**SEPQM Assignment 2**

**Introduction of new complexity metric**

**Group Details**

|  |  |  |  |
| --- | --- | --- | --- |
| **Group ID** | **2021S1\_REG\_WE\_09** | | |
| **Name** | | **Registration No.** | **Responsible Factors** |
| **T. Lorence** | | **IT19039732** | * **Nested Decision Statements** * **Function calls** * **Methods being accessed through interfaces** |
| **Senadeera S.A.P.R.** | | **IT19118314** | * **Try Catch Block** * **Parameters passed inside user defined methods.** * **Nested Decision Statements** |
| **Perera T.W.I.V.** | | **IT19040172** | * **Casting** * **Declaring and initializing of variables** * **Control Statements** |
| **Amarathunga A.A.H.S.B.** | | **IT19035086** | * **Instantiating Objects** * **Lines of code** * **Operators** |

* Advantages and limitations of the CC metric

|  |
| --- |
| **Advantages**   1. Because the Cyclomatic complexity calculates the source program's independent pathways, it gives the concept of a number of test cases for testing the source code. When the number of independent paths is present, it makes it easier to test the code that many times. As a result, the entire code is covered for testing and, if necessary, bug fixing if bugs present. 2. By finding the value of the code complexity, the chances of risks being present will be reduced. Hence the cost is reduced when finding defects in source code. 3. It gives a clue of whether the program is complex and whether it should be broken down further.   **Limitations**   1. Only the decision nodes are considered. 2. Sometimes simple codes are hard to understand and read. Hence a higher complexity is not always a disadvantage. 3. Sometimes code complexity can increase due to high coding standards. |

* Advantages and limitations of the WCC metric

|  |
| --- |
| **Advantages**   1. WCC is a better indicator of complexity because it shows a higher value of complexity for a given class 2. Unlike in CC, in WCC the weight of the whole class is calculated. 3. This also considers the class level and the inheritance when calculating the weight of the class.   **Limitations**   1. WCC has a complexity limitation when considering the accuracy of the complexity for a given class with inheritance. 2. When the weight is increased, a higher effort must be used in order to understand the code. 3. High weight complexity means poor design, which sometimes can lead to unmanageable situations that increase the maintenance efforts drastically. |

* Advantages and limitations of the of CFS metric

|  |
| --- |
| **Advantages**   1. Provide base for cross-platform analysis of complexity and size of both specification and implementations. 2. More robust and independent of language and implementations.   **Limitations**   1. When calculating the weight, if the input values and the output values are zero, the SF value become zero. 2. Cognitive functional size does not consider the values from interfaces. |

* How the limitation of the CC, WCC, and CFS metrics be overcome

|  |
| --- |
| **How the limitations of the CC metric be overcome**  By introducing a new metric that considers not only the decision nodes but also the other factors as well.  **How the limitations of the WCC metric be overcome**  When writing the code, steps must be taken in order to minimize the complexity in places that is possible. And to reduce the overall weight of the code, so it will become more readable and more understandable. When the complexity is low, it will be considered as a code with high coding standards.  **How the limitations of the CFS metric be overcome**  The weightage should be assigned before doing calculations. CFS should consider the values from the interfaces when calculating. |

* Brief explanation of the newly proposed complexity metric

|  |
| --- |
| The complexity is calculated based on several factors. All these factors are assigned weightages, and this is how the  factors considered are injected into the equation. The lines of code index is calculated. The complexity is calculated by adding the total weightages and the lines of code index. This complexity metric takes into account many factors that have not been considered in other complexity metric calculations. |

* Factors considered by the new complexity metric. A group member should propose a **minimum** of **two factors**.

|  |
| --- |
| 1. Lines of code 2. Casting 3. Parameters passed inside user defined methods. 4. Operators 5. Methods being accessed through interfaces. 6. Declaring and initialization of variables 7. Control Structures 8. Nested Decision statements 9. Import statements. 10. Usage of try catch blocks. 11. Instantiating objects 12. Function calls |

* How the new metric captures the complexity introduced by each of its factors. If there are more than eight factors, add sub-headings accordingly.

