Multi-Paradigm Metric and its Applicability on JAVA Projects

Sanjay Misra

Department of Computer Engineering, Atilim University
Kizilcaşar Mh., 06830 Incek, Ankara/Ankara Province, Ankara, Turkey
smisra@atilim.edu.tr

Ferid Cafer

BOTT Information Systems, Silicon Block No:20
Middle East Technical University Teknopolis, 06531, Ankara, Turkey
ferid.cafer@bott.com.tr

Ibahim Akman

Department of Computer Engineering, Atilim University
Kizilcaşar Mh., 06830 Incek, Ankara/Ankara Province, Ankara, Turkey
akman@atilim.edu.tr

Luis Fernandez-Sanz

Universidad de Alcalá, Depto. de Ciencias de la Computación
Plaza de San Diego, s/n, 28801 Alcalá de Henares, Madrid, Spain
luis.fernandezs@uah.es

Abstract: JAVA is one of the favorite languages amongst software developers. However, the numbers of specific software metrics to evaluate the JAVA code are limited. In this paper, we evaluate the applicability of a recently developed multi paradigm metric to JAVA projects. The experimentations show that the Multi paradigm metric is an effective measure for estimating the complexity of the JAVA code/projects, and therefore it can be used for controlling the quality of the projects. We have also evaluated the multi-paradigm metric against the principles of measurement theory.

Keywords: Multi-paradigm metric; Software complexity; JAVA; software development

1 Introduction

One of the important issues in the software development process is to maintain the quality of the software. Complex codes are not desirable because they are hard to maintain and reduce the quality of the software [1] [2]. The complex codes also decrease the understandability and increase the burden on reviewers, testers and maintainers. In this point of view, if the complexity is not controlled from the beginning of software development, it may cause higher maintainability and reduce the quality of the product. As a result, complex code increases the cost of software/software product. To overcome this issue, the complexity of the code should be controlled. Software metrics are the tools to control the complexity. Researchers are making continuous efforts to produce metrics to control the complexity of the code. Further, software metrics tend to compare various parameters such as cost, effort, time, maintenance, understanding and reliability. Metrics are indispensable from several aspects, such as measuring the understandability of a code, the testability of the software, the maintainability and the development processes [3].

Over last two decades, object oriented programming languages have gained considerable acceptance from the software development community. Among several object-oriented languages, JAVA has become a favorite language for developing software products. The popularity of JAVA has arisen as a consequence of its unique features. On the other hand, to evaluate the quality of the software code written in JAVA, few metrics [4] are available in the literature. We mention the effort of researchers [4-10] who tried to control the quality of JAVA by considering different aspects and features of JAVA programming. Dufour [4] proposed dynamic metrics for JAVA. Cahoon et al. [5] worked on data flow analysis for software perfecting linked data structures in JAVA controllers. Sudaresan et al. [6] researched practical virtual method call resolution for JAVA. Vijaykrishnan et al. [7] have produced tuning branch predictors to support virtual method invocation in JAVA. Qian et al. [8] proposed a comprehensive approach to array bounds check elimination for JAVA. Erik Ruf [9] proposed a methodology for the effective synchronization removal for JAVA. Shuf et al. [10] proposed a structured view and opportunities for optimizations by characterizing the memory behaviour of JAVA workloads. Mäkelä et al. [11] proposed a new client based metric, Lack of Coherence in clients (LCIC), and developed a tool for measuring the metric for JAVA projects. The authors tried to improve the quality of code through the LCIC metric, which measures how a class is used by other classes in a context. In their comparison analysis, the authors also suggested which type of refactoring is required. In an another empirical study of JAVA inheritance evaluation, Nasseri et al. [12] found that larger and highly coupled classes were less cohesive and more frequently moved than smaller and less coupled classes. Kaczmarek and Kucharski [13] demonstrated how to estimate size and efforts for JAVA based applications. They presented three models of size estimations, which

were based on class and method size. The authors concluded that for big projects, for example projects of nearly one million lines of code, the average class and method size will be independent from the application size. Giuseppe [14] proposed a semantic similarity metric which combines the features and intrinsic information content. Romano [15] proposed using source code metrics to predict change-prone JAVA interfaces. None of the above works evaluate the quality of the JAVA code, which is responsible for the understandability and therefore the maintainability of the JAVA product. It is worth mentioning that maintainability is identified as one of the most important software quality [16] attributes.

