**ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ІВАНА ФРАНКА**

Факультет прикладної математики та інформатики

(повне найменування назва факультету)

Кафедра програмування

(повна назва кафедри)

**КУРСОВА РОБОТА**

на тему:

Візуалізація операцій над розрідженими матрицями в середовищі Pharo мовою Smalltalk

Студента 5 курсу, групи ПМІ-51м ,

напряму підготовки інформатика

Кормушина Я.К.

(прізвище та ініціали)

Керівник доц. Ярошко С.А.

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_\_\_

Львів – 2019

ЗМІСТ

[Вступ 3](#_Toc9418473)

[1. Формулювання завдання 10](#_Toc9418474)

[2. Огляд засобів Pharo 11](#_Toc9418475)

[3.1. Основні інструменти середовища 11](#_Toc9418476)

[3.2. Візуалізаційний засіб Roassal 13](#_Toc9418477)

[3.2.1. Композиція відображень 13](#_Toc9418478)

[3.2.2. Доповнення Roassal 16](#_Toc9418479)

[3.3. Пакет Reflectivity 17](#_Toc9418480)

[3.3.1. Клас MetaLink 17](#_Toc9418481)

[3.3.2. Приклади створення метапосилань 18](#_Toc9418482)

[3.3.3. Видалення метапосилань 20](#_Toc9418483)

[4. Реалізація 21](#_Toc9418484)

[4.1. Візуалізатори 26](#_Toc9418485)

[4.1.1. Візуалізатор операцій над матрицями на основі списку координат 27](#_Toc9418486)

[4.1.2. Візуалізатор операцій над матрицями на основі колекції списків 29](#_Toc9418487)

[4.1.3. Візуалізатор операцій над матрицями на основі мережі списків 30](#_Toc9418488)

[5. Аналіз отриманих результатів 31](#_Toc9418489)

[6. Висновки 37](#_Toc9418490)

[Список використаної літератури 38](#_Toc9418491)

[Додаток А 39](#_Toc9418492)

# Вступ

**Розріджена матриця** - це матриця, більша частина елементів якої дорівнює нулю. Концептуально, розрідженість відповідає системам, які слабо зв'язані. Це поняття корисне в комбінаториці і прикладних областях, таких як мережева теорія, які мають низьку щільність значущих даних або з'єднань. Системи рівнянь з розрідженими матрицями виникають, зокрема, в задачах аналізу міцності конструкцій у цивільному та промисловому будівництві, в авіабудуванні, ракетобудуванні, суднобудуванні, де застосовується метод скінченних елементів.

Вагому роль в таких областях відіграють розріджені матриці, які дозволяють зберігати лише певну частину даних, тим самим зменшуючи час виконання обчислень та обсяг пам’яті, необхідний для їх збереження.

Існує багато способів реалізації розріджених матриць, кожен з яких має свої переваги та недоліки. Деякі з них мають доволі складні внутрішні структури, формальний опис яких зрозуміти важче, ніж їх графічну репрезентацію. Відповідно до збільшення складності внутрішньої структури таких матриць зростає складність реалізації операцій над ними, таких як додавання, множення, транспонування тощо.

В результаті попередніх робіт ми реалізували розріджені матриці з різними внутрішніми структурами, а саме: список координат, колекція списків і мережа списків. Також ми розробили засоби для відображення цих структур.

Наведемо стислий опис кожного типу матриці, всіх структурних елементів їх програмної реалізації та класів для їх відображення. Всі класи належать до пакету LNU-SparseMatrix. Кожна реалізація складається з двох класів: елемента матриці і самої матриці.

Структура «список координат» відповідає такій схемі для загального випадку матриці розмірністю [nx m] елементів:

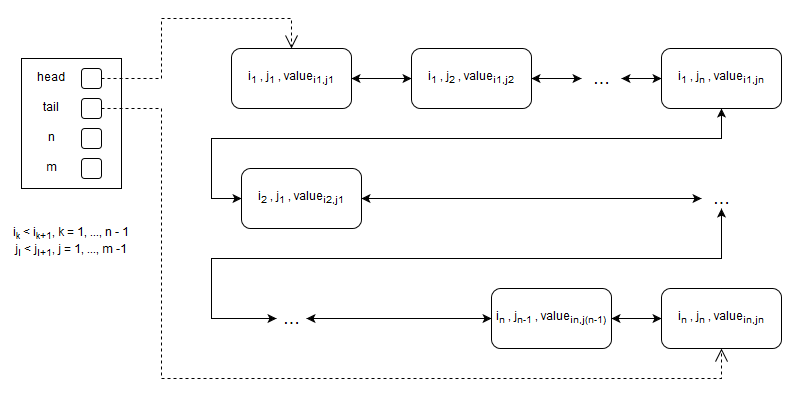


Рисунок 1. Внутрішня структура "список координат"

Ця реалізація зберігає кількість рядків матриці n та кількість стовпців m. Відповідно вона зберігає перший і останній елемент (голову та хвіст) двозв’язного списку елементів (рядок, стовпець, значення), де елементи посортовані у порядку «змійки», тобто за останнім елементом попереднього рядка знаходиться перший елемент наступного.

Матриця на основі списку координат реалізована такими класами: COOMatrixNode (Рис. 2) та COOSparseMatrix (Рис. 3).

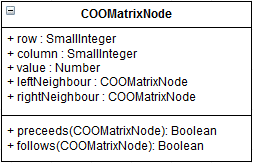
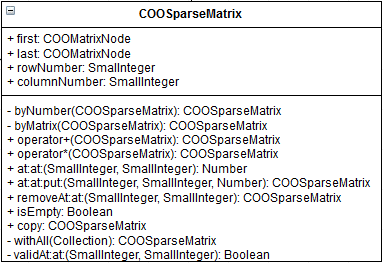
 

Рисунок 2. Клас COOMatrixNode Рисунок 3. Клас COOSparseMatrix

Особливістю класу елемента матриці (COOMatrixNode) є те, що кожен елемент містить:

* посилання на правого сусіда (rightNeighbour);
* посилання на лівого сусіда (leftNeighbour).

Структура «колекція списків» відповідає такій схемі:

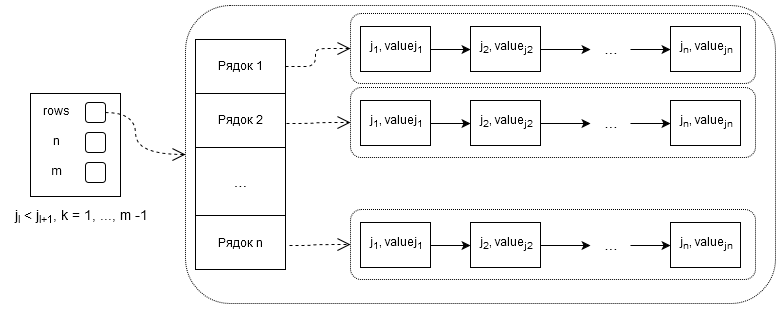


Рисунок 4. Внутрішня структура "колекція списків"

Ця реалізація зберігає по одному списку на кожен рядок, де елементи (стовпець, значення) розташовані в порядку зростання індексу стовпця.

Реалізація матриці на основі колекції списків включає класи LILSparseMatrixNode (Рис. 5) та LILSparseMatrix (Рис. 6).

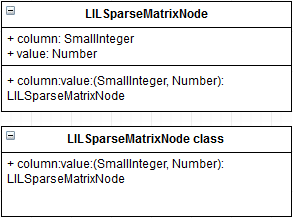
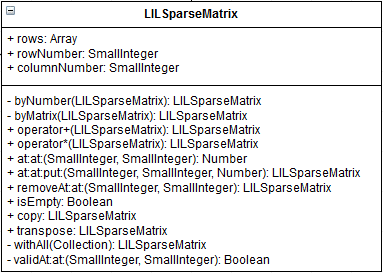


Рисунок 5. Клас LILSparseMatrixNode

Рисунок 6. Клас LILSparseMatrix

Структура «мережа списків» відповідає такій схемі:

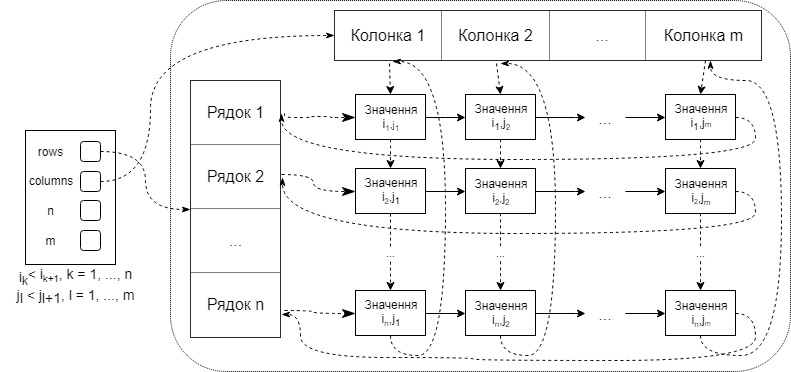


Рисунок 7. Внутрішня структура "мережа списків"

Ця реалізація зберігає по одному елементу (заголовку) на кожен рядок і стовпець, де елементи (значення, рядок, стовпець) розташовані в порядку зростання індексу рядка і стовпця відповідно.

Реалізація матриці на основі колекції списків включає класи TWMatrixNode (Рис. 8) та TWSparseMatrix (Рис. 9).

