Kalindu De Costa
1001392619
decostak

CSCC01: Assignment 3 Report

Items used for refactoring:

```
1. Single responsibility principle
2. Interfaces and Object inheritance
3. Generics
4. Unit testing
```

Single responsibility principle

I approached the given code with the idea of every class being responsible for a single feature on the application. The project is structured using the “package by feature” approach to allow easy modularization.

```
Package Structure
    csc.summer2018.cscc01/
    ├── matrix/
    │   ├── EucledianSymmetricMatrix.java
    │   ├── EucledianSymmetricMatrixInterface.java
    │   ├── Matrix.java
    │   └── Point.java
    ├── App.java
    ├── Cfiltering.java
    └── CfilteringDriver.java
```

Prior to the refactoring **Cfiltering.java** had all of the responsibilities of the application. It was in charge of managing the matrices, performing computation on the matrices and even performing print functions for the matrices. Now its only responsibility is to manage the `user*movie` matrix and the `user*user` matrix. The `user*movie` matrix is a regular `n*m` **Matrix.java** and the `user*user` matrix is a `n*n` **EucledianSymmetricMatrix.java**. The main responsibilities of **Cfiltering.java** now are to populate the user*movie matrix and call the appropriate functions of each matrix.

Most of the heavy calculation responsibilities are given the matrix objecs. For instance a **Matrix.java** can generate a **EuclideanSymmetricMatrix.java** and a **EucledianSymmetricMatrix.java** can generate a list of maximum and minimum points within the matrix. This simply shifts the responsibilities of **Cfiltering.java** to only managing the matrices.

Lastly I have given **CfilteringDriver.java** the responsibility of performing the print functionality. It is responsible for calling the appropriate get functions of **Cfiltering.java** and printing them out in a

specific format.

Interfaces and Object inheritance

I decided to use inheritance of abstract classes to simplify the implementation for this application. **Matrix.java** represents any `n*m` matrix and an **EucledianSymmetricMatrix.java** inherits from **Matrix.java**, since the only difference between a **Matrix.java** and a **EucledianSymmetricMatrix.java** is that it is a Square matrix that is Symmetric. Also, a **Matrix.java** can generate a **EcucledianSymmetricM

Furthermore, a **EuclideanSymmetricMatrix.java** implements for a **EuclideanSymmetricMatrixInterface.java** which add the responsibilities of calculating the maximum and the minimum points within the matrix.

Generic

The **Point.java** class is created as a generic object to realistically represent a point in a matrix. It contains a `x` and a `y` value to represent its coordinates within a matrix as well as a generic `value`. Using this, **Matrix.java** is constructed as a collection of `java.lang.Number` **Point.java** objects.

Final thoughts and reasoning

Initially, I had the idea of implementing **Matrix.java** as an iterator to easily traverse through the matrix. However, after reconsideration, making **Matrix.java** an `Iterable` object does not make much sense. There is more than one way of iterating a matrix. A user can traverse by column or traverse by row. So I provided a simple `get` method to retrieve a certain point given a `x` and a `y` coordinate. This way the user is free to perform a traversal in any way and iteration can be done easily using the number of rows and the number of columns.