|  |
| --- |
| How the new metric captures the complexity introduced due to lines of code:   * A weight will be included for a number of lines of code. For every 100 lines of code, the weight will be increased by 1. The weight for the 1st 100 lines of code will be 1.   How the new metric captures the complexity introduced due to Casting:   * For every type casting the weight will be 1   How the new metric captures the complexity introduced due to the Parameters passed inside user defined methods:   * For every parameter passed into a method, the weight will be 1. Assume n parameters were passed, then the weight would be n.   How the new metric captures the complexity introduced due to Operators:   * For every operator the weight would be 1.   How the new metric captures the complexity introduced due to Methods being accessed through interfaces:   * If a function is accessed through an interface, the weight of it would be 3.   How the new metric captures the complexity introduced due to Declaring of variables:   * If the variables have been declared at the beginning and initialized later, the weight would be 2. * If the variable has been initialized at the point of declaration the weight would be 3.   How the new metric captures the complexity introduced due to the Control Structures:   * Sequential weight = 1 * If-Else branch, weight = 2 * Switch, weight (For n number of cases weight would be n) = Weight = n * Iterations, weight = 3   How the new metric captures the complexity introduced due to the Nested Decision statements:   * If the level is 1, weight is 1, if the level is 2 the weight is 2 and if the level is n, the weight would be n for the decision statement.   How the new metric captures the complexity introduced due to the Import statements.:   * If there are import statements, the weight would be 1.   How the new metric captures the complexity introduced due to the instantiating of objects.:   * For instantiating object, weight is 2 for every instantiation.   How the new metric captures the complexity introduced due to the calling of user defined or any function.   * For every function call the weight will be 1   How the new metric captures the complexity introduced due to the usage of try catch blocks   * For every try catch block the complexity will be 2 |

Complexity calculation formula/equation of the new metric. Mention the meanings of all the characters/symbols used in the formula/equation.

|  |
| --- |
| W = Weights  C(g) = Complexity  L = Lines of code index  L = Lines of code/100 and get the value omitting the decimal places.  E.g.: 543/100 = 5.54 Then, L = 5  If the Lines of Code < 100; then L = 1 |

* Rationale behind the complexity calculation formula/equation of the new metric

|  |
| --- |
| When creating the equation, we have considered multiple factors such as the weight that must be allocated to certain parts of the code. For every type casting a weight of 1 will be added. And for every parameter that is passed into a function a weight of 1 is added. For example, if n number of parameters are added, a weight of n is assigned. If there are operators included in the code, a weight of 1 is added for every operator. If a function is accessed through an interface, a weight of 3 is assigned to that function. When declaring variables if the variables have been declared at the beginning and have been initialized later, a weight of 2 is being allocated. And if the variables have been declared and initialized at the same time a weight of 3 is being allocated to it. When considering control structures if there are sequential statements a weight of 1 is being added, and also if the code contains if else statements a weight of w is being allocated for each and every if statement. If there is a switch statement a weight of 1 is added for every case statement (if there are n number if case statements a weight of n is allocated forte whole switch statement). If there are iterative statements a weight of 3 is being allocated for every statement. And for the nested decision statements a weight of 1 is being allocated for every level of statements (if the level of the nested statement is 1, the value allocated is n, If the level of the nested statement is n, the allocated value is n). If the code contains import statements for every import statement weight of 1 is being allocated. When instantiating objects, for each and every object a weight of 2 is being allocated. And each time a function is being called, a weight of 1 is being allocated to the list. Finally, if there are any exception handling that is included in the code a weight of 2 is being allocated for each try catch block.  With respect to all the aforementioned factors, we have derived an equation that calculates the complexity by adding all the derived weights that was extracted from the codebase with the lines of code index. The lines of code index is derived by dividing the number of lines of codes by 100 (for example, if there are 543 lines of code, when divided by 100 the result would be 5.43. But we only consider the whole number, hence we would take 5 as the result. And if there are less than 100 lines of code, an index of 1 is allocated for L). |

* Calculation of complexity of the first java program using the newly proposed metric.

