МИНИСТЕРСТВО ТРАНСПОРТА РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА (МИИТ)»

Кафедра ЦТУТП

**Отчёт**

По лабораторной работе №1  
по дисциплине «Корпоративные информационные системы»

Тема: «Реализация CDI-компонента»

Вариант №4

Выполнил: Иванов Н. С.

Группа: УИС-411

Преподаватель: доц. Кафедры ЦТУТП

Козьяков П. О.

­

Москва 2024 г.

**СОДЕРЖАНИЕ**

[ПОСТАНОВКА ЗАДАЧИ 3](#_Toc180325056)

[ХОД ВЫПОЛНЕНИЯ РАБОТЫ 4](#_Toc180325057)

[РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ 6](#_Toc180325058)

[КОД ПРОГРАММЫ 8](#_Toc180325059)

[ВЫВОД 14](#_Toc180325060)

# **ПОСТАНОВКА ЗАДАЧИ**

Создать компонент CDI, выполняющий нахождение ранга матрицы различными способами (по определению, методом окаймляющих миноров, по методу Гаусса). Выбор способа нахождение ранга матрицы должен быть реализован через указание соответствующей аннотации или посредством указания класса в XML файле. Посредством перехватчиков провести сравнение по времени эффективности различных методов.

# **ХОД ВЫПОЛНЕНИЯ РАБОТЫ**

Данная лабораторная работа реализует вычисление ранга матрицы с использованием CDI (Contexts and Dependency Injection) для инверсии управления и внедрения зависимостей.

Код начинается с определения аннотаций @BY\_DEFINITION, @GAUSSIAN\_METHOD и @MINORS\_METHOD. Эти аннотации служат квалификаторами для CDI и позволяют указать, какой конкретный метод расчета ранга матрицы должен использоваться. Все аннотации помечены как @Qualifier и применимы к полям, типам и методам.

Интерфейс IMatrixRankCalculator определяет метод calculateRank, который используется для вычисления ранга матрицы. Далее идут три реализации этого интерфейса:

1. **RankByDefinition** — вычисляет ранг матрицы по определению, обнуляя элементы столбцов ниже текущего элемента и уменьшая ранг, если элементы строки оказываются нулевыми.
2. **RankByGaussianElimination** — вычисляет ранг методом Гаусса, постепенно приводя матрицу к ступенчатому виду.
3. **RankByMinorsMethod** — вычисляет ранг с использованием окаймляющих миноров, проверяя наличие ненулевого определителя для различных подматриц.

Класс MatrixRankCalculator служит компонентом, который инъектирует одну из реализаций IMatrixRankCalculator в зависимости от выбранной аннотации. CDI автоматически подбирает реализацию на основе указанной аннотации (@BY\_DEFINITION, @GAUSSIAN\_METHOD или @MINORS\_METHOD).

Для профилирования времени выполнения используется перехватчик ProfileInterceptor. Он измеряет время выполнения метода calculateRank с помощью Java API для работы с временем (наносекундная точность).

Класс Laboratory1 является точкой входа программы. В нем создается контейнер CDI с использованием библиотеки Weld, и через контейнер создается экземпляр MatrixRankCalculator. В матрице размером 3x3 (заданный пример) рассчитывается ранг с помощью указанного метода, а результат и время выполнения выводятся на консоль.

# **РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ**

Таблица 1 – Результаты работы кода

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | T1 (нс) | T2 (нс) | T3 (нс) | T4 (нс) |
| По определению | 153000 | 144400 | 152400 | 143300 |
| Методом окаймляющих миноров | 163200 | 159100 | 161600 | 168800 |
| По методу Гаусса | 175100 | 143000 | 140000 | 142000 |

Среднее время работы для каждого метода в миллисекундах (мс):

