Project Specification

CDE IT Solutions is a startup and is evolving rapidly. At present, they only concentrate on developing **C++** and **Java** based projects. The primary objective of CDE IT solutions is to build quality software that could be easily maintained. Assume your group is working for CDE IT solutions. In an attempt to reduce the maintenance cost of the software developed by the company, the CEO has requested your group to develop a **web-based code complexity measuring tool**. Accordingly, he wants to measure the complexity introduced due to the following factors:

a. Size, variables, and methods
b. Inheritance
c. Coupling
d. Control structures

Measuring the complexity of a program statement due to size

- Complexity of a program statement due to its size can be computed as follows:

$$Cs = (W_{kw} * N_{kw}) + (W_{id} * N_{id}) + (W_{op} * N_{op}) + (W_{nv} * N_{nv}) + (W_{sl} * N_{sl})$$

Where:

C_S = Complexity of a program statement due to its size

W_{kw} = Weight due to keywords or reserved words (class, public, void, true, else, default, return, null, break, this, etc.)

N_{kw} = Number of keywords or reserved words in the program statement

W_{id} = Weight due to identifiers (names of classes, methods, objects, variables, arguments, and data structures)

N_{id} = Number of identifiers in the program statement

W_{op} = Weight due to operators

N_{op} = Number of operators in the program statement

W_{nv} = Weight due to numerical values or numbers

N_{nv} = Number of numerical values in the program statement

W_{sl} = Weight due to string literals (“ ” ,“Hi”, “Hello World”, “The factorial value of the entered number is”, etc.)

N_{sl} = Number of string literals in the program statement

- The weights allocated for the program components which are considered under the size factor are as follows:

Program Component	Weight
A keyword	1
An identifier	1
An operator	1
A numerical value	1
A string literal	1

- Following are **not** considered as keywords or reserved words:
  - Data types (int, float, string, char, boolean, etc.)
  - Names of control structures (‘for’, while, do-while, ‘if’, ‘switch’, ‘case’)

- Names of variables and parameters are **not** considered as identifiers, in lines where they are declared or defined. However, names of variables which are declared or defined inside control structures are considered as identifiers.

- Operators are divided into six categories as follows:
  - Arithmetic operators → { + - * / % ++ -- }
  - Relation operators → { == != > < >= <= }
  - Logical operators → { && || ! }
  - Bitwise operators → { | ^ ~ << >> >>> <<< }
  - Miscellaneous operators → { , -> . :: }
  - Assignment operators → {+= -= *= /= = >>>= |= &= %= <<= >>= ^= }

Measuring the complexity of a program statement due to variables

Note: Only the lines which consist of declared or defined variables are considered under this factor. However, variables declared or defined inside control structures are **not** considered under this factor.

- Complexity of a program statement due to its variables are computed as follows:

  Cv = Wvs **[**(Wpdtv * Npdtv) + (Wcdtv * Ncdtv)**]**

  Where:

  Cv = Complexity of a program statement due to its variables

  Wvs = Weight due to variable scope

  Wpdtv = Weight of primitive data type variables

  Npdtv = Number of primitive data type variables

  Wcdtv = Weight of composite data type variables

  Ncdtv = Number of composite data type variables

- Based on the scope, variables are divided into two categories as follows:
  1. Global variables - Variables which are declared/defined inside a class, but outside a method.
  2. Local variables - Variables declared/defined in a method or constructor.

- A weight of **two** is assigned for each global variable and a weight of **one** is assigned for each local variable.

- Based on the data type, variables are divided into two categories as follows:
  1. Primitive data type variables – Variables which are declared/defined using built-in data types such as in, long, double, float, boolean, char, etc.
  2. Composite data type variables – Variables which are derived based on a combination of primitive and other composite data types such as arrays, objects, records, interfaces, lists, etc.

- A weight of **one** is assigned for each primitive data type variable and a weight of **two** is assigned for each composite data type variable.

Measuring the complexity of a program due to methods

Note: Only the lines which include method signatures are considered under this factor.