|  |
| --- |
| Source code of the first java program  package com.company; import java.util.Scanner; //import : w=1 public class Main {  public static void main(String[] args) {  //Parameters : w=1   Scanner sc = new Scanner(System.*in*);  //Operators : w=2  // Parameters : w=1  // Object : w=2   String mark1StringValue;  //Variables : w=2   float mark2, mark3, mark1;  //Variables : w=6   System.*out*.println("Enter 3 marks");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   mark1StringValue = sc.nextLine();  //Operators : w=2  // Function call : w=2   mark2 = sc.nextFloat();  //Operators : w=2  // Function call : w=2   mark3 = sc.nextFloat();  //Operators : w=2  // Function call : w=2   mark1 = Float.*parseFloat*(mark1StringValue);  //Operators : w=2  // Function call : w=2  //casting : w=1  // Parameters : w=1   CalcAverage ob = new Calculator();  //Operators : w=1  // Object : w=2   float average = ob.calcAverage(mark1, mark2, mark3);  //Operators : w=3  // Parameters : w=1  // access via interface = 3  // Function call : w=2   System.*out*.println("Average is " + average);  //Operators : w=3  // Parameters : w=1  // Function call : w=2   if (average >= 80 && average <= 100) {  //Operators : w=3   System.*out*.println("Grade is an A");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   } else if (average >= 60) {  //Operators : w=1   System.*out*.println("Grade is a B");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   } else if (average >= 45) {  //Operators : w=1   System.*out*.println("Grade is a C");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   } else if (average >= 0) {  //Operators : w=1   System.*out*.println("Fail. Dont give up! Try again!!");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   } else System.*out*.println("Invalid");  //Operators : w=2  // Parameters : w=1  // Function call : w=2  } }  class Calculator implements CalcAverage {  public float calcAverage(float mark1, float mark2, float mark3) {  //Method parameter : w=3   float average;  //Variables : w=2  //Operators : w=1   float total = mark1 + mark2 + mark3;  //Variables : w=3  //Operators : w=3   average = total / 3;  //Operators : w=2   return average;  } }  interface CalcAverage {  float calcAverage(float mark1, float mark2, float mark3); } //Control structure : w=9 |
| Calculation of complexity of the first java program   |  |  | | --- | --- | | **Lines of code** | **Weight** | | package com.company;  import java.util.Scanner;  //import : w=1 | 1 | | public class Main {  public static void main(String[] args) {  //Parameters : w=1 | 1 | | Scanner sc = new Scanner(System.in);  //Operators : w=2  // Parameters : w=1  // Object : w=2 | 5 | | String mark1StringValue;  //Variables : w=2 | 2 | | float mark2, mark3, mark1;  //Variables : w=6 | 6 | | System.out.println("Enter 3 marks");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | mark1StringValue = sc.nextLine();  //Operators : w=2  // Function call : w=2 | 4 | | mark2 = sc.nextFloat();  //Operators : w=2  // Function call : w=2 | 4 | | mark3 = sc.nextFloat();  //Operators : w=2  // Function call : w=2 | 4 | | mark1 = Float.parseFloat(mark1StringValue);  //Operators : w=2  // Function call : w=2  //casting : w=1  // Parameters : w=1 | 6 | | CalcAverage ob = new Calculator();  //Operators : w=1  // Object : w=2 | 3 | | float average = ob.calcAverage(mark1, mark2, mark3);  //Operators : w=3  // Parameters : w=1  // access via interface = 3  // Function call : w=2 | 9 | | System.out.println("Average is " + average);  //Operators : w=3  // Parameters : w=1  // Function call : w=2 | 6 | | if (average >= 80 && average <= 100) {  //Operators : w=3 | 3 | | System.out.println("Grade is an A");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | } else if (average >= 60) {  //Operators : w=1 | 1 | | System.out.println("Grade is a B");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | } else if (average >= 45) {  //Operators : w=1 | 1 | | System.out.println("Grade is a C");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | } else if (average >= 0) {  //Operators : w=1 | 1 | | System.out.println("Fail. Dont give up! Try again!!");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | } else System.out.println("Invalid");  //Operators : w=2  // Parameters : w=1  // Function call : w=2  }  } | 5 | | class Calculator implements CalcAverage {  public float calcAverage(float mark1, float mark2, float mark3) {  //Method parameter : w=3 | 3 | | float average;  //Variables : w=2  //Operators : w=1 | 3 | | float total = mark1 + mark2 + mark3;  //Variables : w=3  //Operators : w=3 | 6 | | average = total / 3;  //Operators : w=2  return average;  }  } | 2 | | interface CalcAverage {   float calcAverage(float mark1, float mark2, float mark3);  }  //Control structure : w=9 | 9 | | **Total** | **110** |     Lines of code = 121  L = 121/100 = 1.21 = 1  C(g) = 112 + 1 = 113 |

* Explanation of how the complexity of the first java program was calculated using the newly proposed metric.

