ASSESSMENT OF SOFTWARE UNDERSTANDABILITY USING SOFTWARE MATRICES

AHMER MANZOOR MAKAYA[1], SHAMINDER SINGH[2]
[1]*M.TECH Student, Desh Bhagat University Mandi Gobindgarh*
[2]*Assistant Professor, Desh Bhagat University Mandi Gobindgarh*

(E-mail: ahmermakaya@gmail.com)

Abstract**—**Understandability is one of the important characteristics of software quality, because it may influence the maintainability of the software. Cost and reuse of the software is also affected by understandability. In order to maintain the software, the programmers need to understand the source code. The understandability of the source code depends upon the psychological complexity of the software, and it requires cognitive abilities to understand the source code. The understandability of source code is get effected by so many factors, here we have taken different factors in an integrated view. In this we have chosen rough set approach to calculate the understandability based on outlier detection. Generally the outlier is having an abnormal behavior, here we have taken that project has may be easily understandable or difficult to understand. Here we have taken few factors, which affect understandability, an brings forward an integrated view to determine understandability.

Keywords**—***Understandability, Roughset, Outlier, Spatial Complexity.*

I. INTRODUCTION

Software products are expensive. Therefore, software project managers are always worried about the high cost of software development, and are desperately looking for way-outs to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have highquality.

When programmers try to reuse code which are written by other programmers, faults may occur due to misunderstanding of source code. The difficulty of understanding limits the reuse technique. On Software Development Life Cycle (SDLC) the maintenance phase tends to have a comparatively much longer duration than all the previous phases taken together, obviously resulting in much more effort. It has been reported that the amount of effort spent on maintenance phase is 65% to 75% [5]of total software development.

In Figure1,the programmers of the original system were absent, then the other programmers need to reuse the components to enhance the functionalities and correcting faults[16]. Fig.1 shows the communication between programmers and software, in the evolution of software systems. Programmer 1 writes the current version of a software system, programmer 2 evolves next version of that software from the current version[14]. If it is difficult to understand, changes to it may cause serious faults, these changes may cost more time than remaking the software systems. However, it is not easy to measure software understandability because understanding is an internal process of humans.

Basic etc. supports the concept of reusability. Reuse of the something that already existed is always nice rather than to create the same all over again. Reusability feature may increase the reliability, decrease the cost and time.

There are the many aspects of the software. Some of them contribute towards the design and algorithmic complexity, some contribute towards readability and understandability of the software, and some other aspects have an influence on the debugging and testability of the software. Developers should look more into writing code for not just as instructions to a computer, but as a medium of communication with other programmers. The time taken for a human to understand code is significantly longer than the time taken from a computer to compile and run a piece of software. Writing code that is more comprehensible by other developers should be emphasized. Software Maintenance Software maintenance[18] is becoming an important activity of a large number of organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Types of Software Maintenance Software maintenance can be required for three main reasons as follows:

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support the new features that users want to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Now a days software maintenance is associated with the problem is very expensive than what it should be and takes more time than required to work.

II. LITERATURE REVIEW

A software metric is a measure of some property of a piece of software or its specifications. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development.

Understandability of software also requires few metrics. Here few metrics of code understandability[3] are explained which are used by many organizations. Source code readability, quality of documentation, should be taken into account while measuring the software maintainability.

LOC: A common basis of estimate on a software project is the LOC(Lines of Code). LOC are used to create time and cost estimates.

Comment percent: RSM(Resource Standard Metrics) counts each comment line. The degree of commenting within the source code measures the care taken by the programmer to make the source code and algorithms understandable. Poorly commented code makes the maintenance phase of the software life cycle an extremely expensive.

In addition to the LOC (Lines Of Code), we may consider eLOC (Effective LinesOf Code), lLOC(Logical Lines Of Code), Blank lines of code and White Space Percent metric areused.

LEN* Length of names: If the names of procedures, variables, constants etc are long, then the more descriptive they probably are.

Example : ‘a ’is not good variable name, ‘age’ is better, ‘employee age’ is much more descriptive.

In addition to length of names, sometimes we may consider average length of names of the variables, functions, constants etc are considered. we may consider Name Uniqueness Ratio also, because when 2 program entities have the same name, it’s possible that they get mixed. UNIQ measures the uniqueness of all names.

UNIQ = Number of unique names /total number of names

Function Metrics: In this we can measure the number of functions and the lines of code per function. Functions that have a larger number of lines of code per function are difficult to comprehend and maintain. They are a good indicator that the function could be broken into sub functions whereby supporting the design concept that a function should perform a singular discrete action.

Function Count Metric: The total number of functions within your source code determines the degree of modularity of the system. This metric is used to quantify the average number of LOC per function, maximum LOC per function and the minimum LOC per function. In addition to the function count, we may consider Average lines of code, maximum LOC per function, minimum LOC per function metrics are also used.

Macro Metrics [2]: Macro will make your less understandable and difficult to maintain. As macros are expanded prior to the compilation step, most debuggers will only see the macro name and have no context as to the contents of the macro, therefore if the macro is the source of a bug in the system, the debugger will never catch it. This condition can waste many hours of labor.