- Complexity of a line which includes of a method signature can be computed as follows:

  Cm = Wmrt + (Wpdtp * Npdtp) + (Wcdtp * Ncdtp)

  Where:

  Cm = Complexity of a line which includes a method signature

  Wmrt = Weight due to method return type

  Wpdtp = Weight of primitive data type parameters

  Npdtp = Number of primitive data type parameters

  Wcdtp = Weight of composite data type parameters

  Ncdtp = Number of composite data type parameters

- Method return types are divided into two categories as follows:
  1. Methods with a primitive return type – Method return types with built-in data types such as int, long, double, float, boolean, char, etc.
  2. Method with a composite return type – Method return types which are derived based on a combination of primitive and other composite data types such as arrays, objects, records, interfaces, lists, etc.

- A weight of **one** is assigned for each method with a primitive return type and a weight of **two** is assigned for each method with a composite return type. However, methods with a 'void' return type are **not** considered under this sub-factor. Thus, a value of **zero** is assigned for a method with a 'void' return type.

- Like method return types, parameters are also categorized as primitive and composite. Accordingly, a weight of **one** is assigned for a primitive data type parameter and a weight of **two** is assigned for a composite data type parameter.

Measuring the complexity of a program statement due to inheritance

- The inheritance complexity of a program statement which belongs to a class is same as the weight assigned for that class due to its inheritance pattern:

Inheritance complexity of a program statement of a class (Ci) = Weight assigned for that class due to its inheritance pattern

- Depending on the inheritance pattern, each inherited class is assigned the following weights:

Inherited Pattern	Weight
A class with out any inheritance (direct or indirect)	0
A class inheriting (directly or indirectly) from **one** user-defined class	1
A class inheriting (directly or indirectly) from **two** user-defined classes	2
A class inheriting (directly or indirectly) from **three** user-defined classes	3
A class inheriting (directly or indirectly) from **more than three** user-defined classes	4

Measuring the complexity of a program statement due to coupling

Note: All **built-in methods** are treated as **regular methods** that reside in the **same file**.

- Complexity of program statement due to coupling (Ccp) is computed as follows:

$$
\begin{aligned}
Ccp = {} & (Wr * Nr) + (Wmcms * Nmcms) + (Wmcmd * Nmcmd) + (Wmcrms * Nmcrms) + (Wmcrmd * Nmcrmd) + \\
& (Wrmcrms * Nrmcrms) + (Wrmcrmd * Nrmcrmd) + (Wrmcms * Nrmcms) + (Wrmcmd * Nrmcmd) + \\
& (Wmrgvs * Nmrgvs) + (Wmrgvd * Nmrgvd) + (Wrmrgvs * Nrmrgvs) + (Wrmrgvd * Nrmrgvd)
\end{aligned}
$$

Where:

Wr	= Weight of a recursive call
Nr	= Number of recursive calls
Wmcms	= Weight of a **regular method** calling another **regular method** in the same file
Nmcms	= Number of calls from **regular method(s)** to other **regular methods** in the same file
Wmcmd	= Weight of a **regular method** calling another **regular method** in a different file
Nmcmd	= Number of calls from **regular method(s)** to other **regular methods** in different files
Wmcrms	= Weight of a **regular method** calling a **recursive method** in the same file
Nmcrms	= Number of calls from **regular method(s)** to **recursive methods** in the same file
Wmcrmd	= Weight of a **regular method** calling a **recursive method** in a different file
Nmcrmd	= Number of calls from **regular method(s)** to **recursive methods** in different files
Wrmcrms	= Weight of a **recursive method** calling another **recursive method** in the same file
Nrmcrms	= Number of calls from **recursive method(s)** to other **recursive methods** in the same file
Wrmcrmd	= Weight of a **recursive method** calling another **recursive method** in a different file
Nrmcrmd	= Number of calls from **recursive method(s)** to other **recursive methods** in different files
Wrmcms	= Weight of a **recursive method** calling a **regular method** in the same file
Nrmcms	= Number of calls from **recursive method(s)** to **regular methods** in the same file
Wrmcmd	= Weight of a **recursive method** calling a **regular method** in a different file
Nrmcmd	= Number of calls from **recursive method(s)** to **regular methods** in different files
Wmrgvs	= Weight of a **regular method** referencing a **global variable** in the same file
Nmrgvs	= Number of references from **regular method(s)** to **global variables** in the same file
Wmrgvd	= Weight of a **regular method** referencing a **global variable** in a different file
Nmrgvd	= Number of references from **regular method(s)** to **global variables** in different files
Wrmrgvs	= Weight of a **recursive method** referencing a **global variable** in the same file
Nrmrgvs	= Number of references from **recursive method(s)** to **global variables** in the same file
Wrmrgvd	= Weight of a **recursive method** referencing a **global variable** in a different file
Nrmrgvd	= Number of references from **recursive method(s)** to **global variable** in different files

