ЗАСТОСУВАННЯ ПАТЕРНІВ ПРОЕКТУВАННЯ ДЛЯ ВИРІШЕННЯ ЗАДАЧ ЛІНІЙНОЇ АЛГЕБРИ/THE USAGE OF DESIGN PATTERN FOR LINEAR ALGEBRA SOFTWARE DEVELOPMENT

Керівник курсової роботи кандидат наук, доцент Бублик В.В.

Виконав студент ПМ-4 Зверьок Б.О.

Вступ

Парадигма патернів проектування в контексті лінійної алгебри відкриває можливості для побудови моделей, які добре відображають математичні концепції та операції, характерні та близькі для цієї області знань.

У дослідженні буде продемонстровано приклади застосування патернів проектування трьох типів: структурні, породжувальні та поведінкові.

Мета, завдання та методи дослідження

Мета - показати ефективність патернів GoF у вирішенні задач лінійної алгебри.

Завдання - Спроектувати програмні моделі з використанням принципів GoF для розв’язання деяких задач лінійної алгебри.

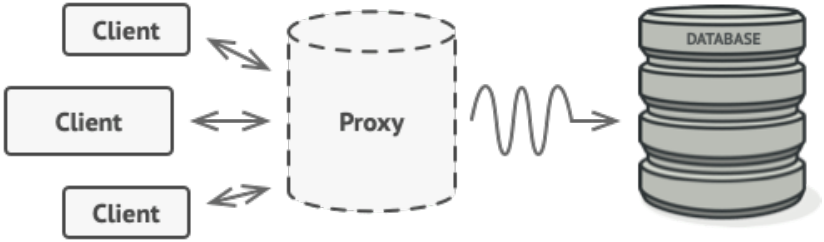
Методи дослідження - аналіз відповідної літератури та створення архітектури для системи

Огляд концепції шаблонів проектування

Патерн проектування GoF - це повторно використовуване рішення загальних проблем проектування програмного забезпечення.

Джерело: https://shorturl.at/0XejP

Огляд та застосування патерну “Замісник”



Джерело: https://shorturl.at/0PemR

“Замісник” для оптимізації роботи з матрицями

Знайти матрицю мінорів матриці А:

$$M_{ij} = det(A_{ij})$$

Знайти матрицю алгебраїчних доповнень:

$$C_{ij} = (-1)^{i+j} * M_{ij}$$

Знайти транспоновану матрицю алгебраїчних доповнень:

$$C^T = (C_{ij})^T$$

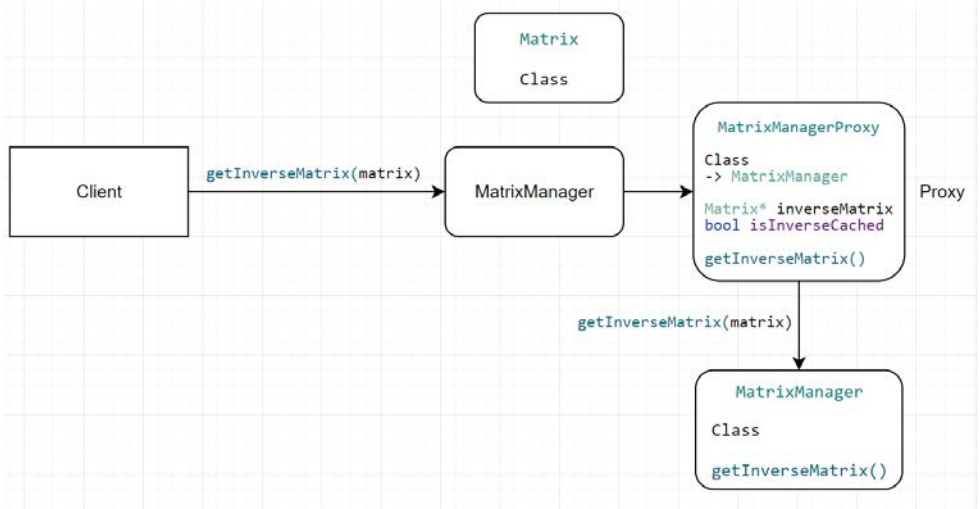
Знайти обернену А:

$$A^{-1} = C^T / det(A)$$

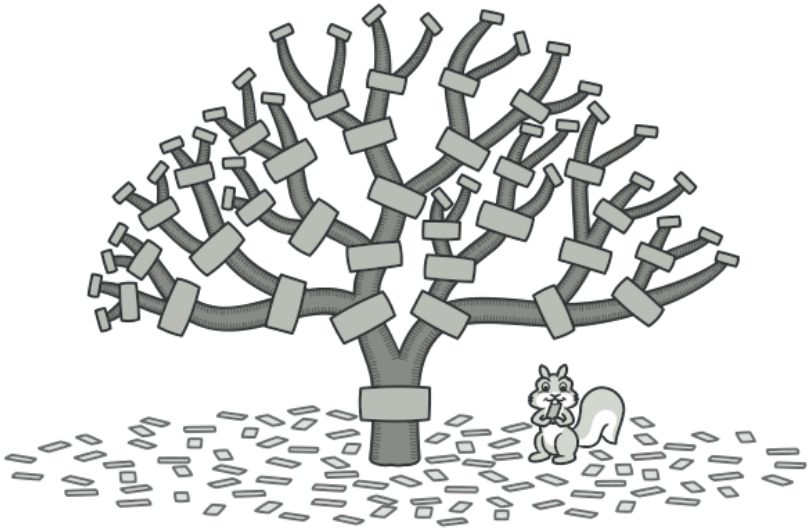
“Замісник” для оптимізації роботи з матрицями

```
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;
    }
};
```

“Замісник” для оптимізації роботи з матрицями



Огляд та застосування патерну “Компонувальник”