In this paper, we apply a recently proposed metric [17] that was developed for multi-paradigm languages on JAVA projects. A multi-paradigm language is a language which includes features of two or more than two programming paradigms. The metric developed in [17] considered procedural, object oriented paradigm, and combined the function point metric [18] to estimate complexity due to the functionality of the code/project and an object oriented metric [19] to estimate the complexity of object oriented features. The proposed multi-paradigm metric [17] was applied in a project written in C++. Since JAVA is also a multi paradigm language, which includes features of procedural and object-oriented language, we apply the same metric to evaluate JAVA projects. In fact, the agenda of the present paper is twofold. Firstly, we want to check the applicability of the multi-paradigm metric in JAVA projects; and secondly, we want to perform more experimentation for the empirical validation of the multi paradigm metric, given that the real applicability of a metric cannot be proved without a series of empirical observations [20]. Even more, we will evaluate the theoretical soundness of the multi paradigm metric by applying the principles of measurement theory to the multi paradigm metric.

Before moving ahead, we would like to summarize our previous works in this area. We have developed metrics for procedural languages [21], object oriented languages [19], [22], [23], and multi paradigm languages [17]. One of the coauthors of this paper is also involved in developing metrics for various purposes, e.g. web-services, [24], [25], SOA and XML schema languages[26], [27], Business Process Modeling[28], etc. Another coauthor has proposed a scheme for the verification and validation of JAVA code by combining code style check and some code metrics to prioritize test cases [29].

The structure of the paper is as follows. The definition of the multi paradigm metric is given in Section 2. In Section 3, we first evaluate the metric to check its soundness from the principles of measurement theory, and then we apply the metric on JAVA projects in Section 4. The conclusion of the work is in the last section.

2 Multi-Paradigm Complexity Measure Measurement

In order to compute the complexity of software system, the authors [17] have suggested how to compute the quality of the code by considering that the overall complexity of the software system depends on the functionality as well as on different factors of the object oriented and procedural parts of the system.

Accordingly, the computation of the quality of code for multi-paradigm programs is presented as,

Code Quality (CQ): The CQ is defined by the number of function points to the complexity values due to all the factors in the multi-paradigm program code.

CQ = (FP / MCM)* 10,000 (1)

where, FP [30] is the Function Point calculations for the code, and MCM represents the multi-paradigm complexity measurement and computes the complexity of the code as given in equation (2). MCM followed the similar approach of a metric developed for python [13].

$$MCM = CIclass + CDclass + Cprocedural \quad (2)$$

where **CIclass** = Complexity of Inherited Classes,

CDclass = Complexity of Distinct Class,

and **Cprocedural =** Procedural Complexity.

All these factors are defined as follows:

The complexity of an independent class is calculated first because it plays a role either in the inheritance hierarchy or as a distinct class. In other words, for calculating CIclass or CDclass, first it is necessary to calculate Cclass, the complexity of an independent class. The complexity (Cclass) of an independent class can be computed as:

$$Cclass = W(attributes) + W(variables) + W(structures) + W(objects) - W(cohesion) \quad (3.1),$$

where **Cclass** represents the Complexity of a single class.

In the above formula, the weight due to cohesion is subtracted because it reduces the complexity and is desirable from the point of view of software developers [1].