- The weight allocated for a program statement due to coupling differs as follows:

Coupling Type	Weight
A recursive call (Refer to Ex1 in fig. 1)	2
A **regular** method calling another **regular** method in the **same file**	2
A **regular** method calling another **regular** method in a **different file**	3
A **regular** method calling a **recursive** method in the **same file** (Refer to Ex4 in fig. 1)	3
A **regular** method calling a **recursive** method in a **different file**	4
A **recursive** method calling another **recursive** method in the **same file** (Refer to Ex2 in fig. 1)	4
A **recursive** method calling another **recursive** method in a **different file**	5
A **recursive** method calling a **regular** method in the **same file**	3
A **recursive** method calling a **regular** method in a **different file**	4
A r**egular** method referencing a **global** variable in the **same file (**Refer to Ex3 in fig. 1)	1
A r**egular** method referencing a **global** variable in a **different file**	2
A **recursive** method referencing a **global** variable in the **same file**	1
A **recursive** method referencing a **global** variable in a **different file**	2

Measuring the complexity of a program statement due to control structures

Note:

- Only the program statements with control structures are considered under this factor.
- Always for program statements with 'case' statements, value of NC would be one. In addition, the Ccspps value of each 'case'' inside a 'switch' would be the Ccs value obtained for the program statement with the 'switch' control structure.

- Complexity of a program statement with a control structure is computed as follows:

Ccs = (Wtcs * NC) + Ccspps

Where:

Ccs = Complexity of a program statement with a control structure

Wtcs = Weight due to control structure type

NC = Number of conditions in the control structure

Ccspps = **Ccs** value of the program statement in the **immediate outer level** of the **current nesting level**. Hence, always the value of **Ccspps** would be **zero** for control structures which reside at the **first nesting level** or **outer most nesting level**.

➢ Depending on the type, each control structure is assigned with the following weights:

Control Structure Type	Weight
A conditional control structure such as an ‘if’ or ‘else-if’ condition	2
An iterative control structure such as a ‘for’, ‘while’, or ‘do-while’ loop	3
The ‘switch’ statement in a ‘switch-case’ control structure	2
Each ‘case’ statement in a ‘switch-case’ control structure	1

```
import java.util.Scanner;
public class Maths2{
 double num = 0.0;

 public static void main(String[ ] args){
    Maths2 m2 = new Maths2();
    m2.answer();
 }

 public double getSqr(double num1) {
   if (num1 == 0)
    return 0;
   else
    return getSqr(num1-1) + (2 * num1) - 1;
 }

 public double getCube(double num2) {
   if (num2 == 0)
    return 0;
   else
    return getCube(num2-1) + 3*(getSqr(num2)) - 3*num2 + 1;
 //Ex1:getCube() method calling itself, is an example for a recursive call.
 //Ex2:getCube() method calling the getSqr() method, is an example for a recursive method calling another recursive method in the same file.
 }

 public void answer(){
   Maths2 m1 = new Maths2();

   Scanner input = new Scanner(System.in);
   System.out.print("Enter a number:");
   num = input.nextInt();
 //Ex3: answer() method referencing 'num' variable, is an example for a regular method referencing a global variable in the same file.

   double n2 = m1.getSqr(num);
 //Ex4: answer() method calling the getSqr() method, is an example for a regular method calling a recursive method in the same file.

   System.out.println("Squared value of " + num + " is " + n2);
   System.out.println("Cube value of " + num + " is " + m1.getCube(num));
 }
}
```

Fig. 1 : A sample program displaying different coupling types

Measuring total complexity of a program statement

- Total complexity of a program statement (TCps) is computed as follows:

  TCps = Cs + Cv + Cm + Ci + Ccp + Ccs

Measuring complexity of a program

- The complexity of a program (Cpr) with 'n' number of program statements is computed as follows:

$$Cpr = \sum_{k=0}^{n} (TCps)_k$$

Displaying the complexity of a program due to size

Line no	Program statements	Nkw	Nid	Nop	Nnv	Nsl	Cs
1	class Pattern {	1	1				2
2	public static void main(String[] args) {	3	1				4
3	int rows = 5;			1	1		2
4	for(int i = 1; i <= rows; ++i) {		4	3	1		8
5	for(int j = 1; j <= i; ++j) {		4	3	1		8
5	System.out.print(j + " ");		4	3		1	8
6	}						0
7	System.out.println("");		3	2		1	6
8	}						0
9	}						0
10	}						0

Displaying the complexity of a program due to variables

Line no	Program statements	Wvs	Npdtv	Ncdtv	Cv
1	class Pattern {				0
2	public static void main(String[] args) {				0
3	int rows = 5;	1	1	0	1
4	for(int i = 1; i <= rows; ++i) {				0
5	for(int j = 1; j <= i; ++j) {				0
5	System.out.print(j + " ");				0
6	}				0
7	System.out.println("");				0
8	}				0
9	}				0
10	}				0

Displaying the complexity of a program due to methods

Line no	Program statements	Wmrt	Npdtp	Ncdtp	Cm
1	class Pattern {				0
2	public static void main(String[] args) {	0	0	1	2
3	int rows = 5;				0
4	for(int i = 1; i <= rows; ++i) {				0
5	for(int j = 1; j <= i; ++j) {				0
5	System.out.print(j + " ");				0
6	}				0
7	System.out.println("");				0
8	}				0
9	}				0
10	}				0

Displaying the complexity of a program due to inheritance

Count	Class Name	No of direct inheritances	No of indirect inheritances	Total inheritances	Ci
1	Pattern	0	0	0	0
2					
3					
4					
5					
5					
6					
7					
8					
9					
10					

Note:

- Total inheritances (Ti) = No of direct inheritances (Ndi) + No of indirect inheritances (Nidi)
- If Ti value is less than or equal to three, then Ci = Ti. However, if the Ti value is greater than three, then Ci = 4

Displaying the complexity of a program due to coupling

Line no	Program statements	Nr	Nmcms	Nmcmd	Nmcrms	Nmcrmd	Nrmcrms	Nrmcrmd	Nrmcms	Nrmcmd	Nmrgvs	Nmrgvd	Nrmrgvs	Nrmrgvd	Ccp
1	class Pattern {														0
2	public static void main(String[ ] args) {														0
3	int rows = 5;														0
4	for(int i = 1; i <= rows; ++i) {														0
5	for(int j = 1; j <= i; ++j) {														0
5	System.out.print(j + " ");		1												2
6	}														0
7	System.out.println("");		1												2
8	}														0
9	}														0
10	}														0

Displaying the complexity of a program due to control structures

Line no	Program statements	Wtcs	NC	Ccspps	Ccs
1	class Pattern {				0
2	public static void main(String[] args) {				0
3	int rows = 5;				0
4	for(int i = 1; i <= rows; ++i) {	3	1	0	3
5	for(int j = 1; j <= i; ++j) {	3	1	3	6
5	System.out.print(j + " ");				0
6	}				0
7	System.out.println("");				0
8	}				0
9	}				0
10	}				0

Displaying the complexity of a program due to all the factors

Line no	Program statements	Cs	Cv	Cm	Ci	Ccp	Ccs	TCps
1	class Pattern {	2	0	0	0	0	0	2
2	public static void main(String[] args) {	4	0	2	0	0	0	6
3	int rows = 5;	2	1	0	0	0	0	3
4	for(int i = 1; i <= rows; ++i) {	8	0	0	0	0	3	11
5	for(int j = 1; j <= i; ++j) {	8	0	0	0	0	6	14
5	System.out.print(j + " ");	8	0	0	0	2	0	10
6	}	0	0	0	0	0	0	0
7	System.out.println("");	6	0	0	0	2	0	8
8	}	0	0	0	0	0	0	0
9	}	0	0	0	0	0	0	0
10	}	0	0	0	0	0	0	0
Total		**38**	**1**	**2**	**0**	**4**	**9**	**5**