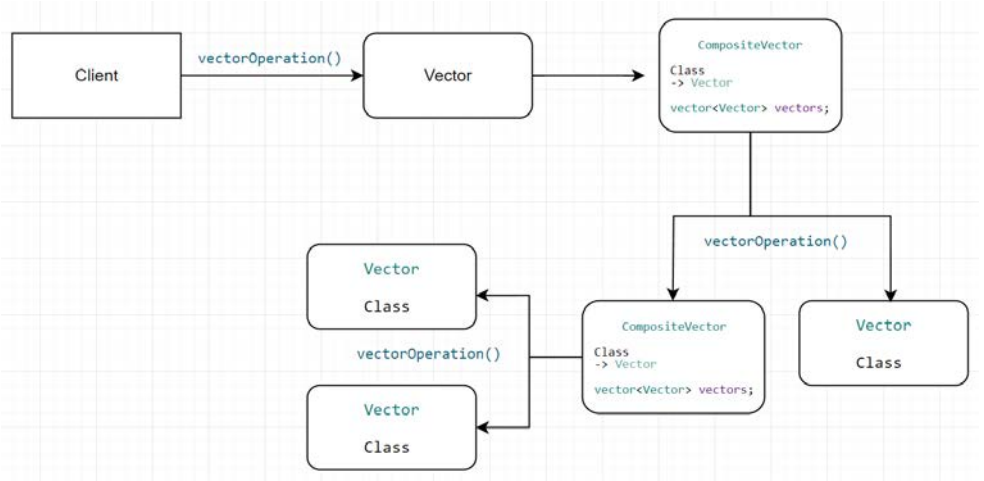


Джерело: https://shorturl.at/7S36v

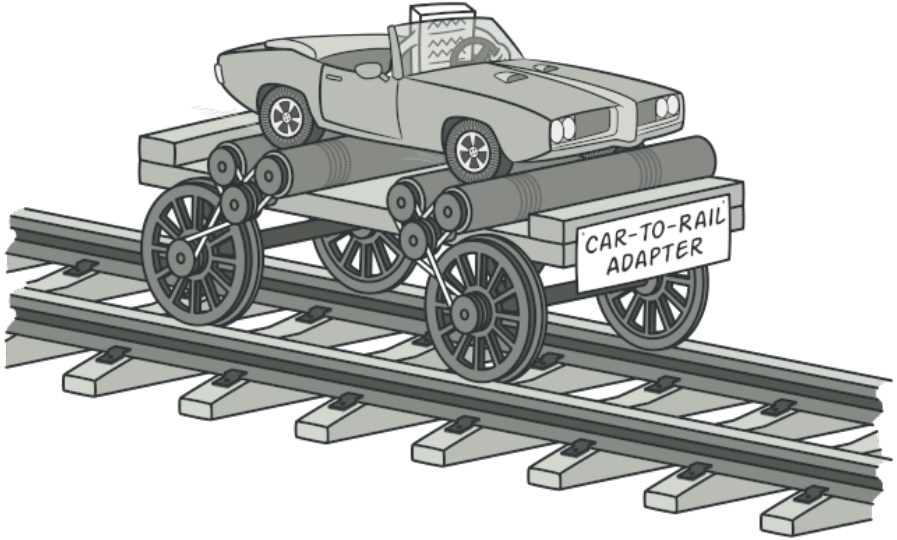
“Компонувальник” для організації деревоподібної структури векторів

```
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();
        }
    }
};
```

“Компонувальник” для організації деревоподібної структури векторів



Огляд та застосування патерну “Адаптер”

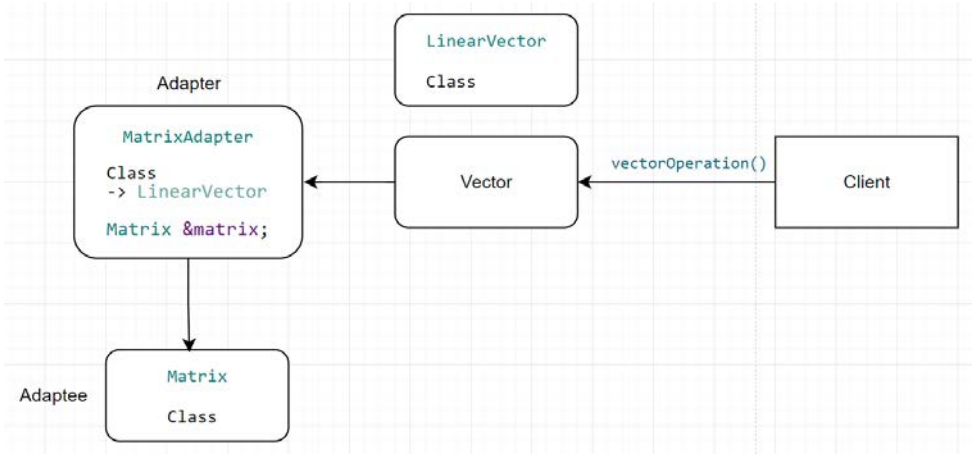


Джерело: https://shorturl.at/SQaxZ

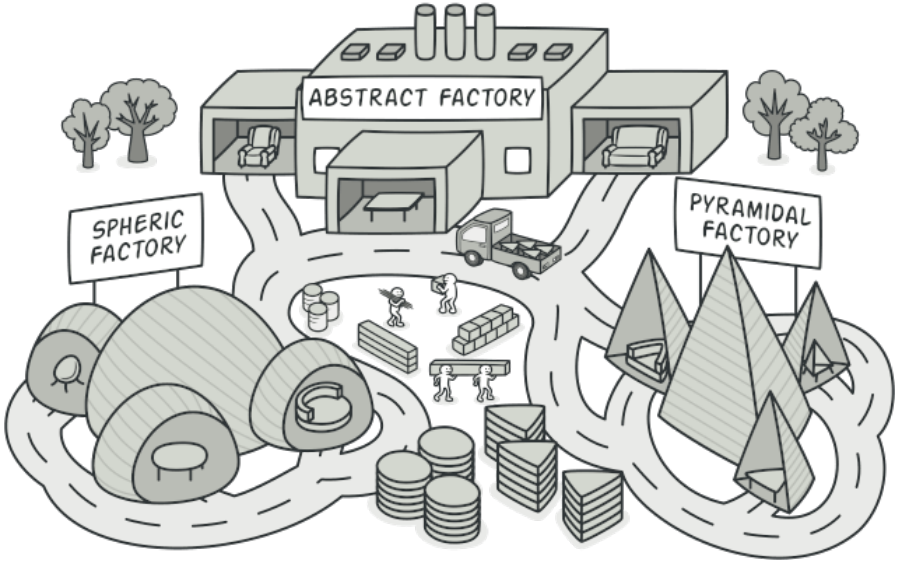
“Адаптер” для організації взаємодії між різними чисельними об’єктами

```cpp
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( row: i,  col: 0,  value: matrix.getValue( row: i,  col: 0) + other.getValue( index: i));
        }
    }

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

    virtual void print() const override {
        std::cout << "[";
        for (size_t i = 0; i < size(); ++i) {
            std::cout << matrix.getValue( row: i,  col: 0);
            if (i != size() - 1) {
                std::cout << ", ";
            }
        }
        std::cout << "]" << std::endl;
    }
};
```