The weight of attributes or variables is computed as:

$$W(variables\ or\ attributes) = 4 * AND + MND \quad (3.1.1)$$

where AND represents the Number of Arbitrarily Named Distinct Variables/Attributes, and

MND represents the Number of Meaningfully Named Distinct Variables/Attributes.

Weight of structure: W (structures) is defined as the weight of structure of the methods inside the class:

$$W(structures) = W(BCS) \tag{3.1.2}$$

where BCS are basic control structures.

Weight of objects Weight (objects) is computed as:

$$W(objects) = 2 \tag{3.1.3}$$

The weights of objects are assigned as 2, because it is similar as to how an object constructor is automatically called while creating it and it is a coupling. In other words, calling a function or creating an object represents the same complexity. The coupling can also occur due to method calls, which are already considered while computing the weight of structure in MCM.

Weight of cohesion is defined as:

$$W(cohesion) = MA / AM \tag{3.1.4}$$

where MA represents the Number of methods where attributes are used, and

AM represents the Number of attributes used inside methods.

While counting the number of attributes, there is no any importance of AND or MND.

CIclass can be defined as:

There are two cases for calculating the complexity of the Inheritance classes depending on the architecture:

- If the classes are in the same level then their weights are added.
- If they are children of a class, then their weights are multiplied due to inheritance property.

If there are *m* levels of depth in the object-oriented code and level j has n classes, then the Cognitive Code Complexity (CCC) [23] of the system is given as

$$CIclass = \prod_{j=1}^{m}\left[\sum_{k=1}^{n} CC_{jk}\right] \tag{3.2}$$

CDclass can be defined as:

$$CDclass = Cclass(x) + Cclass(y) + \ldots \tag{3.3}$$

Note: All classes that are neither inherited nor derived from another are parts of Cdclass even if they have caused coupling together with other classes.

Cprocedural can be defined as:

$$Cprocedural = W(variables) + W(structures) + W(objects) - W(cohesion) \tag{3.4}$$

Weight of variable *W*(variable) is defined as:

$$W(varialbes) = 4 * AND + MND \tag{3.4.1}$$

The variables are defined globally.

Weight of structure *W*(structures) is defined as the weights of all the:

$$W(structures) = W(BCS) + object.method \tag{3.4.2}$$

where BCS are basic control structures, and those structures are used globally. 'object.method' is a reachable method of a class using an object. 'object.method' is counted as 2, because it is calling a function written by the programmer. If the program consists of only procedural code, then the weight of the 'object.method' will be 0.

Weight of objects W(objects) is defined as:

$$W(objects) = 2 \tag{3.4.2}$$

Creating an object is counted as 2, as is described above (3.1.3). Here it refers to the objects created globally or inside any function which is not a part of any class. If the program consists of only procedural code, then the weight of the 'objects' will be 0.

$$W(cohesion) = NF / NV \tag{3.4.3}$$

where NF is the number of functions and NV means number of variables. Coupling is added inside *W* (structures) as mentioned in the beginning of the description of the metric.

3 Theoretical Validation

For the theoretical validation of the proposed metric, we follow the properties proposed by Briand et al. [31]. Briand et al. proposed five properties for evaluating a complexity metric. These properties provide a useful guideline in the construction and validation of complexity measures and have been used by several researchers [22], [32], [33]. In the following sections, we provide all these properties and evaluate our metric against these metrics. We want to clarify that Code quality is dependent on two different complexity metrics: Function Point and multi-paradigm complexity measurement. In our formulation, Function Point calculation is estimated at the whole project level and MCM is computed at class level. The properties proposed by Briand et al. [31] evaluate complexity measures which are applied on programs/classes/modules. From this point of view, we evaluate MCM against these properties, instead of code quality of multi paradigm programs.

Property 1: *Non-Negativity*

The complexity value given by MCM for a class can never be negative. In our formulation there is only one possibility for MCM values to be negative, when the weight of cohesion will be higher than that the sum of weights all other factors of that class. This is not possible because the weight of cohesion is computed as NF/NV, and this number cannot be greater than the sum of number of methods (if we assume the weight of each method in class is one), number of attributes and other parameters like variables, etc. Hence, MCM satisfies Property 1.