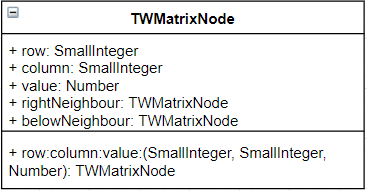


Рисунок 8. Клас TWMatrixNode

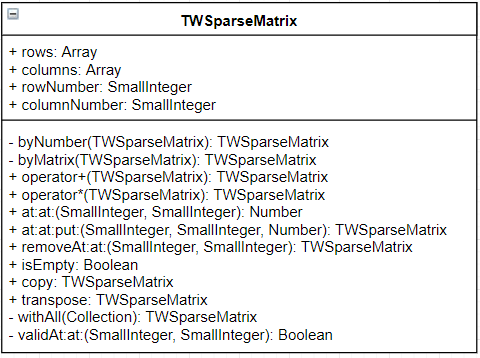


Рисунок 9. Клас TWSparseMatrix

Особливістю класу елемента матриці TWMatrixNode є те, що кожен елемент містить:

* посилання на правого сусіда (rightNeighbour);
* посилання на сусіда знизу (belowNeighbour).

Візуалізатори для кожного типу розрідженої матриці мають спільну поведінку, яка визначена в батьківському класі MatrixVisualizer. Проте деякі речі, зокрема інтерпретація колекції елементів, отриманих з матриці-джерела, та задання відношень між елементами визначені для кожного візуалізатора окремо, в силу різниці внутрішньої структури матриць.

Ієрархія класів наступна (Рис. 10):

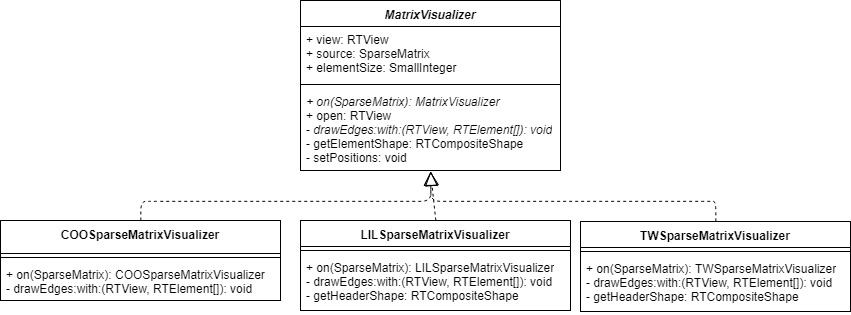


Рисунок 10. Ієрархія класів візуалізаторів

Для перегляду результату візуалізації використаємо наступну матрицю:

Для матриці на основі списку координат відображення має наступний вигляд (Рис. 11):

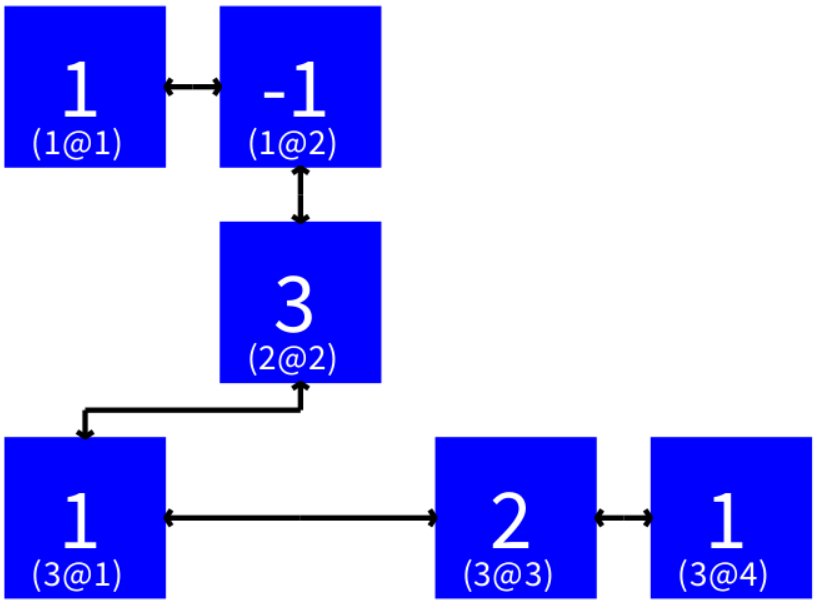


Рисунок 11. Відображення для COOSparseMatrix

Для матриці на основі колекції списків відображення наступне (Рис. 12):

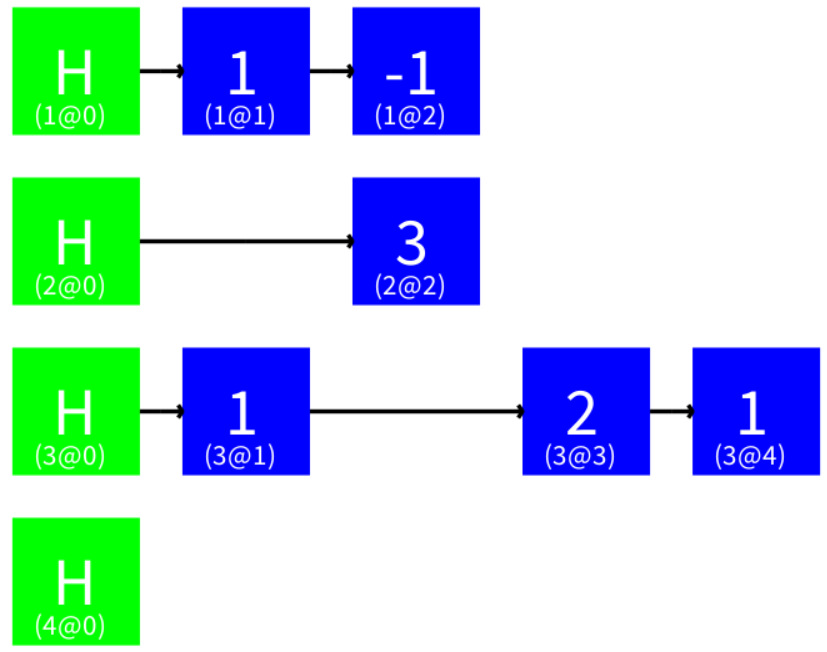


Рисунок 12. Відображення для LILSparseMatrix

Тут основна різниця з попереднім відображенням полягає в тому, що ця матриця містить елементи, які є заголовками, а відповідно не є реальною частиною змісту матриці. Саме тому колір прямокутника для заголовка буде зеленим, а мітка значення міститиме підпис ‘H` (Header).

Для матриці на основі мережі списків відображення наступне (Рис. 13):

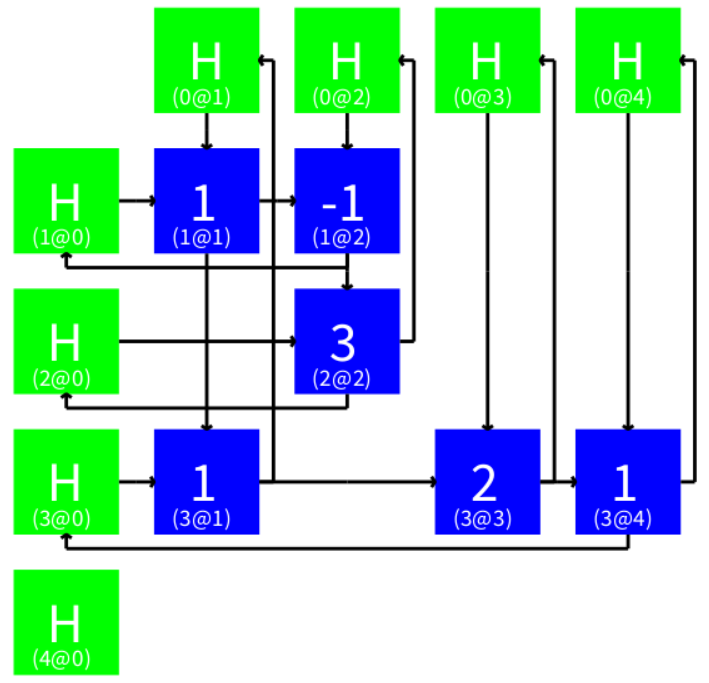


Рисунок 13. Відображення для TWSparseMatrix

Ця реалізація матриці також містить заголовки списків рядків і стовпців, тому вони мають аналогічний вигляд до заголовків у відображення матриці LILSparseMatrix.

Кожна матриця, відповідно до внутрішньої структури, має окремо визначені методи для виконання операцій. В кожної реалізації присутні свої особливості, які, в свою чергу, можуть спрощувати або ускладнювати як розуміння вже існуючого коду, так і написання нового (для реалізації нових операцій).

**Об’єкт дослідження:** графічна репрезентація операцій над розрідженими матрицями на основі списку координат, колекції списків, мережі списків.

**Предмет дослідження**: графічна репрезентація операцій над розрідженими матрицями.

**Мета дослідження**: спроектувати, реалізувати, випробувати і описати програмні засоби, які дають змогу відобразити графічно операції, такі як додавання, множення та транспонування, для розроблених раніше розріджених матриць.

У відповідності з метою дослідження ставляться такі **завдання**:

1. Спроектувати та реалізувати програмні засоби для графічного відображення операцій над розробленими раніше розрідженими матрицями з використанням результатів попереднього дослідження;
2. Випробувати і описати ці програмні засоби.

**Наукова новизна дослідження** полягає у тому, що засобів для відображення операцій розріджених матриць у бібліотеках Pharo Smalltalk немає, а їх наявність покращить вивчення цих структур і алгоритмів, що над ними діють, зокрема додавання, множення тощо.

# Формулювання завдання

Реалізувати програмні засоби для відображення операцій над розрідженими матрицями засобами середовища Pharo Smalltalk 7.0.3.