“Адаптер” для організації взаємодії між різними чисельними об’єктами



Огляд та застосування патерну “Абстрактна фабрика”

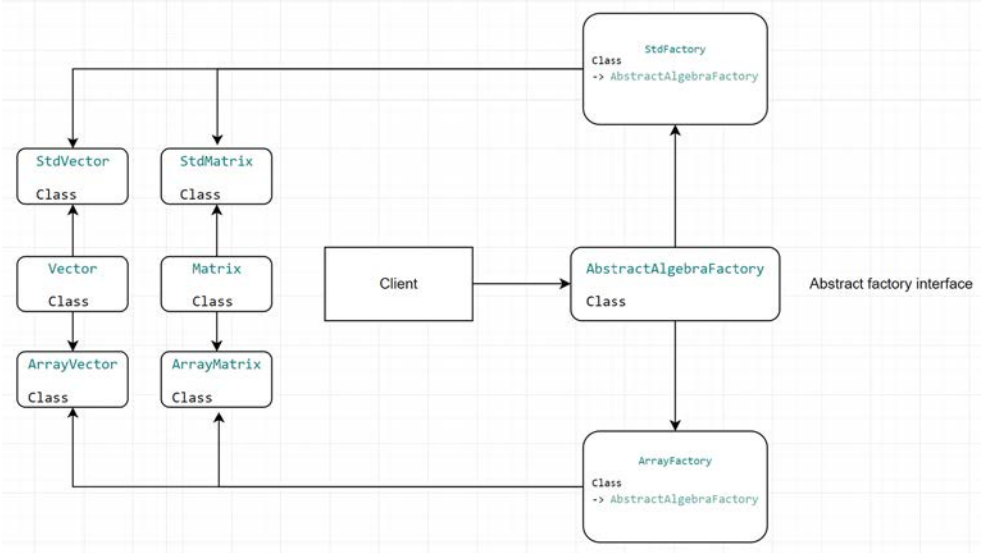


Джерело: https://shorturl.at/6s1Dg

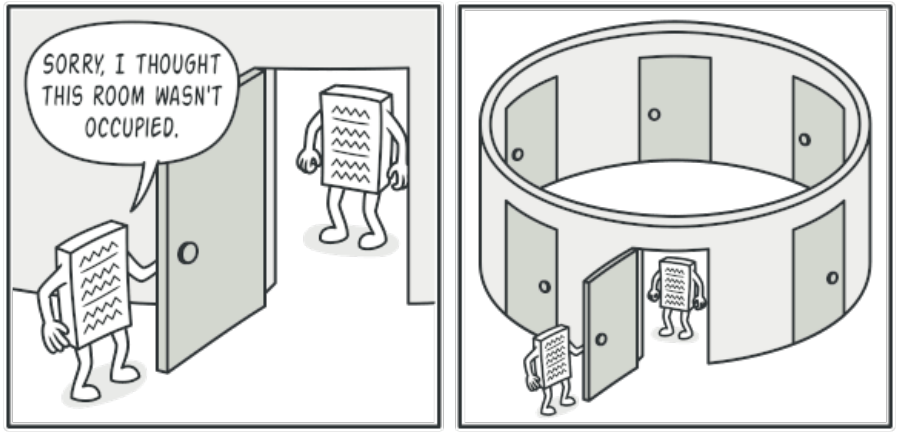
“ Абстрактна фабрика” для побудови екземплярів з різними числовими типами

```cpp
// Конкретна реалізація абстрактної фабрики для створення матриці і вектора на основі масиву чисел
class ArrayFactory : public AbstractAlgebraFactory {
public:
    Matrix createMatrix(int **values) override {
        return ArrayMatrix(values);
    }

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

// Конкретна реалізація абстрактної фабрики для створення матриці і вектора на основі стандартного вектора
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( values: convertToIntVector(values));
    }

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