По определению: 0.1483 мс

Методом окаймляющих миноров: 0.1632 мс

По методу Гаусса: 0.1500 мс ​

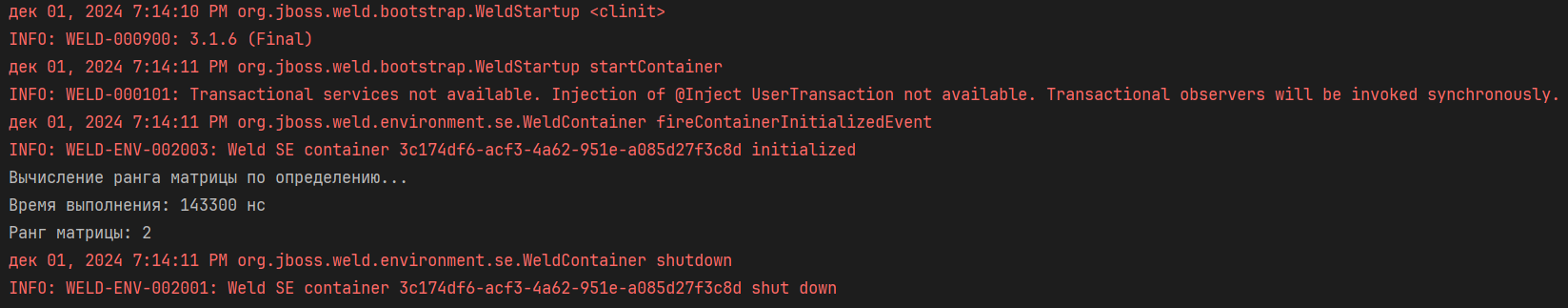


Рисунок 1 – Результат работы программы методом «по определению»

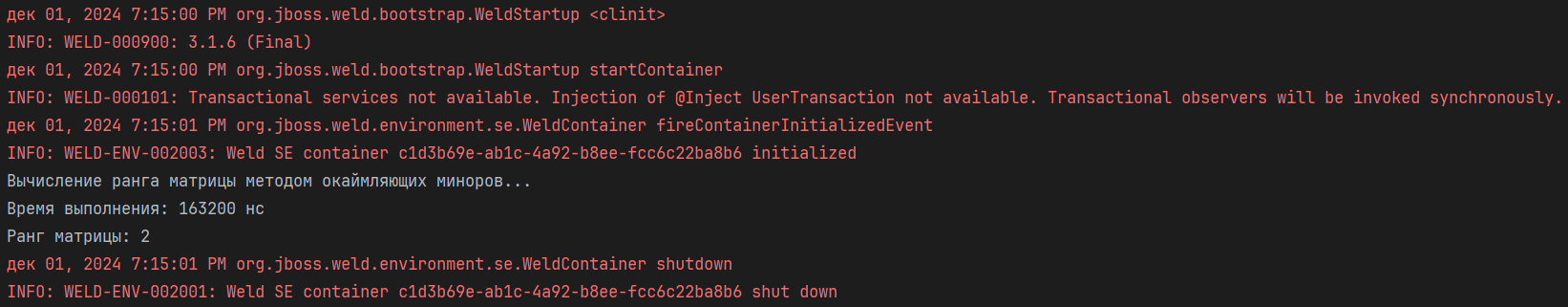


Рисунок 2 – Результат работы программы методом «окаймляющих миноров»

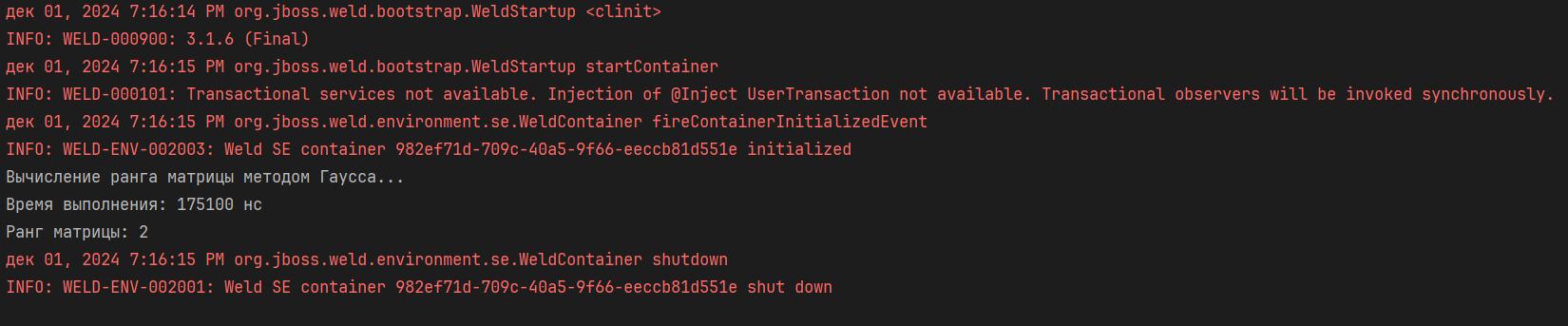


Рисунок 3 – Результат работы программы методом Гаусса

# **КОД ПРОГРАММЫ**

package Lab1\_CDI.Annotation;  
  