Property 2: *Null Value*

It is possible that the elements of our metric will be absent from the class; in this case our metric gives a null value. See the following Table 1.

Table 1
Metric value of elements of a class

Class	**att**	**str**	**var**	**Obj**	**MA**	**AM**	**Cohesion**	**Comp./MCM**
XX	0	0	0	0	0	0	0	0

Since the proposed measures can get a null value in a class our measures satisfy the Property.

Property complexity 3 (Symmetry): By changing the order of statements, methods, attributes, and variables, there is no effect on our metric values. In other words, MCM will not change by changing the order of its elements.

Property complexity 4 (Module Monotonicity): If we add two classes then the MCM values of the combined classes will be equal to the sum of MCM values of the individual classes. In our formulation we have also considered the effect of interference; i.e. if the classes are in a hierarchy, then first we add the complexity of classes which are the same level and then multiply with its parent's class. In this case, also Module monotonicity is preserved. We can take an example of classes in our case study. We consider three classes: Figure2P, Rectangle, and Oval. Figure2P is a parent class of Rectangle and Oval classes. We add all these three classes; the complexity of Rectangle and Oval will be added first and then multiplied by the complexity of the Figure2P. According to the property of monotonicity:

The complexity of combined classes in hierarchy will be estimated by:

$$\text{Figure2P} * (\text{Rectangle} + \text{Oval}) = 10 * (29 + 29) = 580 \quad \text{(A)}$$

If we sum the MCM values of all in depended classes, the MCM values of combined classes are

$$=\text{Figure2P} + \text{Rectangle} + \text{Oval}) = 10 + 29 + 29 = 68 \quad \text{(B)}$$

From equation A and B it is clear that the complexity of the combined classes is always equal to or greater than the sum of the complexity of the independent classes. As these examples confirm, our metric satisfies the module monotonicity property.

Property complexity 5 (Disjoint Module Additivity): This property states that if the two classes are combined, then the combined class's complexity will equal to the sum of complexity of the independent classes. This is the property of additivity, the most important property to achieve the scale of the metric. We will take two different examples to check this property, because classes may be arranged in two ways, first in the same level and second in different levels in class hierarchy.

1. Consider the two classes at the same level. In our case study, two classes Rectangle and Oval are at the same level. Therefore, when we combine these two classes we can easily observe that the MCM values of the combined classes, i.e. 29+29 =58, will be the same as when we combine the classes independently.

2. If we combine the classes at a different level, we will also find the same result. Suppose we combine the Figure 2P- Rectangle and Figure 2P-Oval.

   The MCM values of Figure 2P-Rectangle Class= 10*29= 290

   The MCM values of Figure 2P-Ovel Class= 10*29= 290

   The sum of the these two independent classes

   = MCM values of (Figure 2P- Rectangle + Figure 2P-Ovel)

   = 290+290

   = 580 (C)

   Now we compute the complexity of combined class Figure 2P- Rectangle-Figure 2P-Ovel, which is computed as

   =Figure2P * (Rectangle + Oval) = 10 * (29 + 29)

   = 580 (D)

   From equation C and D, it is proved that MCM satisfies the additive property.

   Hence, MCM satisfies this property too.

   After satisfying all these five properties, i.e. additivity, module monotonocity, symmetry, null values and non-negative, we can conclude that our MCM is a valid and sensible measure from the theoretical point of view. Further, if a complexity metric satisfies the fifth property, then the metric is also on ratio scale. Property 5 proves that MCM satisfies the additive property and is on ratio scale.