“ Абстрактна фабрика” для побудови екземплярів з різними числовими типами



Огляд та застосування патерну “Одинак”



Джерело: https://shorturl.at/FT5zH

“Одинак” для розв’язання СЛАР методом множення

AX=B

A - матриця коефіцієнтів системи

X - матриця невідомих змінних

B - матриця вільних членів

X = A^(-1) * B

“Однак” для розв’язання СЛАР методом множення

Прямий метод множення (O(n^3)):

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

Метод Штрассена $O(n^{\log_2 7})$:

Розбиття на підматриці - $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

Обчислення проміжних матриць -

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Обчислення кінцевих підматриць -

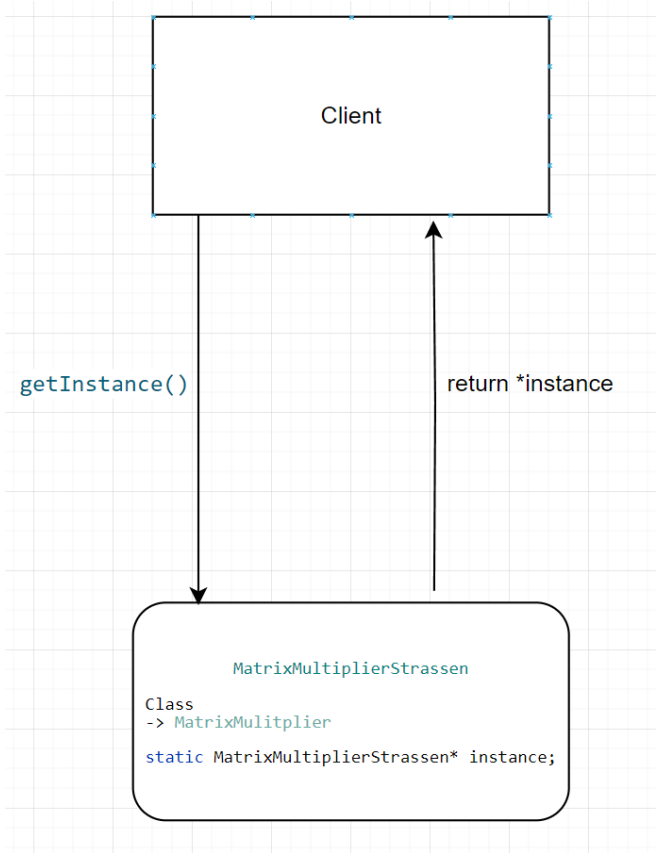
$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Комбінування кінцевих підматриць - $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

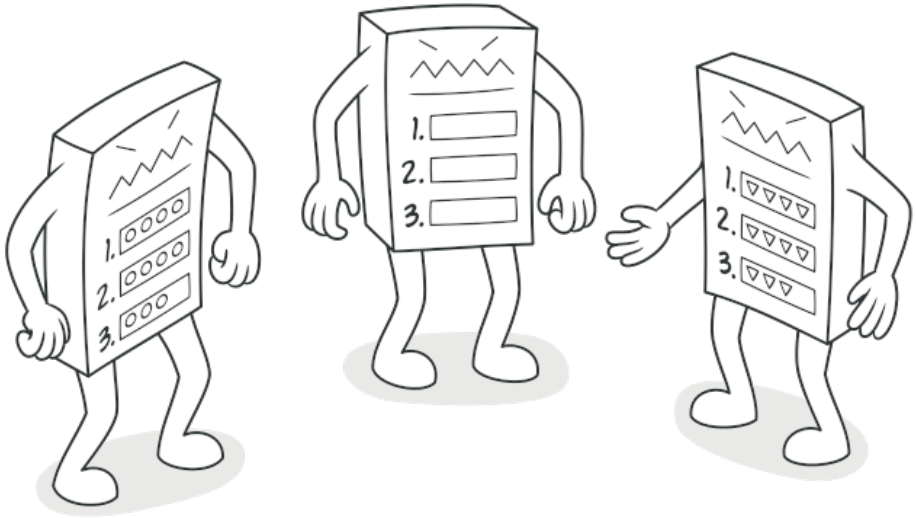
“Одинак” для розв’язання СЛАР методом множення