The number of macros used in a system indicates the design style used to construct the system. Systems heavily laden with macros are subject to portability problems. The macro LOC metric yields insight in to how large macros are in the system. The larger the macro, them ore complex its structure and the greater the probability for erroneous behavior to be hidden by the macro.

Class Metrics: In this we can measure the number of classes and the lines of code per class can be taken. The number of classes in a system indicates the degree of object orientation of the system. In addition to this, we determine the average lines of code per class, maximum LOC per class and minimum LOC per class.

Code and Data Spatial Complexity [7]: Spatial ability is a term that is used to refer to an individuals cognitive abilities relating to orientation, the location of objects in space, and the processing of location related visual information.

Every software consists of two parts: code and data. To understand the behavior of any software, one needs to comprehend both of these entities. The programs code helps in understanding the processing logic and the data variables and constants help in recognizing the input and output of the software. The spatial complexity based on the code is dependent on the definition and use of various components of the software.

Code spatial complexity: To compute the code-spatial complexity, the module is considered as the basic unit, as every module is defined at one place, but is called many times. The functionality of the module is visible in the definition part, while the use of that module is understood through various calls of that module. The processing details of software can be understood by interrelating the definition of every module with its corresponding uses.

code-spatial complexity of a module (MCSC) is defined as average of distances (in terms of lines of code) between the use and definition of the module. Many a times, the software is written using multiple source-code files. Then an attribute may be defined in one file and used in some other file.In that case, the above definition of distance will be incomplete. When an attribute is used for the first time in a file, where it is not defined, the programmer first tries to find that class and attribute at the starting of the current file, because classes are usually declared at the start of a file. If the definition is not present in the current file, the programmer tries to find the details of that class and attribute in the other file. In that case, understanding of such use takes more cognitive effort. The effort is dependent on the file in which the attribute is being used, and the file in which it is defined.

Class method spatial complexity: Every class consists of many methods, A method basically means a function/subroutine in any language containing some processing steps. The purpose and functionality of the class can be better understood, if all methods of the class are defined close to the class declaration.

The distance can be easily computed as long as the method declaration and definition belong to the same file. But some times source code of the software is written in multiple files, and a method is declared in one file and defined in some other file. Then the programmer first tries to find that class in the current file. If it is not present in that file, he looks for that class declaration in the other file.

Distance = (distance of definition from the top of the file containing definition)+(distance of declaration of the method from the top of the file containing declaration).

Total class method spatial complexity (TCMSC) of a class is defined as average of class method spatial complexity of all methods of the class. As the class is an encapsulation of attributes and methods, the class spatial complexity is an integration of both types of spatial complexities, and hence the class spatial complexity (CSC) of a class is proposed as

$$CSC = TCASC + TCMSC$$

This measure of class spatial complexity depends only on intra-properties of the class. In a way this measure helps in measurement of the understandability and cohesiveness of the class from the point of view of cognitive abilities. This measure does not take care of the possible use of that class in the form of objects, which ultimately interact with each other for achieving the complete functionality of the object-oriented software. The spatial complexity generated because of the various objects is measured in the form of object spatial complexity.

Object Spatial Complexity: The Object-Oriented software works with the help of ob- jects and their interactions. Different methods of the class are called through the objects in a specific sequence so as to obtain the proper results from the software. The object spatial complexity is of two types: Object definition spatial complexity and Object-member usage spatial complexity.

Object definition spatial complexity: As soon as an object is defined, the programmer needs to establish the relation of this object with the corresponding class. This cognitive effort will depend upon the distance of the object definition from the corresponding class declarations.

If an object is defined immediately after its class declaration, it will take almost no effort to comprehend the purpose of the object, as the details of the corresponding class will be present in the working memory of the person. If the object is defined in the same source- code file where the corresponding class has been declared, the distance can be calculated as above; but if the Object-Oriented software is written using multiple source code files, and the object is defined in a different file than the file containing class declaration, the effort is dependent on two files, as already discussed.

In that case, the distance for that particular object is defined as:

Distance = (distance of object definition from top of current file)+ (distance of declaration of the corresponding class from the top of the file containing class)

Object-member usage spatial complexity[8]: Once the objects are defined, they keep on calling various members (methods mostly, but attributes also may be referred sometimes). If an object-member is called after a long distance from its definition, spatial abilities needed will be much more. Thus, the object-member usage spatial complexity (OMUSC) of a member through a particular object is defined as the average of distances (in terms of lines of code) between the call of that member through the object and definition of the member in the corresponding classDistance is equal to the absolute difference in number of lines between the method definition and the corresponding call/use through that object.

The OMUSC measure concentrates on the usage of the classes through objects, which do interact with other processing blocks (such as main) and other classes (in which the object ofanotherclassmayhavebeendefined).Justlikepreviouscases,incaseoftwofilescoming in to picture for measurement of this distance, the distance is defined as

Distance = (distance of object definition from top of current file)+ (distance of declaration of the corresponding class from the top of the file containing class)

Total object-member usage spatial complexity (TOMUSC) of an object is defined as average of object-member usage spatial complexity of all members being used of that method. where k is the count of object-members being called through that object. Based on the above formulas, the objects patial complexity of an object is defined as

OSC = ODSC + TOMUSC

This measure of object spatial complexity depends on the inter-usage of the classes within the routines or other classes of the object-oriented software. It may be noted that this measure inherently takes care of the effect of inheritance and polymorphism towards understandability of the software.