4 Applicability of Multi-Paradigm Complexity Metric on JAVA Projects

We have selected two projects for empirical validation of our metrics. Both projects are available online. We chose online projects due to two reasons: 1. the reader may also want to analyze the projects in the same way as the author. 2. They are completed and tested projects so one can assume them without any fault. The details of both the projects are the following:

4.1 Chatting Application

This is an application developed in JAVA for chat. The program is divided into two; client-side and server-side [34]. Inside this program inheritance between classes are not used; in fact, it has a simpler structure than other compared projects though it has a higher level of functionality. Therefore, it has the highest code quality. Its number of LOC (Lines of Code) is 1208.

Firstly, we estimate the MCM and the Function points to evaluate the code quality of this project. The components of the MCM are computed and summarized in Table 2.

Table 2
Chat Application – Classes

Class	att	str	var	obj	MA	AM	cohesion	Comp.
CLIENT_INFO	2	2	0	0	1	2	0.5	3.5
MainFrame(S)	0	39	4	20	0	0	0	63
THBind	3	20	0	6	2	3	0.6	28.4
Client_P	2	102	5	14	2	2	1	122
MSG_RDR	0	8	0	6	0	0	0	14
S_Client	0	2	0	2	0	0	0	4
MainFrame(C)	3	30	4	16	1	3	0.3	52.7
Form	3	68	0	20	2	4	0.5	90.5
Sign_UP	0	18	0	16	0	0	0	34
Frame3	0	15	0	10	0	0	0	25
CHAT_WIN	2	16	0	12	2	2	1	29
MSG_READER	0	8	0	2	0	0	0	10
CMD_L	1	34	0	2	2	1	2	35

A graph of the complexity values (MCM) for all the classes are shown in Figure 1. If we analyze this project (see Figure 1), we can find out that the maximum complexity is 122 which belongs to Client_P Class. This class is the most complex because it has the highest number of strings (22) and variables (5). In other words, this class has several control structures with variables. The average complexity of

the classes of this project is 39. The least complex class is CLIENT_INFO with a complexity of 3.5. This class includes only two attributes and two strings.

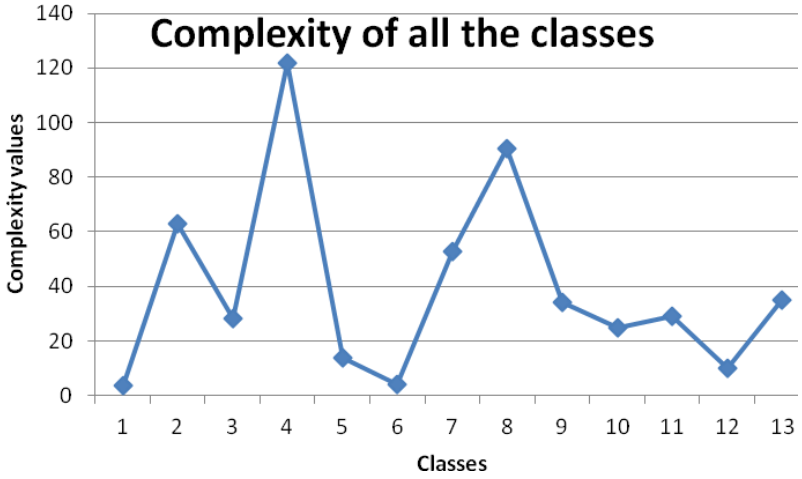

Figure 1
Complexity of Various Classes of the Chatting Application

The procedural complexity of this project is summarized in Table 3.

Table 3
Chat Application – Cprocedural

Non-Class	var+str+obj	Complexity
Cprocedural (S_CHAT)	9	9

The main program is not very complex and it consists of 9 variables, strings and the object (Table 3). Therefore, its complexity is 9 (Cprocedural).