Візуалізація операцій над цими структурами передбачає відображення на екрані візуалізацій матриць, над якими відбувається операція, тобто однієї матриці для унарних операцій, двох – для бінарних операцій, відповідної результуючої матриці, а також допоміжних елементів, які використовуються під час обчислень. Допоміжними елементами можуть виступати тимчасові масиви, в яких зберігаються проміжні результати, інше забарвлення елементів початкових матриць, які розглядаються в певний момент часу, та допоміжні текстові вказівки.

Мета завдання полягає в тому, щоб спроектувати і реалізувати пакет, який дозволить виконати наступне:

* надасть змогу візуалізувати будь-яку зв’язну структуру і операції над нею з мінімальними доповненнями до існуючого коду цієї структури;
* допомогти людям, не знайомим з розрідженими матрицями на основі зв’язних списків чи вузлів, зрозуміти їх внутрішню реалізацію та алгоритми операцій над ними;
* допомогти в пошуку помилок в коді за допомогою наглядного спостереження на його виконанням;
* налагоджувати і оптимізувати існуючі алгоритми.

# Огляд засобів Pharo

**Pharo** — це сучасна повнофункціональна реалізація середовища мовою Smalltalk з відкритим вихідним кодом.

Для реалізації нам потрібна віртуальна машина Pharo та образ середовища Pharo. Їх можна завантажити з офіційного сайту Pharo за посиланням: <http://pharo.org/download>. Ми будемо використовувати останню версію Pharo 7.0.3 станом на травень 2019 року.

## 3.1. Основні інструменти середовища

В порівнянні з версією 6.1, яка була використана в попередній роботі, більшість основних інструментів середовища залишилися незмінними. Ми розглянемо лише деякі з них. З рештою ви можете ознайомитися в попередній роботі.

Новим системним переглядачем середовища, який прийшов на заміну переглядачу ***Nautilus*** є ***Calypso*** (Рис. 14).Цей інструмент дозволяє переглянути чи змінити будь-який клас чи метод, що знаходиться в образі. Відповідно, можна написати власні структури або переписати вже існуючі.

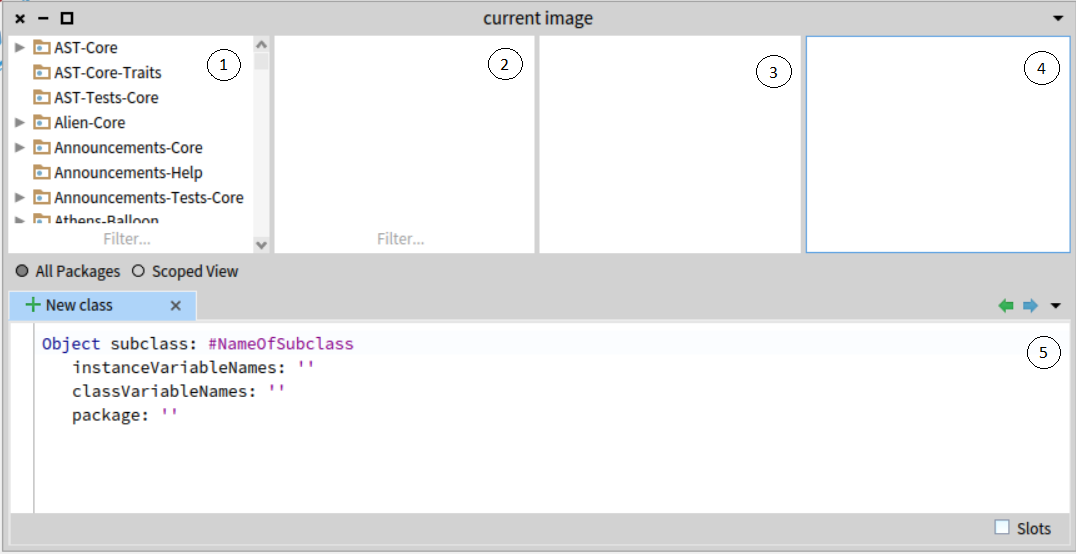


Рисунок 14. Переглядач Calypso

Вікно Calypso складається з чотирьох вікон у верхній частині, які відображають ієрархічну модель методів у системі. Панель 1 містить усі пакети класів в образі. На панелі 2 відображені класи, що знаходяться в обраному пакеті. Якщо обрати один з класів пакету, панель 3 міститиме усі протоколи методів для цього класу. Протоколи – це спосіб поділу методів в класі на логічні категорії. Четверта панель відображає імена всіх методів, що належать до певного протоколу. Обравши певний метод, його вихідний код відображатиметься на панелі 5.

**Iceberg** – це набір інструментів, які дозволяють взаємодіяти з репозиторіями git безпосередньо з образу Pharo. Iceberg полегшує контроль версій і має інтуїтивно зрозумілий інтерфейс (Рис. 15). У версії Pharo 7.0.3 його основну панель значно спростили та довершили підтримку зовнішніх систем контролю версій.

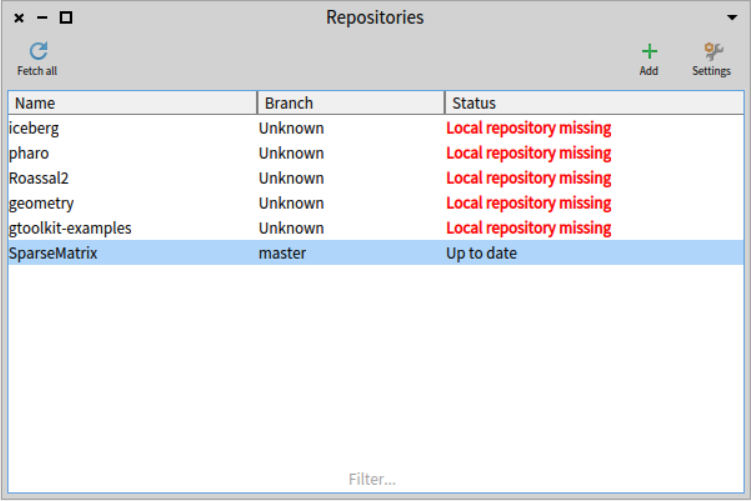


Рисунок 15. Вікно Iceberg

## 3.2. Візуалізаційний засіб Roassal

**Roassal** – це гнучкий програмний засіб, створений для візуалізації та взаємодії з довільними даними, визначеними на основі об'єктів і зв'язків між ними. Зазвичай він використовується для створення інтерактивної візуалізації, а діапазон його застосування різноманітний. Наприклад, спільнота Moose використовує Roassal для візуалізації програмного забезпечення.

З структурною моделлю цього засобу ви можете ознайомитися в попередній роботі. Ми розглянемо лише нові засоби, які будуть використані при реалізації візуалізації операцій та доповнення до них.

### 3.2.1. Композиція відображень

В пакеті Roassal передбачено клас **RTComposer**, за допомогою якого можна створювати композиції відображень (Рис. 16).

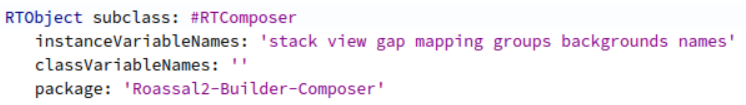


Рисунок 16. Клас RTComposer

Екземпляр цього класу містить наступні змінні:

* view – екземпляр класу RTView: контейнер для елементів композиції;
* gap – проміжок між групами елементів;
* mapping – екземпляр класу Dictionary, який містить групу елементів контейнера за певним ключем;
* groups – ключі груп елементів композиції;
* backgrounds – масив з посиланнями на елементи, які виступають фоном груп в контейнері;
* names – масив з посиланнями на елементи, які виступають назвами груп в контейнері.

Розглянемо приклад створення композиції двох графіків, які відображають певне відношення сили повороту ключа до крутного моменту. Запишемо наступний код:



Результатом його виконання буде відображення на Рис. 17.

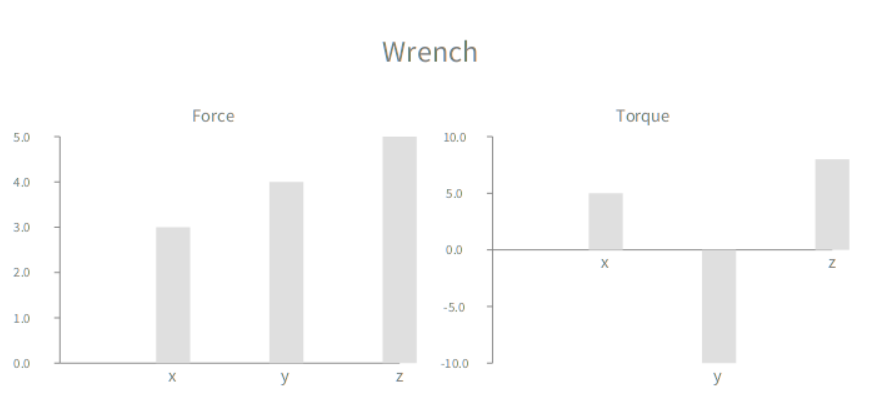


Рисунок 17. Композиція відображень за допомогою RTComposer

Рядки 1-8 задають вхідні дані для діаграм. Рядок 9 створює новий об’єкт типу **RTComposer**. Рядок 12 створює новий об’єкт **RTGrapher**, за допомогою якого буде створюватися одна з діаграм. Рядок 13 задає контейнер **RTView** нашого графіку як **RTView** нашого **RTComposer** об’єкту. Рядки 14-22 задають відображення діаграми, а рядок 23 додає елементи діаграми в контейнер. Рядок 24 групує щойно додані об’єкти першої діаграми в групу з ключем **#force**. Друга діаграма додається аналогічно рядками 27-39.