A. *Object-Oriented Metrics*

Very few metrics have been proposed for object -oriented software systems

The Chidamber and Kemerer Metrics (C&K) Suite: The Chidamber and Kemerer Metrics (C&K)[26][12] suite includes the following metrics: Weighted methods per class (WMC): WMC is a measure of number of methods implemented within a class. This metric measures understandability, maintaina-bility, and reusability as follows:

- The number of methods in a class reflects the time and effort required to develop and maintain the class.
- The larger the number of methods, the greater the potential impact on children, since children inherit all of the methods defined in a class.
- A class with a large number of methods is more application-specific, and therefore is not likely to be reused.

Depth of Inheritance Tree (DIT): DIT is the maximum length from the class node to the root of the tree. It is measured by the number of ancestor classes. This metric measures understandability, reusability, and testability as follows:

- The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit. This makes the deep class more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper the inheritance tree is, the more the potential for reuse.

Number of Children (NOC): NOC is the number of immediate subclasses of a class in the hierarchy. This is an indicator of the potential influence a class can have on the design and on the system hierarchy. This metric measures efficiency, reusability, and testability as follows:

- The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of sub-classing.
- The greater the number of children, the greater the reusability since inheritance is a form of reuse.
- If a class has a large number of children, it may require more testing of the methods of that class ,thus increase the testing time.

Lack of Cohesion in Methods (LCOM): This metric evaluates efficiency and reusability. Here we are not considering this metric for understandability.

Coupling Between Objects (CBO): CBO is a count of the number of other classes to which a class is coupled. CBO is measured by counting the number of distinct non- inheritance related class hierarchies on which a class depends.

- The higher the coupling the more sensitive the system is to changes in other parts of the design, and therefore maintenance is more difficult.
- High coupling also reduces the systems understandability because it makes the mod- ule harder to understand, change, or correct by itself if it is interrelated with other modules.

Response For a Class (RFC): RFC is the number of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This measures the amount of communication with other classes.

- The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class.
- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated as it requires a greater level of understanding on the part of the developer.
- This metric evaluates understandability, maintainability, and testability.

The Lorenz and Kidd Metrics suite Unlike C & K metrics the most of the L & K metrics[21] are directly measures are include directly countable measures. Those metrics are:

Number of Public Methods: This simply counts the number of public methods with in the class. According to L & K this metric is useful to estimate the amount work done to develop a class or subsystem.

Number of Methods: The total number of methods with in the class counts all private, public, and protected methods defined.

Number of Public variables per class: This metric counts number of public variables with in the class. L & K consider the number of variables in a class is to be one measure of its size. The fact that if one class is having more number of public variables then that class has more relationship with other objects and as such it is more likely to be key class.

Number of Variables per Class: This metric includes the total number of variables with in the class. This includes all public, private and protected variables. According to L & K the total number of private and protected variables to the total number variables indicates the effort required by that class in providing information to other classes. Private and protected variables are therefore viewed as data to service the methods in the class.

Number of Methods Inherited by Subclass: This metric can measures the number of methods inherited by subclasses.

Number of Methods Overridden by subclass: A subclass is allowed to re-define or over ride the methods in one of its super class. According to L & K a large number of overridden methods are indicates the design problem.

Number of Methods added by Subclass: A method is defined as an added method in a subclass,if there no method of the same name in any of its super classes. According to L & K normal expectation is that for subclass it will further specializes or adds the methods to the super class object.

Average Method Size: The average method size is calculated as the number of non commented lines and non blank source lines in the class divided by the number of methods in that class. This is clearly a size metric.

Number of Times a class is reused: The definition given by the L & K for this metric is ambiguous. This metric is intended to count the number of times a class is referenced by other classes. This is similar to coupling, so high reusability is undesirable according to coupling definition.

Number of Friends of a class: This metric is especially for C++, by using this metric we can count the number of friends of that class. This metric is also one type of measure for coupling.

Abreu Metrics :The emphasis behind the development of the metrics is on the features of inheritance, encapsulation and coupling. The six Abreu Metrics[21] can be summarized as

Polymorphism Factor : This metrics is based on the number of overriding methods in a class as a ratio of total possible overridden methods. Polymorphism arises from inheritance, Abreu claims that some times overriding reduces the complexity, so it may increases the understandability.

Coupling Factor: This metric counts the number of inter class communications. Here there is a similarity with the number of classes reused metric according to L & K. Abreu views coupling increases the complexity and limiting the understandability.

Method hiding factor: This metric is the ratio of hidden(private & protected) meth- ods to the total number of methods.

Attribute Hiding Factor: This metric is the ratio of hidden(private & protected) attributes to the total number of attributes.

Method Inheritance Factor: This metric is a count of number of inherited methods as a ratio of total methods, Abreu proposes that it expressing the level of reuse in a system.

Attribute Inheritance Factor: This metric is a count of number of inherited at- tributes as a ratio of total attributes, Abreu proposes that it expressing the level of reuse in a system.