```
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( n: matrix1.size(), value: std::vector<int>( n: matrix2[0].size(), value: 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;
    }
};

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

MatrixMultiplier *
getMatrixMultiplier(const vector<vector<int>> &matrix1, const vector<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 : MatrixMultiplier * = getMatrixMultiplier( matrix1: inversedA, matrix2: {B});
    vector<vector<int>> AB = multiplyMatrices( matrix1: inversedA, matrix2: {B}, matrixMultiplier: multiplier);

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

    return result;
}
```

“Одинак” для розв’язання СЛАР методом множення



Огляд та застосування патерну “Шаблон”



Джерело: https://shorturl.at/LEfwD

“Шаблон” для розв’язання СЛАР методами Якобі та Гауса

Метод Гауса:

1. Прямий хід :

Починаємо з першого рядка і зводимо коефіцієнт під діагоналлю до нуля, використовуючи елементарні операції над рядками.

Повторюємо цей процес для всіх рядків, поки не отримаємо верхню трикутну матрицю.

2. Зворотний хід:

Починаємо з останнього рядка і використовуємо отримані значення з верхньотрикутної матриці, щоб визначити значення змінних.

Повторюємо цей процес для всіх рядків, рухаючись знизу вгору.

Ці кроки продовжуються до тих пір, поки не буде отримано розв'язок СЛАР.

“Шаблон” для розв’язання СЛАР методами Якобі та Гауса

Метод Якобі:

Вибирається початкове наближення x(0) На кожній ітерації k -

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

Перевірка умови збіжності -

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \epsilon$$

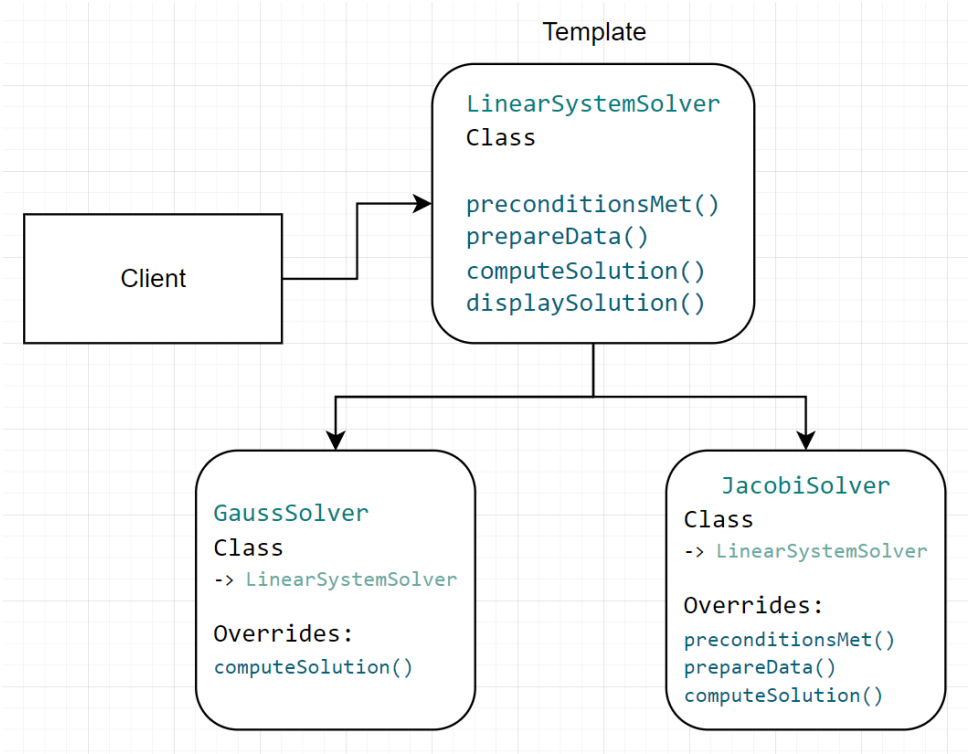
Якщо умова збіжності виконується, процес завершується -

$$\mathbf{x}^{(k+1)} \approx \mathbf{x}$$

“Шаблон” для розв’язання СЛАР методами Якобі та Гауса