Рядок 42 задає розташування групи елементів першого графіка відносно другого графіка так, що перший графік буде знаходитись зліва від другого. В рядках 43-45 ми задаємо назви для кожного графіка і загальну назву їх композиції відповідно.

Цей приклад дозволяє зрозуміти, що відображення різних об’єктів можна компонувати на одному екрані, а також налаштовувати їх розташування одне відносно одного, що чудово підходить для нашого завдання.

### 3.2.2. Доповнення Roassal

Для зручного відображення та оновлення відображень необхідних нам елементів візуалізатора ми доповнили деякі класи, а саме:

1. **RTShapedObject** – абстрактний клас, який виступає базовим класом для RTElement (елемент відображення) та RTEdge (зв’язок між елементами). До нього було додано метод updateModelAndRedraw:, який приймає нову модель елемента, записує її як модель елемента і оновлює відображення елемента на екрані відповідно до оновлення.
2. **RTComposer** – додано наступні методи:
   1. groupToExisting: - метод, який додає нові елементи на екрані, які ще не були згруповані, до групи, ключ якої передається як аргумент;
   2. hasGroup: - метод, який відповідає чи існує група з ключем, який передається як аргумент, на відображенні;
   3. move:between:and: - метод, який приймає три ключі груп і розташовує групу передану як аргумент до move: посередині між групами переданими between: та and:
   4. redrawBackgroundsFor: - приймає масив ключів груп, для яких потрібно перемалювати фон;
   5. redrawNamesFor: - приймає масив ключів груп, для яких потрібно перемалювати імена;
   6. replaceGroup: groupName with: elements – замінює елементи в групі groupName на масив елементів elements;
   7. setColor: color using: block – змінює колір елементів свого контейнера, для яких block відповідає true, на color;
   8. setColor: color forModel: model – змінює колір елементів свого контейнера чия модель дорівнює model на color.

## 3.3. Пакет Reflectivity

Smalltalk – рефлексивна мова програмування. Це означає, що програми здатні «відобразити» своє власне виконання та структуру. З програмної точки зору це означає, що метаоб’єкти системи виконання можна змінювати як звичайні об’єкти, які можна зчитувати і перевіряти. Метаоб’єкти в Smalltalk – це класи, метакласи, словники методів, скомпільовані методи, стек часу виконання і т.д. Така форма рефлексії називається інтроспекцією і підтримується багатьма сучасними мовами програмування.

І навпаки, у Smalltalk можна модифікувати метаоб’єкти та відобразити ці зміни назад у систему виконання. Це також називається заступництвом (або посередництвом) і підтримується головним чином мовами динамічного програмування і лише в дуже обмеженій мірі статичними мовами.

Програма, яка здатна маніпулювати іншими програмами (або навіть собою) називається метапрограмою. Щоб мова програмування була рефлексивною, вона повинна підтримувати як інтроспекцію, так і заступництво. Інтроспекція – можливість вивчити структури даних, які визначають мову, такі як об’єкти, класи, методи і стек виконання. Заступництво – це здатність модифікувати ці структури, тобто змінити семантику мови і поведінку програми з самої програми. Структурна рефлексія стосується вивчення та модифікації структур системи виконання, а поведінкова рефлексія – зміни інтерпретації цих структур.

Reflectivity - це пакет, який розширює звичайні можливості рефлексії Pharo. Він містить класи, за допомогою яких можна «прив’язатися» до будь-якого вузла абстрактного синтаксичного дерева. Іншими словами – він дозволяє виконувати довільний код при виконанні певного елемента коду: методу, змінної, змінної класу тощо. Найбільш уживаним класом з цього пакету є MetaLink.

### 3.3.1. Клас MetaLink

Клас MetaLink пакету Reflectivity – це одна з основних складових цілої системи, яка розширює можливості рефлексії в Pharo. За допомогою цього класу та інших допоміжних компонент пакету можна додавати анотації до вузлів абстрактного синтаксичного дерева. Анотований вузол розкривається, компілюється і виконується на льоту.

Для певних вузлів метапосилання можна встановити до, після чи замість виконання цього вузла. Також існує можливість прив’язки до успішного чи помилкового виконання вузла, але не для всіх вузлів доступні такі анотації. Наприклад, для літералів не можна додати метапосилання на успішне чи неуспішне виконання.

### 3.3.2. Приклади створення метапосилань

В загальному випадку алгоритм створення і прив’язки метапосилань наступний:

1. Оголосити новий об’єкт типу MetaLink;
2. Налаштувати необхідні опції для анотації, зокрема: метаоб’єкт, повідомлення для метаоб’єкта, позиція виконання анотації (до, після чи замість) тощо;
3. Викликати метод link: metalink toAST: ast об’єкта до якого відбувається прив’язка, де metalink – це наш налаштований об’єкт MetaLink, а ast – вузол об’єкта прив’язки, для якого потрібна анотація.

Для зручності були створені додаткові методи, які є розширенням класу Object, для прив’язки до різних вузлів об’єкта. Тут параметр option – це #read, #write або #both, які анотують зчитування, запис або читання та запис вузла відповідно. З них найцікавіші:

* link: aMetaLink toClassVariableNamed: aClassVariableName option: option – для прив’язки до змінної класу aClassVariableName
* link: aMetaLink toMethodNamed: aMethodName – для прив’язки до методу aMethodName
* link: aMetaLink toSlotNamed: aSlotName option: option – для прив’язки до змінної екземпляра класу aSlotName
* link: aMetaLink toTemporaryNamed: aTempVarName inMethod: aMethodName option: option – для прив’язки до тимчасової змінної aTempVarName в методі aMethodName

Розглянемо приклад створення анотації для запису змінної класу. Нехай існує клас ReflectivityExamples2 (Рис. 18). Він містить змінну класу з іменем classVar.

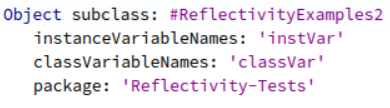


Рисунок 18. Клас ReflectivityExamples2

Виконаємо такий код:



Рядки 2-4 створюють і налаштовують нове метапосилання наступним чином: об’єкту, який ми передали в metaObject: (звичайний вивід дружнього повідомлення в Transcript) буде надіслано повідомлення передане в selector: після виконання вузла, до якого буде додано це посилання. Рядок 6 створює новий об’єкт instance класу ReflectivityExamples2. Рядки 7-10 використовують метод link:toClassVariableNamed:option: для прив’язки нашого метапосилання до запису в змінну класу classVar. Рядки 12-15 демонструють можливість додавання анотації лише до вузлів певного екземпляра класу ReflectivityExamples2.

Інакше кажучи, при першому прикладі встановлення анотації вивід в Transcript буде відбуватися кожного разу, коли хтось буде записувати нове значення в змінну класу classVar. У другому прикладі – вивід відбудеться лише при умові, що запис у змінну буде здійснюватися екземпляром класу instance.

### Видалення метапосилань

Оскільки анотації для вузлів можна встановлювати на невизначений термін, а об’єкти можливо будуть потрібні в майбутньому без доданих метапосилань, необхідний механізм їх видалення. Для цього існує зручний метод uninstall класу MetaLink.

Видалення передбачає вилучення анотацій з усіх вузлів, до яких був прив’язаний той екземпляр метапосилання, який обробляє метод uninstall. Для окремого відв’язування передбачений метод unlinkFromNode: aNode forObject: anObject, який вилучає анотацію з вузла aNode об’єкта anObject, залишаючи решту анотацій неушкодженими.

# 4. Реалізація

Нові класи візуалізаторів додамо до вже існуючого тегу пакету LNU–SparseMatrix під назвою Visual (Рис. 19), в якому будемо оголошувати класи для візуалізації операцій.

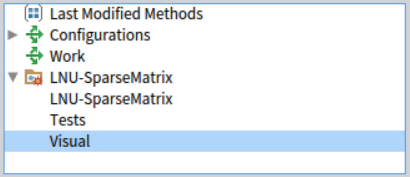


Рисунок 19. Тег Visual в пакеті LNU-SparseMatrix

Візуалізатори операцій міститимуть деяку спільну поведінку, яку варто винести в окремий узагальнений клас. Вони наслідуватимуть клас MatrixOperationVisualizer (Рис. 20), який буде абстракцією візуалізатора операцій для розріджених матриць. MatrixOperationVisualizer, в свою чергу, наслідуватиме клас Object, кореневий клас будь-якого об’єкту системи Pharo.