Additionally in this project, all the classes are independent and no hierarchy amongst the classes is present. So this project

1. Does not have complexity due to inheritance. i.e CIclass = 0.
2. All the classes are treated as distinct classes so the complexity of the CDclasses will be sum of the complexity of all the classes; i.e.,

   CDclass = 511.1

Accordingly, the value of MCM is computed as:

MCM = CIclass + CDclass + Cprocedural

= 0 + 511.1 + 9

= **520.1**

The function point of this project is computed with the help of the count total computed in Table 4 and the value adjustment factors (VAF) based on the responses to the questions [30] given in Table 5.

Table 4
Chat Application – FP

Information Domain Value	**Weighting factor**				
	Count	**Simple**	**Average**	**Complex**	**Total**
EIs	17	3	**4**	6	68
EOs	78	**4**	5	7	312
EQs	1	3	4	**6**	6
ILFs	0	7	10	15	0
EIFs	17	5	**7**	10	119
Count Total					505

Table 5
Responses of questions for VAF

FP = count total* [0.65 + 0.01 x ∑ **(Fi)**]

FP Question	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**∑(Fi)**
Value Adjustment factor	0	5	5	3	1	3	3	0	3	2	3	0	0	4	**32**

= 505 x [0.65 + 0.01 x 32]

= **489.85**

Once we compute the MCM and function point (FP), we finally have to compute the code quality of the project.

The code complexity of this project is computed as,

CQ = (FP / MCM) * 10000

CQ = (489.85 / 520.1) * 10000

CQ = **9418.38108**

The code quality of this project is computed as 9418, which represents that the complexity of this project is not very high. In fact, CQ values are inversely proportional to complexity, i.e. high CQ values correlates to low complexity. This point will be clearer when we compare this project with a more complex project presented and computed in the next section.

4.2 Microprocessor Simulator

Our second project is a Microprocessor simulator. This is a simple 8085 simulator program developed in JAVA [35]. This project includes 16 classes and encompasses numerous nested loops. Its number of LOC is 2332. Due to its extremely complex structure and simpler functionalities, it has a lower CQ value.

We have to compute MCM and the function points to measure the code quality of this project. Table 6 shows the different parameters of MCM for all the classes of the project. Based on the complexity values, we have devised a graph for the complexities of all the classes in Figure 2.

Table 6

Microprocessor Simulator – Classes

class	att	str	var	obj	MA	AM	cohesion	Comp.
UserRam	1	14	0	8	2	1	2	21
RunPro	1	5	0	0	2	1	2	4
Proceed1	22	2512	15	0	7	22	0.3	2548.7
Proceed	17	5454	25	2	13	17	0.7	5497.3
SetFlag	5	44	0	0	1	5	0.2	48.8
RunErrors	0	12	0	8	0	0	0	20
MemArea	4	18	4	6	2	4	0.5	31.5
InstArea	2	18	0	24	1	1	1	43
SetC	2	15	3	0	1	2	0.5	19.5
Check	1	22	8	0	1	1	1	30
Check1	2	50	0	0	1	2	0.5	51.5
About	0	10	0	12	0	0	0	22
Check2	4	16	2	0	1	4	0.2	21.8
Check3	2	18	2	0	1	2	0.5	21.5
Check4	3	27	2	0	1	3	0.3	31.7
FlagsWindow	0	10	2	8	0	0	0	20

The graph in Figure 2 reflects the trends of the complexity of the classes of the projects. The average complexity of the classes is 527, which shows that, in general, the complexities of the classes are high. The highest complexity is 5497, which belongs to the class Proceed and is a consequence of the fact that this class contains the highest number of strings and that the complexity created by these strings is 5454. The lowest complexity belongs to the class RunPro, which is 4 due to its lowest amount of strings and attributes.