```cpp
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_;
};
```

“Шаблон” для розв’язання СЛАР методами Якобі та Гауса



Огляд та застосування патерну “Стратегія”

Джерело: https://shorturl.at/DkAmM

“Стратегія” для знаходження власних чисел і векторів двома методами

Степеневий метод:

Обирається початковий x0.

До збігу обчислюється та нормалізується новий вектор x=Ax.

Пошук власного числа -

$$\lambda = \frac{A_x * x}{x * x}$$

Після збігу вектор x буде наближеним до власного вектора, якому відповідає найбільше за модулем власне число.

“Стратегія” для знаходження власних чисел і векторів двома методами

Метод Якобі:

Шукається недіагональний елемент aij, де |aij| є найбільшим серед всіх недіагональних елементів.

Обчислюється кут обертання θ, створюється матриця обертання P, що містить косинуси і синуси кута обертання.

Обчислюється нова матриця - $A = P^T AP$

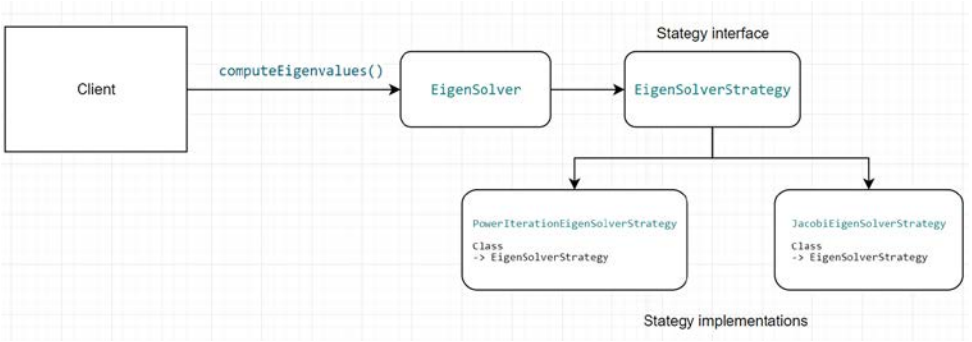
Процес повторюється доти, доки всі елементи поза головною діагоналлю матриці A не стануть дуже малими.

В кінці значення діагоналі A будуть власними числами, а власні вектори будуть утворювати стовпці матриці P.

“Стратегія” для знаходження власних чисел і векторів двома методами

```cpp
// Базовий клас стратегії для обчислення власних чисел та власних векторів
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 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);
    }
};
```

“Стратегія” для знаходження власних чисел і векторів двома методами



Висновки

1. Побудовані програмні моделі для розв'язання задач лінійної алгебри: пошук власних чисел та векторів степеневим методом та методом Якобі , розв'язання систем лінійних рівнянь методом Гауса та Якобі, множення методом Штрассена та інші.

2. Програмні моделі довели свою адекватність математичному представленню відповідних алгоритмів

3. Створено програмний код мовою програмування C++, заснований на зазначених патернах та принципах об'єктно-орієнтованого програмування.

Дякую за увагу