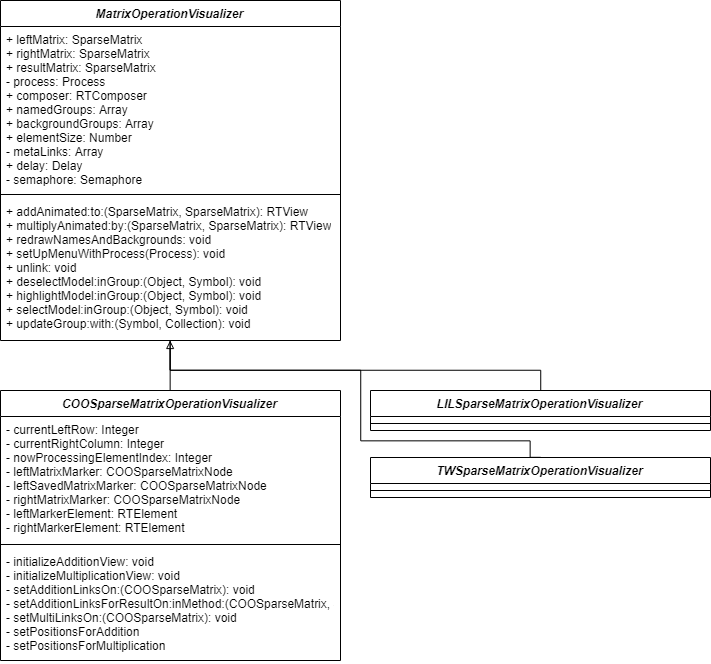


Рисунок 20. Клас MatrixOperationVisualizer

MatrixOperationVisualizer – клас абстракції візуалізаторів операцій над розрідженими матрицями, який містить:

* leftMatrix – посилання на матрицю, яка є матрицею зліва в бінарних операціях, або матрицею операції в унарних;
* rightMatrix – посилання на матрицю, яка є матрицею справа в бінарних операціях;
* resultMatrix – посилання на матрицю, яка є результуючою матрицею операції;
* process – процес, який виконує операцію;
* composer – композиційний об’єкт, за допомогою якого буде складатися відображення;
* namedGroups – список груп композиції, для яких визначено надписи імен;
* backgroundGroups – список груп композиції, для яких промальовується фон;
* розмір відображення елемента (elementSize) – ціле число (екземпляр класу Integer);
* metaLinks – список метапосилань, які були прив’язані до матриць, задіяних під час операції (для правильного вилучення анотацій);
* delay – затримка при виконанні певного кроку операції;
* semaphore – об’єкт Semaphore для синхронізації процесів відображення і самої операції.

Визначені наступні методи:

* addAnimated:to – метод, який отримує дві матриці для додавання і повертає об’єкт RTView з ініціалізованою композицією для відображення операції їх додавання;
* multiplyAnimated:by: - метод, який отримує дві матриці для множення і повертає об’єкт RTView з ініціалізованою композицією для відображення операції їх множення;
* redrawNamesAndBackgrounds – приватний метод, який використовується для зручного перемальовування імен і фону груп, які були записані в змінні namedGroups і backgroundGroups при ініціалізації візуалізатора;
* setUpMenuWithProcess: – приватний метод, який додає до відображення кнопки для контролю виконання операції (призупинення і відновлення);
* unlink – приватний метод, який відв’язує усі метапосилання в metaLinks;
* deselectModel:inGroup: - метод, який змінює колір елемента групи з inGroup: моделі, яку передано в deselectModel:, на колір за замовчуванням для цього елемента;
* highlightModel:inGroup: - метод, який змінює колір елемента групи з inGroup: моделі, яку передано в highlightModel:, на колір виділення highlightColor, який було задано при ініціалізації візуалізатора (жовтий за замовчуванням);
* selectModel:inGroup: - метод, який змінює колір елемента групи з inGroup: моделі, яку передано в selectModel:, на колір виділення selectColor, який було задано при ініціалізації візуалізатора (червоний за замовчуванням);
* updateGroup:with: - видаляє попередні елементи групи з updateGroup: і замінює їх елементами з with:

Розглянемо детальніше метод selectModel:inGroup: (Рис. 21):

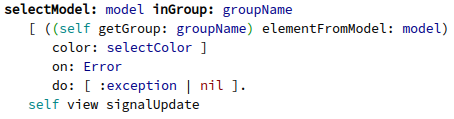


Рисунок 21. Метод selectModel:inGroup:

Цей метод отримує ім’я групи groupName композиції, в якій очікується елемент, прообразом якого є модель model. Використовуючи метод elementFromModel: об’єкта RTGroup, який ми отримуємо в результаті повідомлення getGroup:, ми отримуємо посилання на шуканий елемент візуалізатора. Йому ми задаємо колір selectColor. Якщо елемента для такої моделі в заданій групі не буде знайдено, то відображення на екрані не зміниться.

Після цих змін ми надсилаємо повідомлення signalUpdate до візуалізатора, щоб зміни відобразилися на екрані.

Метод updateGroup:with: (Рис. 22):

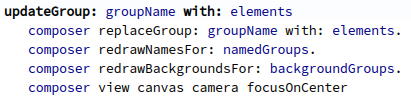


Рисунок 22. Метод updateGroup:with:

Цей метод використовує наше розширення RTComposer – replaceGroup:with:, для заміни елементів групи, перемальовує імена і фон для відповідних груп візуалізатора, а тоді центрує і масштабує відображення так, щоб на екрані було видно всі елементи.

Метод setUpMenuWithProcess: (Рис. 23):



Рисунок 23. Метод setUpMenuWithProcess

Оскільки нам необхідно мати певний контроль над процесом виконання операції при її відображенні, то цей метод додає меню, яке буде розташоване в лівому верхньому куті відображення, і буде мати кнопки Pause і Resume для призупинення і відновлення операції відповідно. Тут ми використовуємо методи suspend і resume об’єкта Process, який виконує візуалізовану операцію в іншому потоці.

Метод addAnimated:to: (Рис. 24)

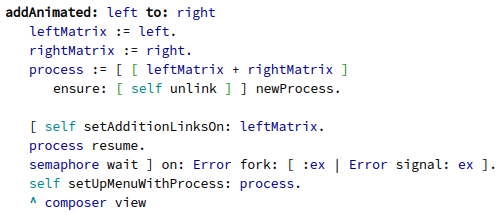


Рисунок 24. Метод addAnimated:to:

Цей метод приймає матриці, додавання яких ми хочемо відобразити, і повертає об’єкт RTView цього відображення.

Спершу ми зберігаємо посилання на об’єкти матриць, задіяних в операції. Тоді, ми створюємо новий процес process, який буде виконувати саме додавання, і обов’язково видалить анотації до матриць після закінчення виконання операції.

Оскільки деяких об’єктів, які нам потрібно відобразити, не існує в пам’яті до початку виконання операції, то ми використовуємо об’єкт Semaphore для того, щоб спершу виконати частину операції до моменту, коли у нас будуть посилання на всі необхідні об’єкти для відображення, і призупинити її. Таким чином, ми даємо можливість користувачу самому почати фактичне виконання додавання, але маючи композицію відображень усіх елементів, які вже ініціалізовані в методі.

Отримання посилань на об’єкти відбувається за допомогою анотацій, які ми додаємо до виконання операції. Призупинення процесу виконання операції також здійснюється анотаціями. Метод setAdditionLinksOn: - абстрактний метод, який реалізований в кожному візуалізаторі залежно від алгоритму додавання певної структури матриці.

Метод multiplyAnimated:by: реалізований аналогічно.

## 4.1. Візуалізатори

Візуалізатори операцій для кожного типу розрідженої матриці мають спільну поведінку, яка визначена в батьківському класі MatrixOperationVisualizer. Проте деякі речі, зокрема композиція відображення для певних операцій, кількість і тип допоміжних елементів під час їх виконання, специфіку додавання анотацій до об’єктів методів, необхідно визначити для кожного візуалізатора окремо, в силу різниці внутрішньої структури матриць та особливості реалізації операцій над ними.

Ієрархія класів наступна (Рис. 25):

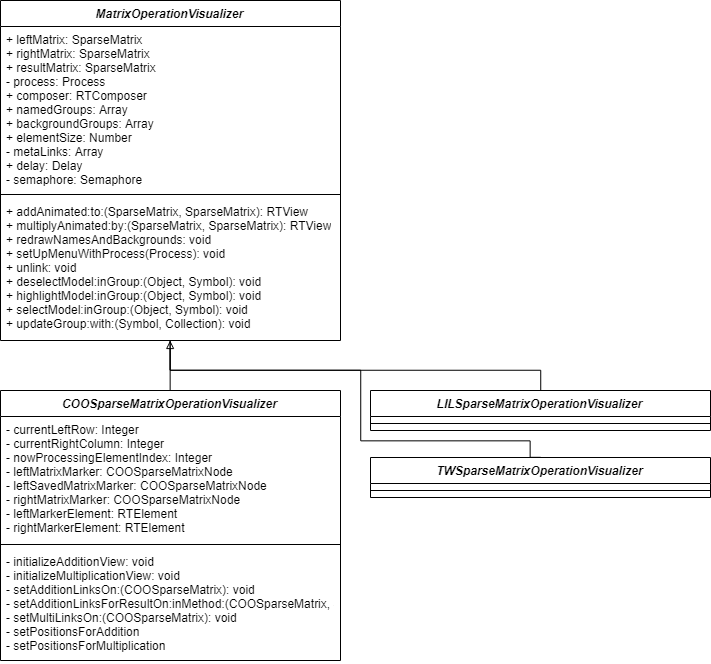


Рисунок 25. Ієрархія візуалізаторів операцій

### 4.1.1. Візуалізатор операцій над матрицями на основі списку координат

Оскільки загальна логіка візуалізації операцій вже реалізована в батьківському класі, нам потрібно реалізувати методи для додавання анотацій до відповідних вузлів операцій.

Повний код методу setAdditionLinksOn: для класу COOSparseMatrixOperationVisualizer наведено в Додатку А. Розглянемо деякі його логічні частини.

Метод setAdditionLinksForResultOn:inMethod: в рядку 5 додає анотації до результуючої матриці таким чином:

1. Додається анотація до першого елемента результату, яка буде оновлювати відображення результату при записі в нього;
2. Додається специфічна анотація до останнього елемента, яка виконає наступні дії при першому записі в останній елемент результату:
   1. Оновить відображення результату;
   2. Видалить сама себе;
   3. Додасть нову анотацію до останнього елемента результату, яка буде оновлювати відображення результату при записі в нього;
   4. Посилає сигнал за допомогою об’єкта semaphore до батьківського методу (addAnimated:to:) для синхронізації;
   5. Призупиняє процес виконання операції.