|  |
| --- |
| At the beginning of the code there is one import statement. Hence, we have allocated a weight of 1 for that. Next, we can see 1 parameter that is being implemented, so weight of 1 is being allocated to the code. In the third section we can see a operator, a parameter and an object declaration., hence a weight of 5 has been allocated with regards to the functionalities that have been implemented. For the 4th section we have allocated a weight of 2 for the variables. And in the 5th section we have allocated a weight of 6 for the 3 available variables. For the next part we have allocated a weight of 5 just as the 3rd section. Next, for the 7,8 and 9 parts, a weight of 4 has been allocated for the available operator and the function call. For the section 10 a weight of 6 is being allocated for the available operators, function calls, casting and the parameters. For the 11th section a weight of 3 is being allocated due to the availability of operators and an object. For the next or the 12th section a weight of 9 is being allocated due to the availability of operators, parameters, access via interface and function call. For the next part a weight of 6 is being allocated. For the if statement a weight of 3 is being allocated. Sections 15, 17, 19, 21 and 22 have being allocated a weight of 5 due to the availability of operators, parameters and function calls. Sections, 16, 18 and 20 have been allocated with 1 weight. For the section 23 a weight of 3 is being allocated for the method parameter. And for the next part a weight of 3 is also being allocated for the variables and operators. For the next part a weight of 6 is being allocated. For the section before the last weight of 2 is being allocated for the operators. For the final part a weight of 9 is being allocated for the control structure. Finally, a total weight of 110 is being calculated with respect to the above-mentioned factors. |

|  |
| --- |
| Calculation of complexity of the second java program using the newly proposed metric.Source code of the second java program  package com.company;  import java.util.Calendar; import java.util.InputMismatchException; import java.util.Scanner; //import : w=3  public class Main {   public static void main(String[] args) {  //Parameters : w=1  Scanner key = new Scanner(System.*in*);  //Operators : w=2  // Parameters : w=1  // Object : w=2   int nic = 0;  //Variables : w=3  //Operators : w=1   System.*out*.println("------------Generate Company User ID------------");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   System.*out*.print("Enter NIC Number (Without letters): ");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   try {  nic = key.nextInt();  //Operators : w=2  // Function call : w=2   } catch (InputMismatchException e) {  System.*out*.println("InputMismatchException. ID Generation fail!");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   System.*exit*(-99);  //Operators : w=1  // Parameters : w=1  // Function call : w=2   } catch (Exception e) {  e.printStackTrace();  //Operators : w=1  // Function call : w=2  }    IidGenerator idGenerator = new IdGenerator();  //Operators : w=1  // Object : w=2   System.*out*.println("Your ID is : " + idGenerator.generate(nic));  //Operators : w=3  // Parameters : w=1  // access via interface = 3  // Function call : w=2   System.*out*.println("Thank You!");  //Operators : w=2  // Parameters : w=1  // Function call : w=2   } }  class IdGenerator implements IidGenerator {  public String generate(int nic) {  //Method parameter : w=1   String CompanyID;  //Variables : w=2   int lastNumsOfNic = nic % 10000;  //Operators : w=2  // Variables : w=3   String lastNum = String.*valueOf*(lastNumsOfNic);  //casting : w=1  // Operators : w=2  // Parameters : w=1  // Variables : w=3  // Function call : w=2   CompanyID = "SLIIT\_" + Calendar.*getInstance*().get(Calendar.*YEAR*) + "\_" + lastNum;  //Operators : w=7  // Parameters : w=1  // Variables : w=3   return CompanyID;  } }  interface IidGenerator {  String generate(int nic); } //Control structure : w=1 //try catch : w=2 //lines of code : w=1(before comments) |
| Calculation of complexity of the second java program   |  |  | | --- | --- | | **Lines of Code** | **Weight** | | package com.company;    import java.util.Calendar;  import java.util.InputMismatchException;  import java.util.Scanner;  //import : w=3 | 3 | | public class Main {    public static void main(String[] args) {  //Parameters : w=1 | 1 | | Scanner key = new Scanner(System.in);  //Operators : w=2  // Parameters : w=1  // Object : w=2 | 5 | | int nic = 0;  //Variables : w=3  //Operators : w=1 | 4 | | System.out.println("------------Generate Company User ID------------");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | System.out.print("Enter NIC Number (Without letters): ");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | try {  nic = key.nextInt();  //Operators : w=2  // Function call : w=2 | 4 | | } catch (InputMismatchException e) {  System.out.println("InputMismatchException. ID Generation fail!");  //Operators : w=2  // Parameters : w=1  // Function call : w=2 | 5 | | System.exit(-99);  //Operators : w=1  // Parameters : w=1  // Function call : w=2 | 4 | | } catch (Exception e) {  e.printStackTrace();  //Operators : w=1  // Function call : w=2  } | 3 | | IidGenerator idGenerator = new IdGenerator();  //Operators : w=1  // Object : w=2 | 3 | | System.out.println("Your ID is : " + idGenerator.generate(nic));  //Operators : w=3  // Parameters : w=1  // access via interface = 3  // Function call : w=2 | 9 | | System.out.println("Thank You!");  //Operators : w=2  // Parameters : w=1  // Function call : w=2    }  } | 5 | | class IdGenerator implements IidGenerator {  public String generate(int nic) {  //Method parameter : w=1 | 1 | | String CompanyID;  //Variables : w=2 | 2 | | int lastNumsOfNic = nic % 10000;  //Operators : w=2  // Variables : w=3 | 5 | | String lastNum = String.valueOf(lastNumsOfNic);  //casting : w=1  // Operators : w=2  // Parameters : w=1  // Variables : w=3  // Function call : w=2 | 9 | | CompanyID = "SLIIT\_" + Calendar.getInstance().get(Calendar.YEAR) + "\_" + lastNum;  //Operators : w=7  // Parameters : w=1  // Variables : w=3    return CompanyID;  }  } | 11 | | interface IidGenerator {  String generate(int nic);  }  //Control structure : w=1  //try catch : w=2 | 3 | | **Total** | **87** |     Lines of code = 104  L = 104/100 = 1.04 = 1  C(g) = 87 + 1 = 88 |

