>
    computeEigenvectors(const std::vector<std::vector<double>> &matrix)
const override {
        std::cout << "Обчислення власних векторів за методом
степенів.\n";
        int n = matrix.size();
        std::vector<std::vector<double>> copiedMatrix(matrix);

        std::vector<std::vector<double>> eigenvectors(n,
std::vector<double>(n, 0.0));
        double epsilon = 1e-6;
        for (int i = 0; i < n; ++i) {
            std::vector<double> x(n, 1.0);
            double lambda = 0.0;
```

```
            double lambda_prev = 1.0;
            int max_iterations = 1000;
            int iter = 0;
                while (std::abs(lambda - lambda_prev) > epsilon && iter <
max_iterations) {
              lambda_prev = lambda;
                std::vector<double> Ax = matrixVectorMultiplication(matrix,
x);
              lambda = dotProduct(Ax, x);
              normalize(x);
              ++iter;
            }
            eigenvectors[i] = x;

            for (int j = 0; j < n; ++j) {
              copiedMatrix[j][j] -= lambda;
            }
          }
          return eigenvectors;
        }
      };

class EigenSolver {
      private:
        const EigenSolverStrategy *strategy_;

      public:
```

```
        EigenSolver(const EigenSolverStrategy *strategy) : strategy_(strategy) {}

        void setStrategy(const EigenSolverStrategy *strategy) {
            strategy_ = strategy;
        }

        std::vector<double> computeEigenvalues(const std::vector<std::vector<double>> &matrix) const {
            return strategy_->computeEigenvalues(matrix);
        }

        std::vector<std::vector<double>> computeEigenvectors(const std::vector<std::vector<double>> &matrix) const {
            return strategy_->computeEigenvectors(matrix);
        }
    };
```

ВИСНОВКИ

У даній роботі було розглянуто основні поняття та принципи лінійної алгебри та патернів програмування. Дослідження включало аналіз та опис фундаментальних понять лінійної алгебри, таких як степеневий метод знаходження власних чисел, метод Гауса для розв’язування систем лінійних рівнянь та інших, а також розгляд і застосування всіх типів патернів проектування, таких як структурні, породжувальні, поведінкові. Крім того було розглянуто конкретні шаблони, такі як “Адаптер”, “Одинак”, “Стратегія” та інші. Побудовані програмні моделі для розв'язання деяких задач лінійної алгебри, таких як пошук власних чисел та векторів методом Якобі, розв'язання систем лінійних рівнянь методом Якобі, довели свою адекватність математичному представленню відповідних алгоритмів. Створено програмний код мовою програмування C++, заснований на зазначених патернах та принципах об'єктно-орієнтованого програмування. Цей код виявився ефективним для вирішення визначених задач лінійної алгебри. На основі виконаної роботи можна запропонувати наступні рекомендації для подальших досліджень: дослідження ефективності використання паралельного програмування, для вирішення більш складних проблем у сфері наукових обчислень, використання візуальних даних, оптимізація алгоритмів лінійної алгебри для забезпечення їх ефективності та швидкодії тощо.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке Патерн? [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/what-is-pattern.
2. Будай А. Дизайн-патерни — просто, як двері / Андрій Будай., 2012. – 90 с.
3. Shivanshu G. 5. Design Pattern:- Proxy [Електронний ресурс] / Goyal Shivanshu // Medium. – 2023. – Режим доступу до ресурсу: https://medium.com/nerd-for-tech/5-design-pattern-proxy-ea1a40a014ec.
4. Патерн проектування Proxy [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://javarush.com/ua/groups/posts/uk.2368.pattern-proektuvannja-proxy.
5. 8. Компонувальник — Composite [Електронний ресурс] – Режим доступу до ресурсу: http://www.znannya.org/?view=patterns-composite.
6. 4.2. Патерн Адаптер (Adapter) [Електронний ресурс] – Режим доступу до ресурсу: https://abitap.com/4-2-patern-adapter-adapter/.
7. Розганяєв Д. Породжувальні патерни проєктування [Електронний ресурс] / Денис Розганяєв – Режим доступу до ресурсу: https://blog.ithillel.ua/articles/creational-patterns.
8. Singleton Method Design Pattern [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/singleton-design-pattern/.
9. STRASSENS’S ALGORITHM FOR MATRIX MULTIPLICATION [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.topcoder.com/thrive/articles/strassenss-algorithm-for-matrix-multiplication.

10.3. Поведінкові патерни проектування (Behavioral) [Електронний ресурс] – Режим доступу до ресурсу: https://designpatternsphp.readthedocs.io/uk/latest/Behavioral/README.html.

11. Design Patterns - Template Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/design_pattern/template_pattern.htm.

12. 11.6: Solving Systems with Gaussian Elimination [Електронний ресурс] – Режим доступу до ресурсу: https://math.libretexts.org/Bookshelves/Algebra/Algebra_and_Trigonometry_1e_(OpenStax)/11%3A_Systems_of_Equations_and_Inequalities/11.06%3A_Solving_Systems_with_Gaussian_Elimination.

13. Strong D. M. Iterative Methods for Solving Ax = b - Jacobi\'s Method [Електронний ресурс] / David M. Strong – Режим доступу до ресурсу: https://maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-jacobis-method.

14. Верещак Р. Часткова проблема власних значень матриці. Степеневий метод [Електронний ресурс] / Ростислав Верещак. – 2024. – Режим доступу до ресурсу: https://www.mathros.net.ua/stepenevyj-metod.html.

15. Lambers J. Jacobi Methods / Jim Lambers. – Stanford: Stanford University, 2011. – 5 c.