В рядках 7-119 метод setAdditionLinksOn: додає анотації до лівого (leftMarkerLinkBefore, leftMarkerLinkAfter) і правого (rightMarkerLinkBefore, rightMarkerLinkAfter) маркерів операції, які будуть підсвічувати елементи на відображенні, які в певний момент часу задіяні в операції, а також оновлювати допоміжні елементи на відображенні. Особливість тут полягає в такому:

1. При першому записі в тимчасову змінну маркера всередині операції ми зберігаємо посилання на цей маркер (рядки 25 та 73), підсвічуємо його на відображенні (рядки 42-45 та 108-111) та ініціалізуємо допоміжні елементи, пов’язані з цим вузлом (рядки 38-40 та 105-107);
2. При наступних записах в змінну маркера всередині операції ми лише оновлюємо допоміжні елементи (рядки 44 та 110), оновлюємо посилання на цю змінну (рядки 25 та 73) і додаємо нову анотацію, яка підсвітить цей елемент іншим кольором, якщо він буде обраний для додавання до результату, і сама ж видалиться після спрацьовування (рядки 29-36 та 77-102);
3. Перед записом в змінну маркера ми повертаємо колір елемента за замовчуванням (рядки 11 та 59).

Метапосилання, які додає метод, записуються в масив metaLinks (рядки 120-125), який використовується при видаленні анотацій після закінчення операції. Також, в кінці кожного спрацьовування анотацій підсвічування, вибору або оновлення результату спрацьовує затримка потоку, який виконує операцію, за допомогою об’єкта Delay (рядки 32, 45, 90, 98 та 111). Перед викликом затримки використаний метод copy, оскільки один і той же об’єкт Delay може бути викликаний з різних потоків, що спричинить помилку.

Метод setMultiLinksOn: реалізований аналогічно, з врахуванням певних додаткових елементів при виконанні множення матриць, а саме: індексу шуканого елемента і його значення на певному кроці алгоритму. Індекс шуканого елемента залежить від тимчасових змінних currentLeftRow і currentRightColumn, а його значення від змінної sum.

Реалізувати транспонування схожим чином не було можливим в силу відсутності однозначної інтерпретації звернення до певних вузлів на певних кроках алгоритму. Тому це відображення реалізовано розширенням класу матриці COOSparseMatrix і її методу transpose.

Метод transposeAnimatedWithDelay: приймає затримку в секундах, яка буде між операціями, і повертає об’єкт RTView з відображенням транспозиції. Реалізація відображення аналогічна з додаванням і множенням, за умови, що ми не додаємо анотацій до вузлів, а прописуємо відповідні дії в коді алгоритму.

### 4.1.2. Візуалізатор операцій над матрицями на основі колекції списків

Оскільки без фундаментальної модифікації коду операцій матриці LILSparseMatrix було неможливо створити відображення, то візуалізація операцій цієї матриці була виконана в розширених методах самої матриці. Для зручного доступу до композиції був використаний клас LILSparseMatrixOperationVisualizer.

Метод addAnimated:delay: приймає правий доданок-матрицю і затримку. Для зручної ініціалізації відображення клас MatrixOperationVisualizer був розширений методом getComposedViewOf:with:result:, який приймає об’єкти двох матриць бінарної операції та об’єкт результуючої матриці, додає їх до відображення і задає розташування.

Візуалізатор матриць типу LILSparseMatrix відображає об’єкти-посередники елементів, а не самі елементи матриць. Для того, щоб мати можливість виділяти об’єкти на відображенні, маючи посилання на їх прообраз, до екземплярів класу посередників MatrixElement додамо змінну source – посилання на елемент матриці. Для зручного виділення клас MatrixOperationVisualizer розширимо методами selectUsing:, deselectUsing: і highlightUsing:, які приймають блок коду для фільтрації елементів і змінюють колір для тих, для кого фільтр дає результат true.

Аналогічним чином реалізований метод byMatrixAnimated:delay: для відображення множення матриць і метод transposeAnimatedDelay: транспонування.

Особливістю операції множення цієї структури є використання проміжних зв’язних списків для формування нового рядка результуючої матриці. Для їх відображення клас LILSparseMatrixVisualizer був розширений методом onLinkedList:inRow:withView:, який приймає список типу LinkedList, індекс рядка, куди цей список буде записаний, і об’єкт RTView, до якого потрібно додавати відображення цього списку.

### 4.1.3. Візуалізатор операцій над матрицями на основі мережі списків

Оскільки без фундаментальної модифікації коду операцій матриці TWSparseMatrix було неможливо створити відображення, то візуалізація операцій цієї матриці була виконана в розширених методах самої матриці. Для зручного доступу до композиції був використаний клас TWSparseMatrixOperationVisualizer.

Метод addAnimated:delay: приймає правий доданок-матрицю і затримку. При ініціалізації відображень використаний метод getComposedViewOf:with:result:

Алгоритм додавання для цієї структури спочатку переглядає рядки матриць на наявність двох порожніх рядків з тим же індексом. Таким чином порожній рядок навіть не розглядається алгоритмом і відповідно залишається порожнім в результуючій матриці. Для того, щоб відобразити цю логіку, додамо затемнення рядків матриці, які будуть ігноруватися.

Аналогічним чином реалізований метод byMatrixAnimated:delay: для відображення множення матриць і метод transposeAnimatedDelay: транспонування. При множенні алгоритм ігнорує порожні рядки лівої матриці і порожні стовпці правої матриці, таким чином виключаючи кількість кроків для виконання алгоритму. Ми відповідно затемнюємо рядки і стовпці на відображенні.

# 5. Аналіз отриманих результатів

Для перегляду роботи класів візуалізації використаємо такі матриці:

і

Представимо їх в середовищі Pharo так:



Створимо на основі цих колекцій по матриці кожного типу: списку координат, колекції списків і мережі списків:



Відобразимо наступні операції за допомогою команди “Do it and go”:



Додавання матриць на основі списку координат має такий вигляд (Рис. 26, 27):

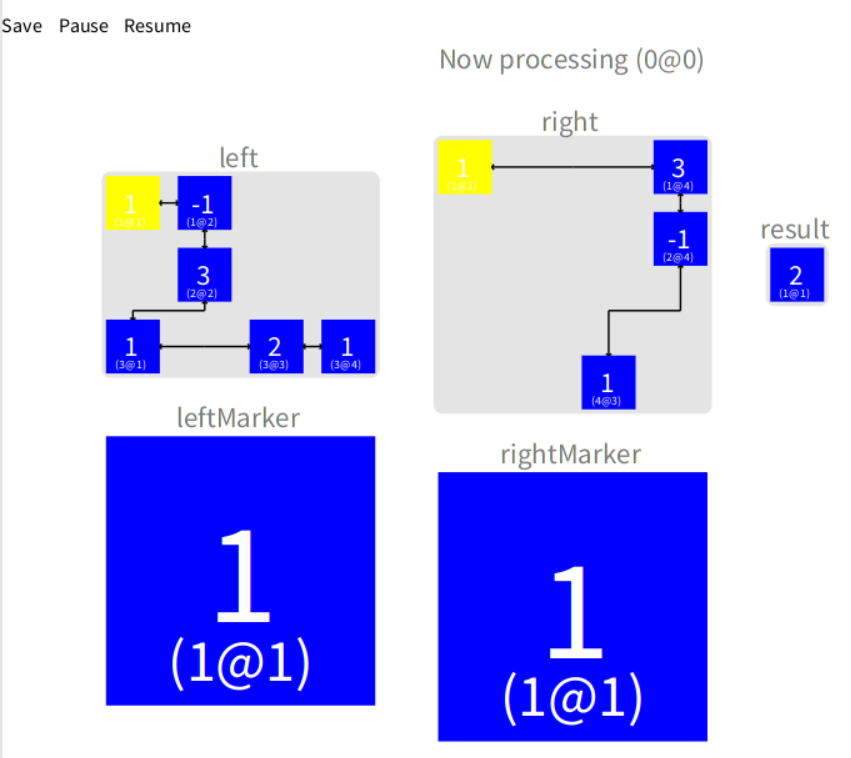


Рисунок 26. Перший крок додавання

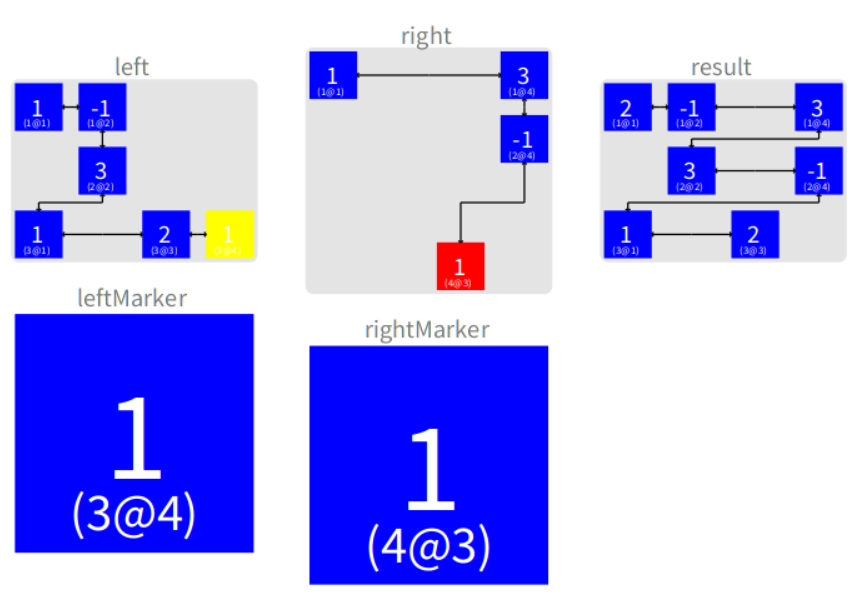


Рисунок 27. Один із кроків додавання. Жовтий елемент додається до результату

Додавання матриць на основі колекції списків (Рис. 28):

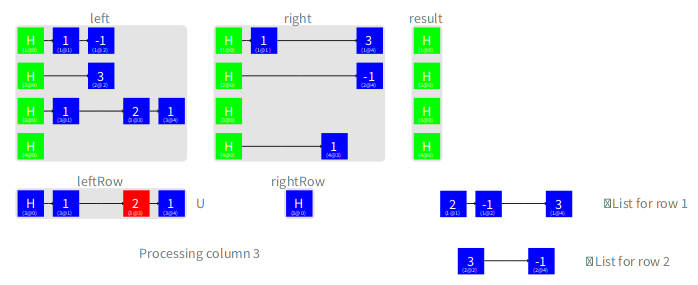


Рисунок 28. Формування третього рядка результуючої матриці.