import javax.inject.Qualifier;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.METHOD})  
public @interface BY\_DEFINITION {  
}

package Lab1\_CDI.Annotation;  
  
import javax.inject.Qualifier;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.METHOD})  
public @interface GAUSSIAN\_METHOD {  
}

package Lab1\_CDI.Annotation;  
  
import javax.inject.Qualifier;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.METHOD})  
public @interface MINORS\_METHOD {  
}

package Lab1\_CDI.Calculator;  
  
public interface IMatrixRankCalculator {  
 int calculateRank(double[][] matrix);  
}

package Lab1\_CDI.Calculator;  
  
import Lab1\_CDI.Annotation.BY\_DEFINITION;  
  
@BY\_DEFINITION  
public class RankByDefinition implements IMatrixRankCalculator {  
  
 @Override  
 public int calculateRank(double[][] matrix) {  
 System.out.println("Вычисление ранга матрицы по определению...");  
 return calculate(matrix);  
 }  
  
 private int calculate(double[][] matrix) {  
 int rows = matrix.length;  
 int cols = matrix[0].length;  
 int rank = Math.min(rows, cols);  
  
 for (int i = 0; i < rank; i++) {  
 if (matrix[i][i] != 0) {  
 for (int j = 0; j < rows; j++) {  
 if (j != i) {  
 double ratio = matrix[j][i] / matrix[i][i];  
 for (int k = 0; k < cols; k++) {  
 matrix[j][k] -= ratio \* matrix[i][k];  
 }  
 }  
 }  
 } else {  
 boolean reduce = true;  
 for (int j = i + 1; j < rows; j++) {  
 if (matrix[j][i] != 0) {  
 double[] temp = matrix[i];  
 matrix[i] = matrix[j];  
 matrix[j] = temp;  
 reduce = false;  
 break;  
 }  
 }  
 if (reduce) {  
 rank--;  
 for (int j = 0; j < rows; j++) {  
 matrix[j][i] = matrix[j][rank];  
 }  
 }  
 i--;  
 }  
 }  
 return rank;  
 }  
}

package Lab1\_CDI.Calculator;  
  
import Lab1\_CDI.Annotation.GAUSSIAN\_METHOD;  
  
@GAUSSIAN\_METHOD  
public class RankByGaussianElimination implements IMatrixRankCalculator {  
  
 @Override  
 public int calculateRank(double[][] matrix) {  
 System.out.println("Вычисление ранга матрицы методом Гаусса...");  
 return calculate(matrix);  
 }  
  
 private int calculate(double[][] matrix) {  
 int rows = matrix.length;  
 int cols = matrix[0].length;  
 int rank = cols;  
  
 for (int row = 0; row < rank; row++) {  
 if (matrix[row][row] != 0) {  
 for (int col = 0; col < rows; col++) {  
 if (col != row) {  
 double ratio = matrix[col][row] / matrix[row][row];  
 for (int i = 0; i < rank; i++) {  
 matrix[col][i] -= ratio \* matrix[row][i];  
 }  
 }  
 }  
 } else {  
 boolean reduce = true;  
 for (int i = row + 1; i < rows; i++) {  
 if (matrix[i][row] != 0) {  
 double[] temp = matrix[row];  
 matrix[row] = matrix[i];  
 matrix[i] = temp;  
 reduce = false;  
 break;  
 }  
 }  
 if (reduce) {  
 rank--;  
 for (int i = 0; i < rows; i++) {  
 matrix[i][row] = matrix[i][rank];  
 }  
 }  
 row--;  
 }  
 }  
 return rank;  
 }  
}

package Lab1\_CDI.Calculator;  
  
import Lab1\_CDI.Annotation.MINORS\_METHOD;  
  
@MINORS\_METHOD  
public class RankByMinorsMethod implements IMatrixRankCalculator {  
  