III. Effect of Spatial Complexity on Object-Oriented Programs

By considering all the metrics, most of the research already done in software complexity. In addition to despite of all the metrics, we want to explore our work on the spatial complexity. The researchers already did in class spatial complexity and object spatial complexity. In these days object-oriented programming is used by programmers in their projects to get the benefit of reusability, so here, we considered the effect of inheritance in the spatial complexity.

A. Effect of Inheritance

Fortunately, C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating by new classes, reusing the properties of the existing ones.

The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass. The derived class inherits some or all of the traits from the base class. Inheritance is the one of object-oriented features, which helps in reusability. Inheritance can be implemented in different combinations.

B. Spatial Complexity of Derived Classes

Aclasscanalsoinheritpropertiesfrommorethanoneclassorfrommorethanonelevel.So, here we concentrating on the spatial complexity between the classes. According to previous researchers they focused mainly on particular class spatial complexity. Here we focused on the number of classes derived from the base class, and the number of methods & attributes inherited by that class.

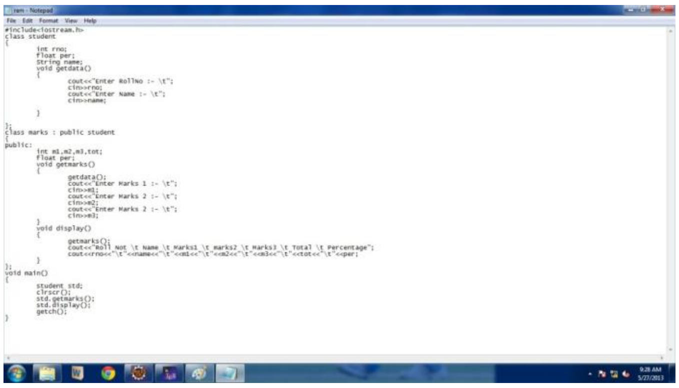
Single Inheritance: The spatial complexity of the derived class is different from the spatial complexity of base class. Here, while calculating the derived class spatial complexity (DCSC), we consider the attributes and methods which inherit from the base class.

Attribute Spatial Complexity: While calculating the derived class spatial complexity, we need to consider the attributes and methods of that derived class. Attribute spatial complexity of a derived class can be calculated as

CASC is the Class Attribute Spatial Complexity and CIASC is the Class Inherited Attribute Spatial Complexity.

Distance is measured in Lines Of Code(LOC) in between the successive use of that variable or definition to the use of that variable. Example: The program which is in Fig 1.1 is the example program which illustrates the single inheritance.

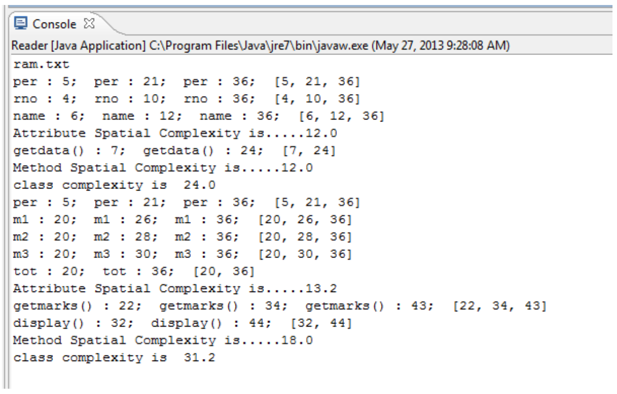
Method Spatial Complexity In the case of method spatial complexity, we need to consider all the methods of its own class and the inherited methods. DCMSC(Derived



```
#include<iostream.h>
class student
{
        int rno;
        float per;
        String name;
        void getdata()
        {
                cout<<"Enter RollNo :- \t";
                cin>>rno;
                cout<<"Enter Name :- \t";
                cin>>name;

        }
};
class marks : public student
{
public:
        int m1,m2,m3,tot;
        float per;
        void getmarks()
        {
                getdata();
                cout<<"Enter Marks 1 :- \t";
                cin>>m1;
                cout<<"Enter Marks 2 :- \t";
                cin>>m2;
                cout<<"Enter Marks 2 :- \t";
                cin>>m3;
        }
        void display()
        {
                getmarks();
                cout<<"Roll Not \t Name \t Marks1 \t marks2 \t Marks3 \t Total \t Percentage";
                cout<<rno<<"\t"<<name<<"\t"<<m1<<"\t"<<m2<<"\t"<<m3<<"\t"<<tot<<"\t"<<per;
        }
};
void main()
{
        student std;
        clrscr();
        std.getmarks();
        std.display();
        getch();
}
```

Figure 1.1: Example program for single inheritance



Console

Reader [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (May 27, 2013 9:28:08 AM)

```
ram.txt
per : 5;  per : 21;  per : 36;  [5, 21, 36]
rno : 4;  rno : 10;  rno : 36;  [4, 10, 36]
name : 6;  name : 12;  name : 36;  [6, 12, 36]
Attribute Spatial Complexity is.....12.0
getdata() : 7;  getdata() : 24;  [7, 24]
Method Spatial Complexity is.....12.0
class complexity is  24.0
per : 5;  per : 21;  per : 36;  [5, 21, 36]
m1 : 20;  m1 : 26;  m1 : 36;  [20, 26, 36]
m2 : 20;  m2 : 28;  m2 : 36;  [20, 28, 36]
m3 : 20;  m3 : 30;  m3 : 36;  [20, 30, 36]
tot : 20;  tot : 36;  [20, 36]
Attribute Spatial Complexity is.....13.2
getmarks() : 22;  getmarks() : 34;  getmarks() : 43;  [22, 34, 43]
display() : 32;  display() : 44;  [32, 44]
Method Spatial Complexity is.....18.0
class complexity is  31.2
```

Figure 1.2: Class Spatial Complexity of the above single inheritance program in Fig. 1.1

CMSC can also be calculated similar to CASC by using The total derived class spatial complexity can be defined as the sum class attribute spatial complexity and class method spatial complexity.