* Explanation of how the complexity of the second java program was calculated using the newly proposed metric

|  |
| --- |
| For the import statements a value of 3 is being allocated for the 3 available import statements. For the main statement a single weight is allocated. For the 3rd part a weight of 5 is being allocated for the operators' parameters and the object that is available in the section. Next a weight of 4 is being allocated for the variables and operators. For the 5th and 6th parts a weight of 5 is being allocated for the available operators, parameters and the function call. For the 7th part a weight of 4 has being added for the try statement. For the section 8 a weight of 5 has being allocated. For the operators, parameters and the function call. And for the next part a weight of 4 has being allocated for the operators parameters and the function call. And for ty final catch statement a weight of 3 has being allocated. For the next section a weight of weight of 3 has being allocated for the operator and the object. For the 12th section a weight of 12 has being allocated for the operators, parameters, access via interface and the function call. For the sop statement a weight of 5 has being allocated. For the 14th section a weight of 1 is allocated. For the next part a weight of 2 is allocated. For the section 16 a weight of 5 is allocated for the operators and variables. For the 17th section a weight of 9 is being assigned due to the availability of casting, operators, variables, parameters and function call. And for the 18th part a weighty of 11 is being assigned for the fact that it contains operator, parameters and variables in that section. For the final section a weight of 3 has being allocated since it contains control structure. And for the try catch block a weight of 2 is being allocated. With respect to the above factors the addition of all the weight is calculated as 87 and the complexity is calculated as 88. |

* References

|  |
| --- |
| 1. De Silva, D.I., Kodagoda, N., Kodituwakku, S.R. and Pinidiyaarachchi, A.J., 2015, September. Limitations of an object-oriented metric: Weighted complexity measure. In 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS) (pp. 698-701). IEEE. 2. Yingxu Wang and Jingqiu Shao, "Measurement of the cognitive functional complexity of software," The Second IEEE International Conference on Cognitive Informatics, 2003. Proceedings., 2003, pp. 67-74, doi: 10.1109/COGINF.2003.1225955. 3. Misra, Sanjay and Akman, I. (2008) Weighted Class Complexity: A Measure of Complexity for Object Oriented System. JOURNAL OF INFORMATION SCIENCE AND ENGINEERING, 24. pp. 1689-1708. 4. Vinju, J.J. and Godfrey, M.W., 2012, September. What does control flow really look like? Eyeballing the cyclomatic complexity metric. In 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (pp. 154-163). IEEE. 5. Chhillar, U. and Bhasin, S., 2011. A new weighted composite complexity measure for object-oriented systems. International journal of information and communication technology research, 1(3). 6. Misra, S. and Akman, I., 2008. Weighted class complexity: a measure of complexity for object oriented system. Journal of Information Science and Engineering, 24, pp.1689-1708. 7. Maheswaran, K. and Aloysius, A., 2018. Cognitive weighted inherited class complexity metric. *Procedia Computer Science*, *125*, pp.297-304 |