

Software Engineering

Assignment 3

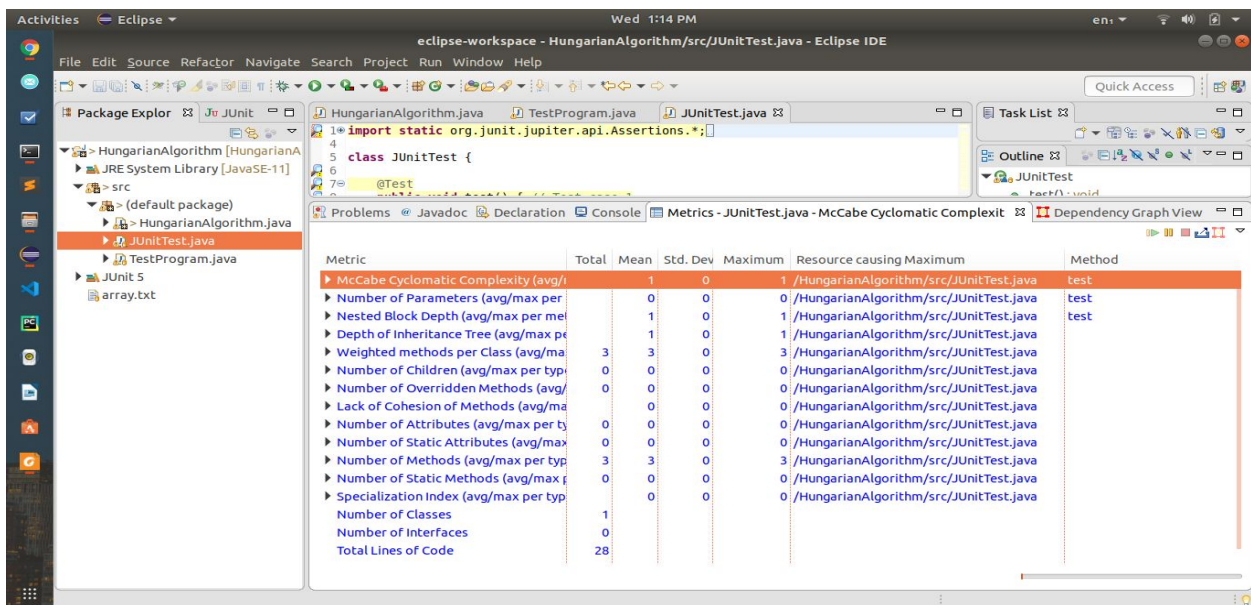
Part B

Kunal Kumar

170010012

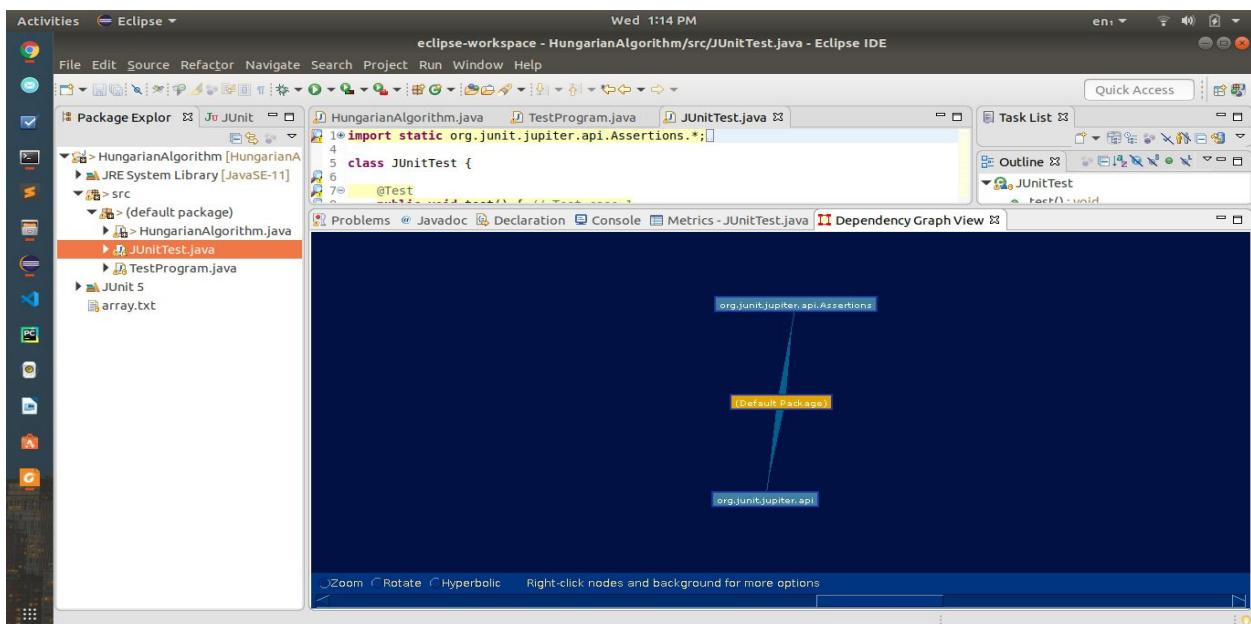
Metrics

1. Applied metrics plugin to code
2. Output of Metrics



Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/max per method)	1	1	0	1	/HungarianAlgorithm/src/JUnitTest.java	test
▶ Number of Parameters (avg/max per method)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	test
▶ Nested Block Depth (avg/max per method)	1	1	0	1	/HungarianAlgorithm/src/JUnitTest.java	
▶ Depth of Inheritance Tree (avg/max per class)	1	1	0	1	/HungarianAlgorithm/src/JUnitTest.java	
▶ Weighted methods per Class (avg/max per class)	3	3	0	3	/HungarianAlgorithm/src/JUnitTest.java	
▶ Number of Children (avg/max per type)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
▶ Number of Overridden Methods (avg/max per class)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
▶ Lack of Cohesion of Methods (avg/max per class)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
▶ Number of Attributes (avg/max per type)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
▶ Number of Methods (avg/max per type)	3	3	0	3	/HungarianAlgorithm/src/JUnitTest.java	
▶ Number of Static Methods (avg/max per type)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