Тут під лівою і правою матрицею розглядаються відповідні рядки, а під результуючою відображені списки – попередні рядки результату, які будуть записані в колекцію результату наприкінці операції.

Множення матриць на основі колекції списків (Рис. 29, 30):

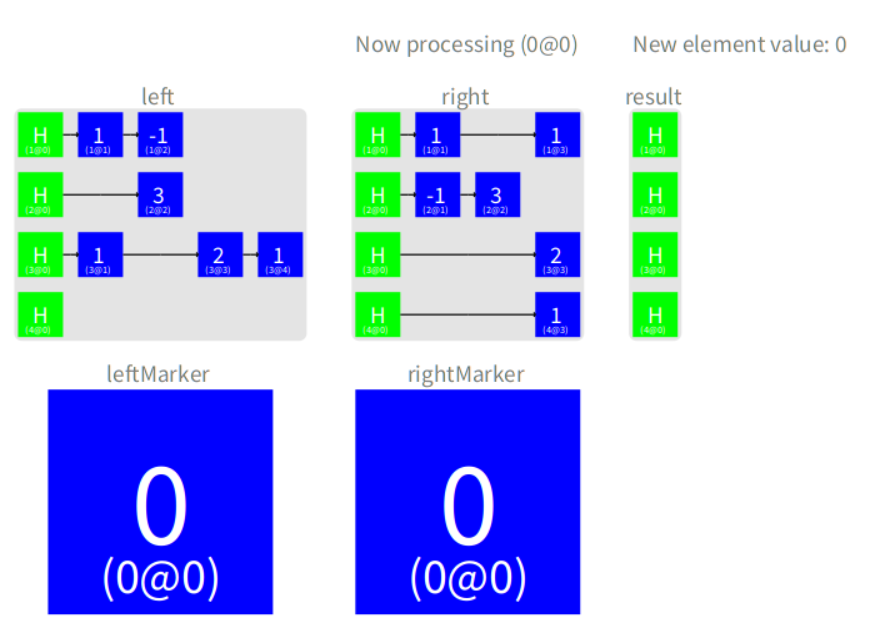


Рисунок 29. Щойно ініціалізоване відображення множення.

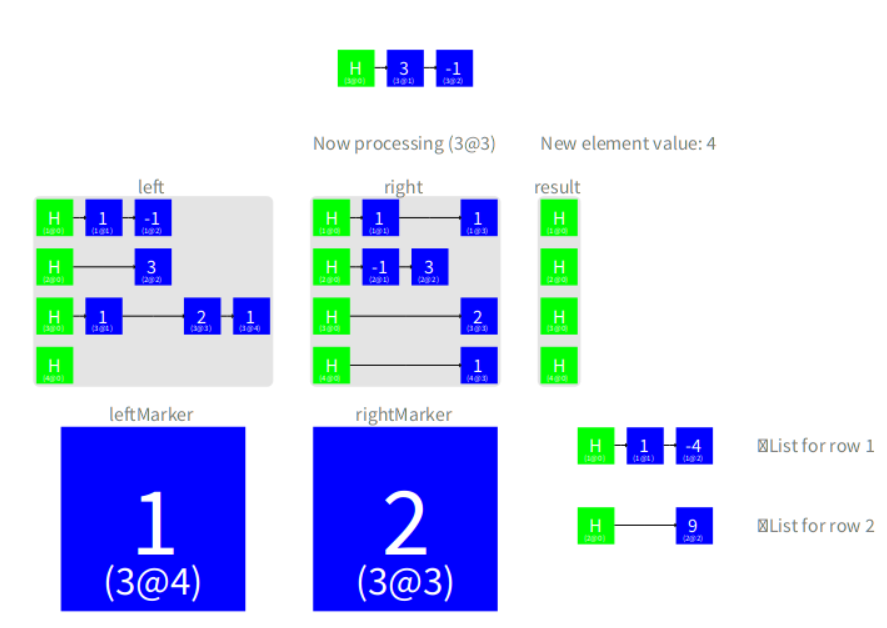


Рисунок 30. Крок множення: в список третього рядка результуючої матриці додається елемент з індексом (3, 3) і значенням 4

Множення матриць на основі мережі списків (Рис. 31, 32):

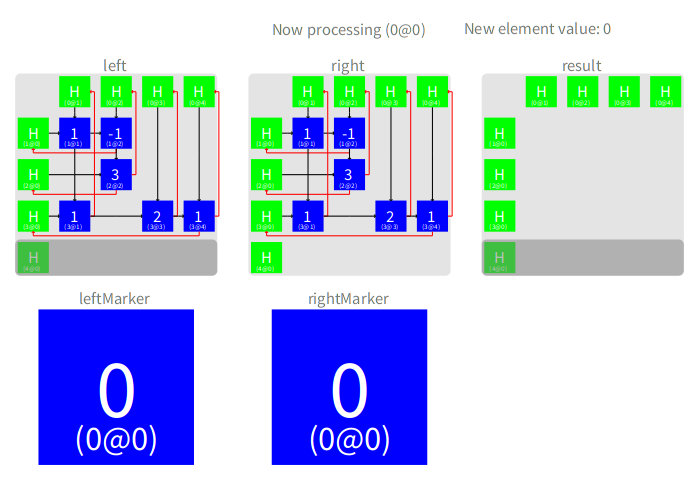


Рисунок 31. Щойно ініціалізоване відображення множення. Рядки, які зафарбовані сірим, ігноруються.

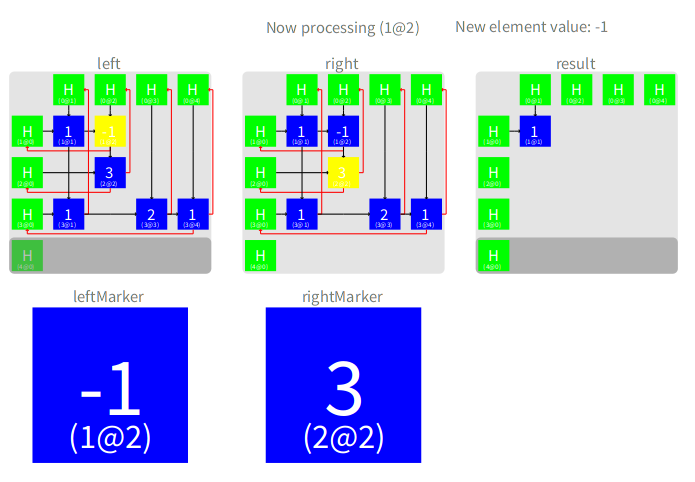


Рисунок 32. Крок множення: обчислюємо елемент (1, 2). Елементи, виділені жовтим, перемножуються, і результат додається до значення нового елемента, яке зараз становить -1.

Транспонування матриці на основі мережі списків (Рис. 33):

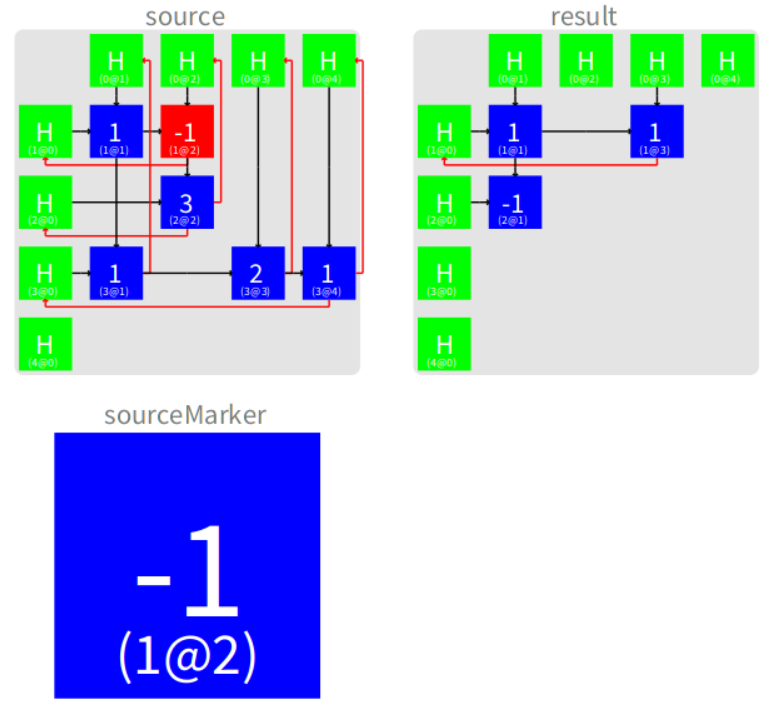


Рисунок 33. Розглядається елемент з індексом (1, 2), який щойно додали до результату як елемент (2, 1)

Транспонування матриці на основі списку координат (Рис. 34):

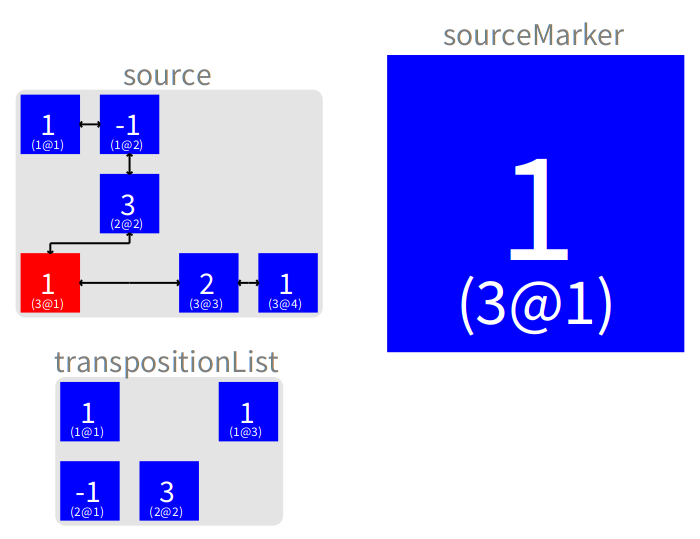


Рисунок 34. Елемент (3, 1) додається в результат як елемент (1, 3). Для елементів результуючої матриці ще не задані зв’язки.