 @Override  
 public int calculateRank(double[][] matrix) {  
 System.out.println("Вычисление ранга матрицы методом окаймляющих миноров...");  
 return calculate(matrix);  
 }  
  
 private int calculate(double[][] matrix) {  
 int rows = matrix.length;  
 int cols = matrix[0].length;  
 int rank = Math.min(rows, cols);  
  
 while (rank > 0) {  
 if (hasNonZeroMinor(matrix, rank)) {  
 return rank;  
 }  
 rank--;  
 }  
 return 0;  
 }  
  
 private boolean hasNonZeroMinor(double[][] matrix, int size) {  
 int rows = matrix.length;  
 int cols = matrix[0].length;  
  
 for (int i = 0; i <= rows - size; i++) {  
 for (int j = 0; j <= cols - size; j++) {  
 if (calculateDeterminant(subMatrix(matrix, i, j, size)) != 0) {  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 private double[][] subMatrix(double[][] matrix, int rowStart, int colStart, int size) {  
 double[][] subMatrix = new double[size][size];  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 subMatrix[i][j] = matrix[rowStart + i][colStart + j];  
 }  
 }  
 return subMatrix;  
 }  
  
 private double calculateDeterminant(double[][] matrix) {  
 int n = matrix.length;  
 if (n == 1) {  
 return matrix[0][0];  
 }  
 if (n == 2) {  
 return matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0];  
 }  
 double determinant = 0;  
 for (int i = 0; i < n; i++) {  
 determinant += Math.pow(-1, i) \* matrix[0][i] \* calculateDeterminant(minor(matrix, 0, i));  
 }  
 return determinant;  
 }  
  
 private double[][] minor(double[][] matrix, int row, int col) {  
 int n = matrix.length;  
 double[][] minor = new double[n - 1][n - 1];  
 int r = 0;  
 for (int i = 0; i < n; i++) {  
 if (i == row) continue;  
 int c = 0;  
 for (int j = 0; j < n; j++) {  
 if (j == col) continue;  
 minor[r][c] = matrix[i][j];  
 c++;  
 }  
 r++;  
 }  
 return minor;  
 }  
}

package Lab1\_CDI.Component;  
  
import Lab1\_CDI.Annotation.BY\_DEFINITION;  
import Lab1\_CDI.Annotation.GAUSSIAN\_METHOD;  
import Lab1\_CDI.Annotation.MINORS\_METHOD;  
import Lab1\_CDI.Calculator.IMatrixRankCalculator;  
import Lab1\_CDI.Interceptor.ProfileInterceptor;  
import lombok.NoArgsConstructor;  
  
import javax.inject.Inject;  
import javax.interceptor.Interceptors;  
  
@Interceptors(ProfileInterceptor.class)  
@NoArgsConstructor  
public class MatrixRankCalculator {  
  
 @Inject  
// @BY\_DEFINITION  
 @MINORS\_METHOD  
// @GAUSSIAN\_METHOD  
 private IMatrixRankCalculator calculator;  
  
 public int calculateRank(double[][] matrix) {  
 return calculator.calculateRank(matrix);  
 }  
}

package Lab1\_CDI.Interceptor;  
  
import lombok.NoArgsConstructor;  
  
import javax.interceptor.AroundInvoke;  
import javax.interceptor.InvocationContext;  
  
@NoArgsConstructor  
public class ProfileInterceptor {  
  
 @AroundInvoke  
 public Object profileExecution(InvocationContext context) throws Exception {  
 long startTime = System.nanoTime();  
 try {  
 return context.proceed();  
 } finally {  
 long endTime = System.nanoTime();  
 System.out.println("Время выполнения: " + (endTime - startTime) + " нс");  
 }  
 }  
}

package Lab1\_CDI;  
  
import Lab1\_CDI.Component.MatrixRankCalculator;  
import org.jboss.weld.environment.se.Weld;  
import org.jboss.weld.environment.se.WeldContainer;  
  
public class Laboratory1 {  
 protected static Weld weld;  
 protected static WeldContainer container;  
  
 public static void main(String[] args) {  
 weld = new Weld();  
 container = weld.initialize();  
  
 try {  
 MatrixRankCalculator rankCalculator = container.select(MatrixRankCalculator.class).get();  
  
 double[][] matrix = {  
 {1, 2, 3},  
 {4, 5, 6},  
 {7, 8, 9}  
 };  
  
 int rank = rankCalculator.calculateRank(matrix);  
 System.out.println("Ранг матрицы: " + rank);  
  
 } finally {  
 weld.shutdown();  
 }  
 }  
}

# **ВЫВОД**

Реализованная работа демонстрирует использование CDI для управления зависимостями, что упрощает замену алгоритмов вычисления ранга матрицы. Добавление перехватчика позволяет профилировать выполнение методов, повышая информативность приложения. Работа показывает преимущества инверсии управления и декомпозиции задач в программировании.