DCSC=DCASC+DCMSC .... (1.1)

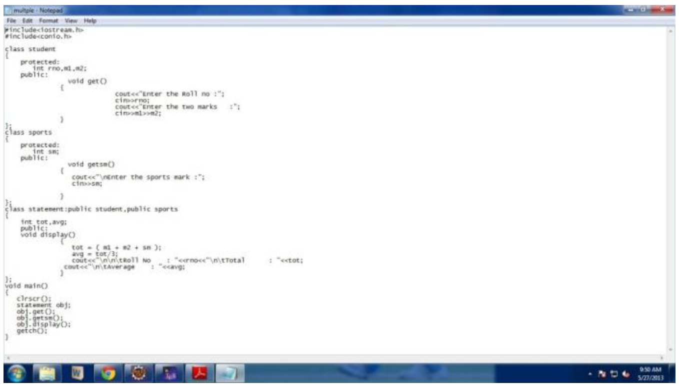
So, the spatial complexity involved in single inheritance is given by

CSC Single Inheritance = DCSC+CSC Base Class ...(1.2)

C. Multiple Inheritance

In case of multiple inheritance, the new class is derived from multiple base classes. While calculating the spatial complexity of a derived class, we need to consider all the base class spatial complexities.

Class Attribute Spatial Complexity While calculating the derived class attribute spatial complexity, we need to consider the attributes and methods of that derived class.



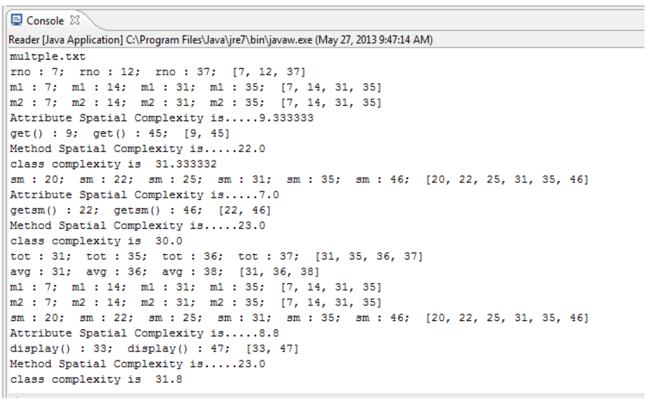
```
#include<iostream.h>
#include<conio.h>

class student
{
    protected:
        int rno,m1,m2;
    public:
              void get()
            {
                         cout<<"Enter the Roll no :";
                         cin>>rno;
                         cout<<"Enter the two marks   :";
                         cin>>m1>>m2;
            }
};
class sports
{
    protected:
        int sm;
    public:
              void getsm()
            {
              cout<<"\nEnter the sports mark :";
              cin>>sm;

            }
};
class statement:public student,public sports
{
    int tot,avg;
    public:
    void display()
            {
              tot = ( m1 + m2 + sm );
              avg = tot/3;
              cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal      : "<<tot;
            cout<<"\n\tAverage    : "<<avg;
            }
};
void main()
{
   clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

Figure 1.3: Example program for multiple inheritance.

Example: The program which is in Fig 1.3 is the example program which illustrates the multiple inheritance.



Console

Reader [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (May 27, 2013 9:47:14 AM)