Як можна бачити, візуалізатори коректно відобразили операції залежно від алгоритму реалізації для кожної структури. При потребі можна масштабувати відображення за допомогою кнопок  розташованих на панелі інструментів. При цьому можна призупинити виконання операції (Рис. 27, 28, 30, 32, 33, 34) і зберегти відображення на диск як зображення або HTML-файл за допомогою кнопки і елемента меню ‘Export as:’. Більше того, всі елементи відображення інтерактивні, тобто при натисканні на елемент відкриється його модель – відповідний елемент матриці.

# 6. Висновки

В ході виконання роботи я:

* поглибив знання про середовище розробки Pharo та закріпив навички роботи зі стандартною бібліотекою Pharo Smalltalk 7.0.3;
* поглибив знання візуалізаційного засобу Roassal та розширив його;
* реалізував програмні засоби для візуалізації операцій додавання, множення і транспонування над розрідженими матрицями на основі внутрішнього представлення «список координат», «колекція списків» і «мережа списків»;
* перевірив правильність їх роботи.

Внаслідок аналізу виявив, що:

* візуалізація алгоритмів, які виконують складні операції над списковими структурами, полегшує їх вивчення, розуміння і спрощує пошук помилок;
* інтерактивність відображення дозволяє швидко і ефективно проводити паралель між об’єктом відображення (матрицею) і його графічним представленням.

В майбутньому я планую:

* модифікувати код операцій матриць, для яких неможливо було винести логіку відображення в інший клас;
* покращити зворотній зв’язок між відображенням матриці і об’єктом матриці, а саме візуалізувати зміни в моделях матриць одразу після їх внесення.

# Список використаної літератури

1. Bergel Alexandre. Agile Visualization: Roassal [Онлайновий]. - http://agilevisualization.com/AgileVisualization/Roassal/0104-Roassal.html.
2. **Bergel Alexadre** Agile Visualization First Edition [Книга]. - [місце видання невідоме] : Alexandre Bergel, 2016.
3. **Bergel Alexandre [та ін.]** Deep into Pharo [Книга]. - [місце видання невідоме] : Square Bracket Associates, 2013.
4. **Ducasse Stéphane [et al.]** Pharo By Example 5 [Book]. - [s.l.] : Lulu.com & Square Bracket Associates, 2017.
5. Pharo [Онлайновий] // Wikipedia. - https://uk.wikipedia.org/wiki/Pharo.
6. Pharo By Example Two [Онлайновий]. - http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/Profiling.pdf.
7. Pharo MOOC [Онлайновий] // Pharo MOOC. - http://rmod-pharo-mooc.lille.inria.fr/.
8. Sparse Matrix [Онлайновий] // Wikipedia. - https://en.wikipedia.org/wiki/Sparse\_matrix.
9. **Tymchuk Yuriy** Concurrent Programming in Pharo [Онлайновий]. - 4 August 2014 p.. - http://pillarhub.pharocloud.com/hub/Uko/concurrentProgrammingInPharo.

# Додаток А

setAdditionLinksOn: matrixInstance

  | leftMarkerLinkBefore leftMarkerLinkAfter rightMarkerLinkBefore rightMarkerLinkAfter method |

  method := #+.

  self setAdditionLinksForResultOn: matrixInstance inMethod: method.

  "remove selection from previous marker element"

  leftMarkerLinkBefore := MetaLink new.

  leftMarkerLinkBefore

    metaObject: [ :value |

      value isNotNil

        ifTrue: [ self deselectModel: value inGroup: #left ] ].

  leftMarkerLinkBefore selector: #value:.

  leftMarkerLinkBefore arguments: #(value).

  leftMarkerLinkBefore control: #before.

  matrixInstance

    link: leftMarkerLinkBefore

    toTemporaryNamed: #leftMatrixMarker

    inMethod: method

    option: #write.

  "select current marker element"

  leftMarkerLinkAfter := MetaLink new.

  leftMarkerLinkAfter

    metaObject: [ :element |

      | leftMarkerHighlightLink |

      leftMatrixMarker := element.

      "highlights marker if it was copied - most common path in the method"

      "uninstalls itself after one use since the algorithm does not revert"

      element isNotNil

        ifTrue: [ leftMarkerHighlightLink := MetaLink new.

          leftMarkerHighlightLink

            metaObject: [ self highlightModel: element inGroup: #left.

              delay copy wait.

              leftMarkerHighlightLink uninstall ].

          leftMarkerHighlightLink selector: #value.

          leftMarkerHighlightLink control: #after.

          element link: leftMarkerHighlightLink toMethodNamed: #copy ].

      "draw the marker element on the view if its the initialization"

      leftMarkerElement isNil

        ifTrue: [ leftMarkerElement := (COOSparseMatrixVisualizer new

            getElementShapeScaled: 5) elementOn: element ].

      "select the element and update the view"

      element isNotNil

        ifTrue: [ self selectModel: element inGroup: #left.

          leftMarkerElement updateModelAndRedraw: element.

          delay copy wait ] ].

  leftMarkerLinkAfter selector: #value:.

  leftMarkerLinkAfter arguments: #(value).

  leftMarkerLinkAfter control: #after.

  matrixInstance

    link: leftMarkerLinkAfter

    toTemporaryNamed: #leftMatrixMarker

    inMethod: method

    option: #write.

  "remove selection from previous marker element"

  rightMarkerLinkBefore := MetaLink new.

  rightMarkerLinkBefore

    metaObject: [ :value |

      value isNotNil

        ifTrue: [ self deselectModel: value inGroup: #right ] ].

  rightMarkerLinkBefore selector: #value:.

  rightMarkerLinkBefore arguments: #(value).

  rightMarkerLinkBefore control: #before.

  matrixInstance

    link: rightMarkerLinkBefore

    toTemporaryNamed: #rightMatrixMarker

    inMethod: method

    option: #write.

  "select current right matrix marker element"

  rightMarkerLinkAfter := MetaLink new.

  rightMarkerLinkAfter

    metaObject: [ :element |

      | rightMarkerHighlightLink |

      rightMatrixMarker := element.

      "highlights marker if it was copied - most common path in the method"

      "uninstalls itself after one use since the algorithm does not revert"

      element isNotNil

        ifTrue: [ rightMarkerHighlightLink := MetaLink new.

          rightMarkerHighlightLink

            metaObject: [ :elementCopy |

              "elementCopy is the copied object which we use to set a link to value:"

              "with that we know if it's selecting left and right markers or only one of them"

              | leftMatrixMarkerHightlightLink |

              leftMatrixMarkerHightlightLink := MetaLink new.

              leftMatrixMarkerHightlightLink

                metaObject: [ "the link is set to right only according to the method code"

                  "left is never the object which is copied in this situation"

                  self highlightModel: leftMatrixMarker inGroup: #left.

                  delay copy wait ].

              leftMatrixMarkerHightlightLink selector: #value.

              leftMatrixMarkerHightlightLink control: #after.

              elementCopy

                link: leftMatrixMarkerHightlightLink

                toMethodNamed: #value:.

              "highlight the right marker and uninstall itself after first use"

              self highlightModel: element inGroup: #right.

              delay copy wait.

              rightMarkerHighlightLink uninstall ].

          rightMarkerHighlightLink selector: #value:.

          rightMarkerHighlightLink arguments: #(value).

          rightMarkerHighlightLink control: #after.

          element link: rightMarkerHighlightLink toMethodNamed: #copy.

          metaLinks add: rightMarkerHighlightLink ].

      rightMarkerElement isNil

        ifTrue: [ rightMarkerElement := (COOSparseMatrixVisualizer new

            getElementShapeScaled: 5) elementOn: element ].

      element isNotNil

        ifTrue: [ self selectModel: element inGroup: #right.

          rightMarkerElement updateModelAndRedraw: element.

          delay copy wait ] ].

  rightMarkerLinkAfter selector: #value:.

  rightMarkerLinkAfter arguments: #(value).

  rightMarkerLinkAfter control: #after.

  matrixInstance

    link: rightMarkerLinkAfter

    toTemporaryNamed: #rightMatrixMarker

    inMethod: method

    option: #write.

  metaLinks

    addAll:

      {leftMarkerLinkBefore.

      leftMarkerLinkAfter.

      rightMarkerLinkBefore.

      rightMarkerLinkAfter}