▶ Specialization Index (avg/max per type)	0	0	0	0	/HungarianAlgorithm/src/JUnitTest.java	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	28					

Graph:



3. Brief Description of Metrics Report:

The metrics view indicates the progress of the metrics calculations as they are being performed in the background. It basically shows statistics of each part of the code. This plugin calculates various metrics for our code during build cycles and warns us, via the Problems view, of 'range violations' for each metric. This allows us to stay continuously aware of the health of your code base.

The view is *started from the metrics view menu or toolbar and it only works when a source folder or entire project is selected*. It shows a dynamic hyperbolic graph of the package dependencies, which can be zoomed and rotated. Use the radio buttons and scrollbar to manipulate the graph. The red and blue rectangles represent packages, and the edges their dependencies.

Checkstyle

1. Applied Checkstyle plugin to code
2. output of checkstyle plugin

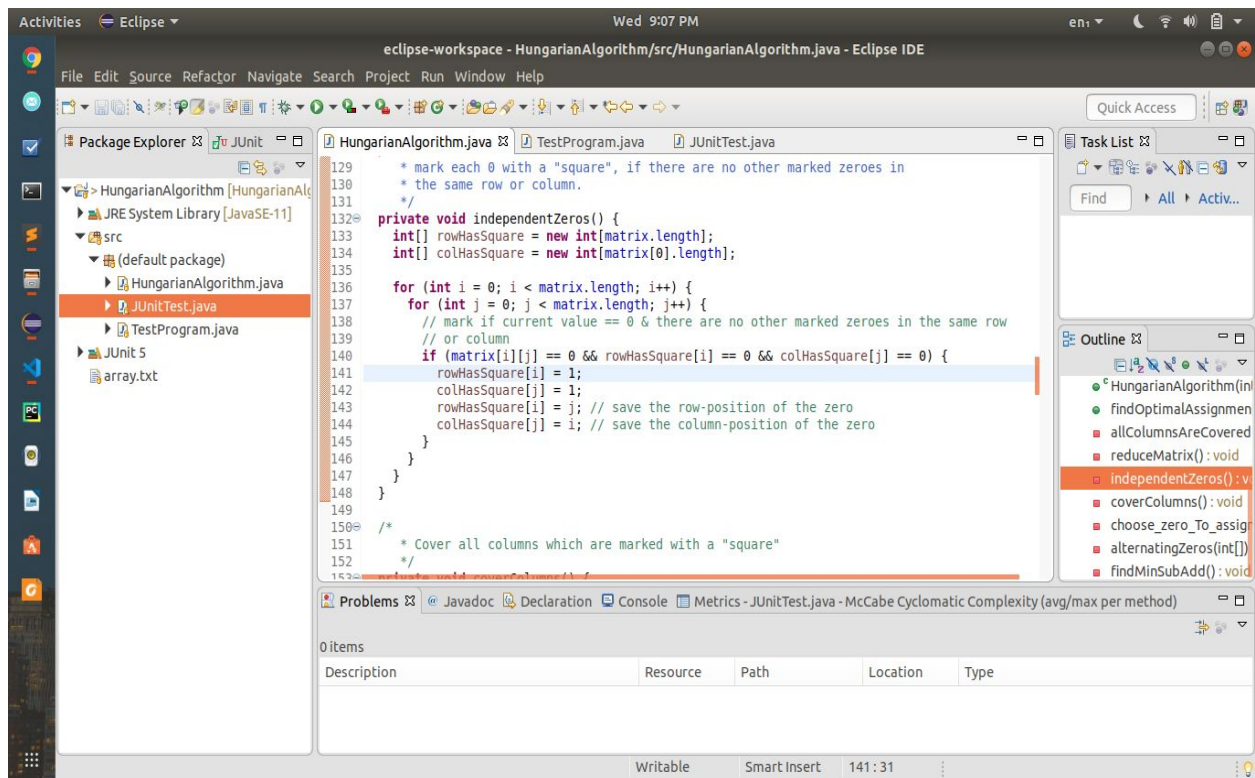
The screenshot shows the Eclipse IDE interface. The Package Explorer on the left shows the project structure. The main editor displays the code for `HungarianAlgorithm.java`. The Metrics view at the bottom shows the results of the McCabe Cyclomatic Complexity analysis.

Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/i	5	3.724	13	/HungarianAlgorithm/src/HungarianAlgorit	Find_min_sub_add	
▶ Number of Parameters (avg/max per	0.267	0.442	1	/HungarianAlgorithm/src/HungarianAlgorit	HungarianAlgorithm	
▶ Nested Block Depth (avg/max per me	2.933	1.436	5	/HungarianAlgorithm/src/HungarianAlgorit	choose_zero_To_assign	
▶ Afferent Couplina (ava/max per packe	0	0	0	/HungarianAlaorithm/src		

3. Brief Description of CodeStyle report:

The Eclipse Checkstyle Plugin integrates the static source code analyzer CheckStyle into the Eclipse IDE. With the Checkstyle Eclipse Plugin our code is constantly inspected for coding standard deviations. Within the Eclipse workbench you are immediately notified of problems via the Eclipse Problems View and source code annotations similar to compiler errors or warnings. If we work in a development team consisting of more than one person, then obviously a common ground for coding standards (formatting rules, line lengths etc.) must be agreed upon - even if it is just for practical reasons to avoid superficial, format related merge conflicts. Checkstyle (and the Eclipse Checkstyle Plugin for that matter) helps us define and easily apply those common rules. This ensures an extremely short feedback loop right at the developers fingertips.

4. Modified code



Refactoring (jSparrow)

1. Applied refactorer plugin to code
2. output of refactorer plugin

The screenshot shows the Eclipse IDE with the 'Run Summary' window open. The window title is 'Replace For-Loop with Enhanced-For-Loop'. It states: 'Since Java 1.5 enhanced for-loops can be used to iterate over collections. This rule replaces old for-loops utilizing iterators with enhanced for-loops in order to improve readability.' Below this, it shows 'Issues Fixed: 5' and 'Time Saved: 25 Minutes'. The files listed are 'HungarianAlgorithm.java - HungarianAlgorithm/src' and 'TestProgram.java - HungarianAlgorithm/src'. The main area shows a 'Java Source Compare (HungarianAlgorithm.java)' window with two columns: 'Original Source' and 'Refactored Source'. The original source code uses a standard for-loop to iterate over the rows of a matrix. The refactored source code uses an enhanced for-loop to iterate over the rows of a matrix. The refactored code is more concise and readable.

Original Source	Refactored Source
92 * each column minima from each element of the column	92 * each column minima from each element of the column
93 */	93 */
94 private void reduceMatrix() {	94 private void reduceMatrix() {
95 // rows	95 // rows
96 for (int i = 0; i < matrix.length; i++) {	96 for (int[] aMatrix : matrix) {
97 //find the min value of the current row	97 //find the min value of the current row
98 int currentRowMin = matrix[0][0];	98 int currentRowMin = matrix[0][0];
99 for (int j = 0; j < matrix[i].length; j++) {	99 for (int j = 0; j < aMatrix.length; j++) {
100 if (matrix[i][j] < currentRowMin) {	100 if (aMatrix[j] < currentRowMin) {
101 currentRowMin = matrix[i][j];	101 currentRowMin = aMatrix[j];
102 }	102 }
103 }	103 }
104 // subtract min value from each element of the current row	104 // subtract min value from each element of the current row
105 for (int k = 0; k < matrix[i].length; k++) {	105 for (int k = 0; k < aMatrix.length; k++) {
106 matrix[i][k] -= currentRowMin;	106 aMatrix[k] -= currentRowMin;
107 }	107 }

3. Brief Description on refactor

jSparrow detects and automatically replaces bugs and code smells in Java sources with a rule-based approach. jSparrow finds bugs and other code smells and replaces them with clean and modern Java code. This keeps your system healthy and strong. jSparrow automates refactoring of Java code thus it makes repetitive work unnecessary when upgrading to a new Java version. jSparrow improves our Java source code. Removal of potential bugs and code smells is one of the main purposes of jSparrow. jSparrow is very efficient in disposing of these kinds of threats with its rules. You can group the profiles into individual rule profiles, which also can be exported and imported to share them within your development team. After the run of jSparrow, it shows us how the quality of our sources can be improved. jSparrow delivers a solution for each found old language construct or threat and shows exactly how to solve it. We can run the rules

through our whole sources and review issues and their solutions according to best practices.

4. Modified code