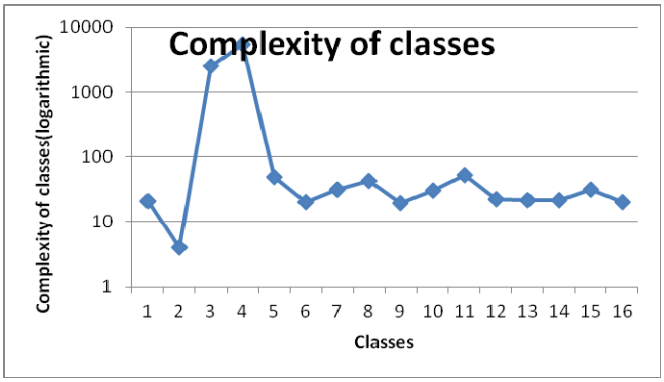

Figure 2
Complexity of Various Classes of the Microprocessor Simulator

We have to compute the complexity of the main program to calculate the procedural complexity (Cprocedural) of the project. The main program of the project has no variables, strings or objects, so complexity of this part is estimated as zero (Table 7).

Table 7
Microprocessor Simulator – Cprocedural

Non-Class	var+str+obj	Complexity
Cprocedural	0	0

Additionally, this project has several inheritance hierarchies. Figure 3 demonstrates the hierarchies among different classes. In fact, these hierarchies are the main reason of the increment of the overall complexity of the project.

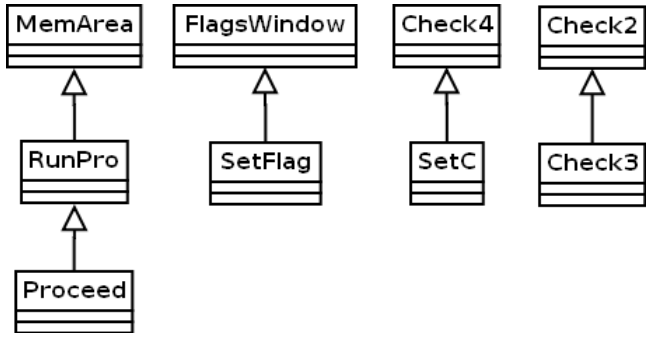

Figure 3
Microprocessor – Inheritance

Figure 3 shows that five classes are the child/subclasses in four different hierarchies. In one of the hierarchies, the depth of the inheritance tree is two.

Proceed class is at level two in the hierarchy. In the formulation of the complexity of the classes due to inheritance (CIclass), the complexities of classes are multiplied by each other. Accordingly, the total complexity of the classes caused by inheritance is computed as:

CIclass =Complexity of (MemArea*(RunPro*(Proceed))+ FlagWindows*SetFlag + Check4*SetC + Check2*Check3)

=31.5(4(5497.3)) + 20(48.8) + 31.7(19.5) + 21.8(21.5)

= 692659.8 + 976 + 618.15 + 468.7

= 694722.7

The classes which are not in the hierarchy are treated as distinct classes. The total complexities of the distinct classes are computed as:

CDclass = 2736.2

Subsequently, the complexity of overall projects is calculated as:

MCM = CIclass + CDclass + Cprocedural

= 694722.7 + 2736.2+0

= **697458.9**

The function point calculation is estimated with the information domain values for the project (to compute the total count) and value adjustment factors, as given in Tables 8 and Table 9, respectively.

Table 8

Microprocessor Simulator – FP

Information Domain Value	**Weighting factor**				
	Count	**Simple**	**Average**	**Complex**	**Total**
EIs	0	3	4	6	0
EOs	17	4	**5**	7	85
EQs	0	3	4	6	0
ILFs	0	7	10	15	0
EIFs	14	5	**7**	10	98
Count Total					183

Table 9

Responses of questions for VAF

FP = count total* [0.65 + 0.01 x ∑**(Fi)**] = 183 x [0.65 + 0.01 x 19]

FP Question	1	2	3	4	5	6	7	8	9	10	11	12	13	14	∑**(Fi)**
Value Adjustment factor	0	3	0	2	0	0	0	0	3	2	5	0	0	4	**19**

= **153.72**

The Code quality of the project is computed as:

CQ = (FP / MCM) * 10000

CQ = (153.72 / 697458.9) * 10000

CQ = **2.20400**

The code quality of this project is estimated as 2.20400.