```
multple.txt
rno : 7;  rno : 12;  rno : 37;  [7, 12, 37]
m1 : 7;  m1 : 14;  m1 : 31;  m1 : 35;  [7, 14, 31, 35]
m2 : 7;  m2 : 14;  m2 : 31;  m2 : 35;  [7, 14, 31, 35]
Attribute Spatial Complexity is.....9.333333
get() : 9;  get() : 45;  [9, 45]
Method Spatial Complexity is.....22.0
class complexity is  31.333332
sm : 20;  sm : 22;  sm : 25;  sm : 31;  sm : 35;  sm : 46;  [20, 22, 25, 31, 35, 46]
Attribute Spatial Complexity is.....7.0
getsm() : 22;  getsm() : 46;  [22, 46]
Method Spatial Complexity is.....23.0
class complexity is  30.0
tot : 31;  tot : 35;  tot : 36;  tot : 37;  [31, 35, 36, 37]
avg : 31;  avg : 36;  avg : 38;  [31, 36, 38]
m1 : 7;  m1 : 14;  m1 : 31;  m1 : 35;  [7, 14, 31, 35]
m2 : 7;  m2 : 14;  m2 : 31;  m2 : 35;  [7, 14, 31, 35]
sm : 20;  sm : 22;  sm : 25;  sm : 31;  sm : 35;  sm : 46;  [20, 22, 25, 31, 35, 46]
Attribute Spatial Complexity is.....8.8
display() : 33;  display() : 47;  [33, 47]
Method Spatial Complexity is.....23.0
class complexity is  31.8
```

.

Figure 1.4: Class Spatial Complexity of the above multiple inheritance program in Fig. 1.3

Class Method Spatial Complexity In the case of method spatial complexity, we need to consider all the methods of its own class and the inherited methods from the parent/base class.

Now, the derived class spatial complexity can be written as

DCSC=DCASC+DCMSC ....(1.3)

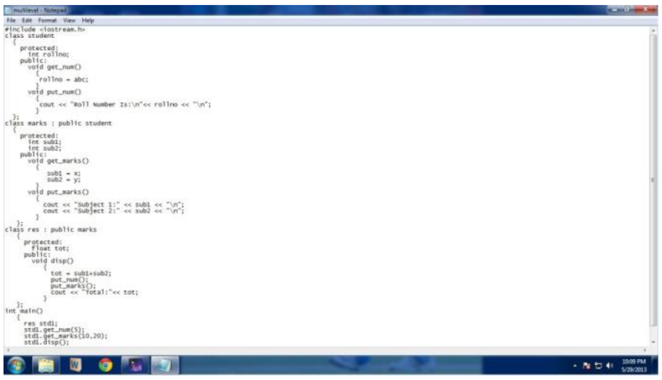
But, in multiple inheritance, we need to consider all parent/base class spatial complexities also. Because while considering the inherited attributes/methods, we need to calculate, the spatial complexities of all members.

D. *Multilevel Inheritance*

In the case of multilevel inheritance, a derived class is derived from another derived class. Here, we have to consider the different levels of parent class from which different attributes and methods are inherited by derived classes.

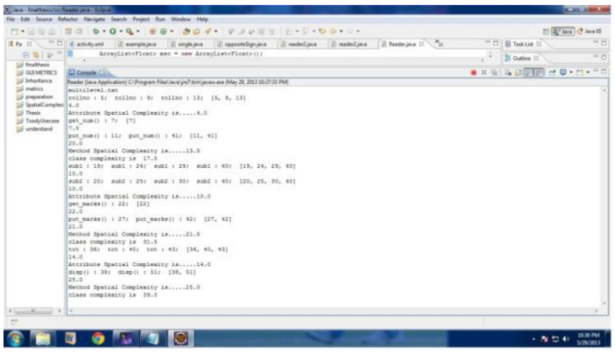
Class Attribute Spatial Complexity Let us consider that the level of inheritance is started from 1st to lth level.

Class Method Spatial Complexity In the multilevel inheritance, in addition to the attributes, different methods are also inherited from the parent/base classes which are at different levels. Let us consider that the level of inheritance is started from 1st to lth level.



```
#include <iostream.h>
class student
  {
    protected:
      int rollno;
    public:
      void get_num()
        {
         rollno = abc;
        }
      void put_num()
        {
         cout << "Roll Number Is:\n"<< rollno << "\n";
        }
  };
class marks : public student
  {
    protected:
      int sub1;
      int sub2;
    public:
      void get_marks()
        {
          sub1 = x;
          sub2 = y;
        }
      void put_marks()
        {
          cout << "Subject 1:" << sub1 << "\n";
          cout << "Subject 2:" << sub2 << "\n";
        }
   };
class res : public marks
   {
     protected:
       float tot;
     public:
       void disp()
         {
           tot = sub1+sub2;
           put_num();
           put_marks();
           cout << "Total:"<< tot;
         }
   };
int main()
   {
     res std1;
     std1.get_num(5);
     std1.get_marks(10,20);
     std1.disp();
```

Figure 1.5: Example program for multi-level inheritance program.



```
Reader [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (May 29, 2013 10:27:33 PM)
multilevel.txt
rollno : 5;  rollno : 9;  rollno : 13;  [5, 9, 13]
4.0
Attribute Spatial Complexity is.....4.0
get_num() : 7;  [7]
7.0
put_num() : 11;  put_num() : 41;  [11, 41]
20.0
Method Spatial Complexity is.....13.5
class complexity is  17.5
sub1 : 19;  sub1 : 24;  sub1 : 29;  sub1 : 40;  [19, 24, 29, 40]
10.0
sub2 : 20;  sub2 : 25;  sub2 : 30;  sub2 : 40;  [20, 25, 30, 40]
10.0
Attribute Spatial Complexity is.....10.0
get_marks() : 22;  [22]
22.0
put_marks() : 27;  put_marks() : 42;  [27, 42]
21.0
Method Spatial Complexity is.....21.5
class complexity is  31.5
tot : 36;  tot : 40;  tot : 43;  [36, 40, 43]
14.0
Attribute Spatial Complexity is.....14.0
disp() : 38;  disp() : 51;  [38, 51]
25.0
Method Spatial Complexity is.....25.0
class complexity is  39.0
```

Figure 1.6: Class Spatial Complexity of the above multi-level inheritance program in Fig 1.5

Where l is the level of inheritance, n is the number of methods of its own class, m is the number of methods derived from its base class. Then the derived class spatial complexity in multilevel inheritance can be calculated as

DCSC = DCASC +DCMSC ....(5.4)

E. *Spatial complexity metric analysis with Weyuker's Properties*

Weyuker proposed a formal list of nine properties which are used to evaluate a software complexity metric. But it is not necessary to satisfy all the properties by a single metric. Here, the spatial complexity of the derived class is evaluated by using Weyuker's nine properties. While describing these properties, X denotes an Object-Oriented Program/Class by default and |X| represents its complexity, which will always be a non-negative number.

Property 1: This property states that (∃ X), (∃ Y) such that (|X| /= |Y |)

Two programs X and Y can always differ in values of derived class spatial complexity measures, because these measures are defined in terms of distance LOC (Lines Of Code), which will have most of the times different values for two different programs. Thus, object- oriented spatial measures satisfy Property 1.

Property 2: Let c be a non-negative number, and then there are only a finite number of programs of complexity c.

This property is a strengthening of Property 1. Derived class spatial complexity can be calculated by using the class attribute spatial complexity, and class method spatial complexity. Class attributes spatial complexity itself can be calculated by considering the number of inherited attributes, and attributes of its own class. Similar to this, class method spatialcomplexitycanalsobecalculatedbyusingthedifferentnumb erofinheritedmethods and methods of its own class. There can be always a finite number of programs having the same value of these factors and thus, Property 2 is well satisfied with the object-oriented spatial complexity measures.

Property 3: This property states that (∃ X), (∃ Y) such that (|X| = |Y |)

This property states that a complexity measure must not be too fine, i.e. any specific value of this metric should not only given by a single program. This property requires that derived class spatial complexities of two different classes may be equal, i.e DCSCX
= DCSCY , where DCSCX , DCSCY are the derived class spatial complexities for two
different classes X,Y. This is quite possible that two different and totally unrelated Object- Oriented programs X and Y may come out with the same spatial complexity values. Thus, derived spatial complexity measures satisfies the Property 3.

Property 4: This property states that (X ≡ Y &(|X| /= |Y|)
This property states that , if two programs are equal in same functionality may differ in spatial complexities. Because, spatial complexity depends on implementation of that functionality. Two object-oriented programs X and Y of the same functionality but different implementations will have different spatial complexities.
i.e. DCSCX /= DCSCY for two object-oriented programs X & Y even though X=Y.
Property5: (∀X)(∀Y)(|X|≤|X;Y| and |Y|≤|X;Y|)

This property explains the concept of monotonicity with respect to composition. This property explains that the spatial complexity of concatenated programs which are obtained from the concatenation of two programs can never be less than the spatial complexity of either of the programs. Let DCSCX and DCSCY be the derived class spatial complexities of two object-oriented programs X and Y respectively and DCSCXY be the derived class spatial complexity of the concatenated program of X and Y.
According to the definition of derived class spatial complexity, the resulting derived class spatial complexity of the combined program would be approximately sum of the class spatial complexities two individual programs. If they were independent and if the code of program X, without disturbing the individual distances of definitions and usage of members of the class i.e. DCSCXY∼=DCSCX+DCSCY If the programs X and Y were not in dependent, then the common classes may appear once in concatenated program . In that case the class spatial complexity of common portion will contribute once in the measures and independent portions of both X and Y will continue to have their original contribution to wards DCSC.
In that situation
DCSCXY = DCSCX + DCSCY – CSC common−class

Property 6
1. (∃ X) (∃ Y)(∃ Z) (|X| = |Y |) & (|X; Z||Y ; Z|)
2. (∃ X) (∃ Y)(∃ Z) (|X| = |Y |) &(|Z;X||Z;Y|)
As already stated in property 3, object-oriented programs having different implementations may have the same values for object-oriented spatial complexity measures. When these two different programs having equal spatial complexity are combined with the same program, this may result into different spatial complexities for the two different combinations.
This means,
(∃ X) (∃ Y)(∃ Z) (DCSCX = DCSCY) & (DCSCZX /= DCSCZY)
Thus, property 6a and 6b are well satisfied with the object-oriented spatial complexity measures.
Property 7: This property says that there are two programs X and Y such that Y is formed by permuting the order of the statements of X, and |X|/=
|Y |, that means a complexity measure should be sensitive to the permutation of statements.
The object-oriented spatial complexity measures are mainly defined in terms of distances in lines of code between uses of different program elements such as class-members (at-tributes/methods). Thus, the spatial complexity of an object-oriented program depends on the order of the statements of the program. When program Y is formed by permuting the order of statements of the program X, the spatial complexity measures of program Y will also change from values obtained from the program X due to change in lines of code between
program elements. Thus, for programs X and Y,DCSCX DCSCY where program Y is formed by permuting the order of the statements of X. Hence, the object-oriented spatial complexity measures satisfy property 7.
Property 8: If X is a renaming of Y, then |X| = |Y|. This property states that by changing the name of the program or its elements. does not affect the spatial complexities of object-

oriented program. i.e DCSCx = DCSCy where X is renaming of Q. Thus, Property 8 is satisfied with the object-oriented measures.

Property 9: (∃X) (∃Y)such that (|X| + |Y | <|X; Y |) According to this property, the spatial complexity of a new program obtained from the combinations of two programs, can be greater than the sum of spatial complexities of two individual programs.

VI. CONCLUSION

Software understandability affects quality of overall software engineering. If software understandability is favorable, software development process can be mastered definitely. In this work, we considered so many different types of metrics. But, we want to focus few more metrics in our further research. We used a rough set approach to detect the project which is having abnormal behavior. This type of behavior tells us that the particular project is either easily understandable or very much difficult to understand. The algorithm which is used by us is having less time complexity than fuzzy based approach. In our further work we want to include threshold values which have been calculated based on the standard values of different attributes, based on that threshold value we will give outlier ranking.

Furthermore, we have considered the affect of spatial complexity metric on object-oriented programming features like inheritance. Here, we have calculated the spatial complexity of different derived classes, which are involved in different types of inheritance. In further, we want to explore this spatial complexity to templates, macros etc.

References

[1] Rough set. Website. http://en.wikipedia.org/wiki/Roughset.

[2] Rsm metrics. Website. http://msquaredtechnologies.com/m2rsm/docs/rsmmetrics.

[3] Understandability metrics. Website.http:www.aivosto.com/project/help.

[4] Using uml part two- behavioral modelling diagrams. Website.http://www.sparxsystems.com.

[5] Krishan K. Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. pages 235–240, GGS Indraprastha University, Delhi and Regional Engineering College, Kurukshetra, 2012. 2012 PROCEEDINGS Annual RELIABILITY and MAINTAIN ABILITY Symposium.

[6] Richard H. Carver, Steve Counsell, and Reuben V. Nithi. An evaluation of the mood setofobject-oriented software metrics. IEEETrans. Software Eng.,24(6):491–496, 1998.

[7] Jitender Kumar Chhabra, K. K. Aggarwal, and Yogesh Singh. Code and data spa- tial complexity: two important software understandability measures. Information &Software Technology, 45(8):539–546,2013.

[8] Jitender Kumar Chhabra, K. K. Aggarwal, and Yogesh Singh. Measurement of object- oriented software spatial complexity. Information & Software Technology,46(10):689– 699,2044.

[9] Jitender Kumar Chhabra and VarunGupta. Evaluation of object-oriented spatial complexity measures.ACMSIGSOFT Software Engineering Notes,34(3):1–5,2019.

[10] Nicolas E. Gold, Andrew Mohan, and Paul J. Layzell. Spatial complexity metrics: An investigation of utility. IEEE Trans. Software Eng., 31(3):203–212,2015.

[11] Maurice H.Halstead. Elementsof Software Science.ISBN:0-444-00205-7.Amsterdam, 1977.

[12] Seyyed Mohsen Jamali. Object oriented metrics. Department of Computer Engineering Sharif University of Technology,2016.

[13] Feng Jiang, Yuefei Sui, and Cungen Cao. A rough set approach to outlier detection. volume 37, pages 519–536. International Journal of General Systems, october2018.

[14] K.Shima, Y.Takemura, and K.Matsumoto. An approach to experimental evaluation of software understandability. Proceedings of the 2012 International Symposiumon Empirical Software Engineering (ISESE'12),2012.

[15] Xiangjun LI and Fen RAO. An rough entropy based approach to outlier detection. Journal of Computational Information Systems, pages 10501–10508, 2012. Department of Computer science and Technology, Nanchang University,Nanchang 330031, China and College of Economy and Management, Nanchang University,Nanchang 330031, China.

[16] Jin-Cherng Lin and Kuo-Chiang Wu. A model for measuring software understandabil- ity. In CIT, page 192,2016.

[17] Jin-Cherng Lin and Kuo-Chiang Wu. Evaluation of software understandability based on fuzzy matrix. In FUZZ-IEEE, pages 887–892,2018.

[18] Rajib Mall. Fundamentals of Software Engineering. Prentice Hall, 3rd edition,2009.

[19] Sanjay Misra and A. K. Misra. Evaluating cognitive complexity measure with weyuker properties. In IEEE ICCI, pages 103–108,2014.

[20] Yuto Nakamura, Kazunori Sakamoto, Kiyohisa Inoue, Hironori Washizaki, and Yoshi- aki Fukazawa.Evaluation of understandability of uml class diagrams by using word similarity. In IWSM/Mensura, pages 178–187,2011.

[21] R.Horrison, S.Counsell, and R.Nithi. An overview of object-oriented design metrics. Dept. of Electronics and Computer Science,Southampton,1997.IEEE.

[22] S. W. A. Rizvi and R. A. Khan. Maintainability estimation model for object-oriented software in design phase (memood). CoRR, abs/1004.4447,2018.

[23] Patricia L. Roden, Shamsnaz Virani, Letha H. Etzkorn, and Sherri L. Messimer. An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes. In SCAM, pages 171–179,2017.

[24] Yingxu Wang and Jingqiu Shao. Measurement of the cognitive functional complexity of software. In IEEE ICCI, pages 67–74, 2013.

[25] Tong Yi, Fangjun Wu, and Chengzhi Gan. A comparison of metrics for uml class diagrams. ACMSIGSOFT Software Engineering Notes,29(5):1–6,2014.

[26] Aida Atef Zakaria and Dr. Hoda Hosny. Metrics for aspect-oriented software design. pages228–233.ACM press, 1975