Now we compare the above two projects. The code quality of both projects are computed as 9418 and 2.20. According to the structure of the metric, a high value of CQ represents low complexity and vice-versa. This means that the second project is comparatively more complex than the first project because its CQ value of 2.20 is much smaller than the 9418. The number of classes in the chatting application and Microprocessor simulator are 13 and 16, respectively, which are not too different (in terms of number). However, the average complexity of the Microprocessor class is 527 and the Chat application is 39, which reflects that the complexity of classes in Microprocessor simulator is much more complex in comparison to the classes of chatting applications. The complexity of the Microprocessor simulator is high because it contains a complex structure characterized by several nested loops.

The above two projects are different in nature. The MCM has well differentiated both projects in terms of their complexities. These experimentations prove the applicability of multi-paradigm metric in JAVA projects.

Conclusion and Future Work

A multi-paradigm complexity metric is evaluated through measurement theory and applied to the two JAVA projects. The evaluation of compliance with measurement theory has proved that this metric satisfies the additive property. This additive nature of the metric proves its theoretical soundness. Furthermore, the metric is applied to two real JAVA projects. The projects are different in nature (in terms of their architecture of the classes), and the MCM demonstrates a good differentiation between them in terms of their complexity, which reinforces that the MCM is useful in estimating the complexity of JAVA projects. As a future work, we aim to fix the thresholds [36] for MCM. To achieve thresholds, we will perform more experimentation on real projects in industry. We also plan to apply the MCM on projects developed in other languages.

References

[1] Francalanci, C., Merlo, F.: The Impact of Complexity on Software Design Quality and Costs: An Exploratory Empirical Analysis of Open Source Applications White paper available at: (last accessed 16.03.2010) http://is2.lse.ac.uk/asp/aspecis/20080122.pdf

[2] Banker R. D., Datar S. M., Zweig, D. (1989) Software Complexity and Maintainability, Proceedings of the tenth international conference on Information Systems, pp. 247-255, Boston, Massachusetts, United States

[3] Dawei E. (2007) The Software Complexity Model and Metrics for Object-Oriented, In Proc. of IEEE International Workshop on Anti-counterfeiting, Security, Identification, pp. 464-469

[4] Dufour B., Driesen K., Hendren L., Verbrugge C. (2003) Dynamic Metrics For JAVA, In Proceedings of the Conference On Object-Oriented Programming, Systems, Languages, and Applications, October 26-30, 2003, Anaheim, California, USA, pp. 149-168

[5] Cahoon B., McKinley K. S. (2001) Data Flow Analysis for Software Prefetching Linked Data Structures In JAVA Controller. In Proc. of The 2001 International Conference on Parallel Architectures and Compilation Techniques, pp. 280-291, September 2001, Barcelona, Spain

[6] Sundaresan V., Hendren L., Razafimahefa C., Rai R. V., Lam P., Gagnon E., Godin C. (2000) Practical Virtual Method Call Resolution for JAVA. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, And Applications, pp. 264-280, ACM Press

[7] Vijaykrishnan N., Ranganathan N. (1999) Tuning Branch Predictors to Support Virtual Method Invocation In JAVA. In Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, May 1999, pp. 16-16

[8] Qian F., Hendren L., Verbrugge C. (2002) A Comprehensive Approach To Array Bounds Check Elimination For JAVA. In Proc. of the International Conference on Compiler Construction, Lecturer Notes in Computer Science, 2304, pp. 325-341

[9] Ruf E. (2000) Effective Synchronization Removal for JAVA. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 208-218

[10] Shuf Y., Serrano M. J., Gupta M., Singh J. P. (2001) Characterizing the Memory Behavior of JAVA Workloads: A Structured View and Opportunities for Optimizations. In Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems, pp. 194-205

[11] Mäkelä S. Leppänen V. (2009) Client-based Cohesion Metrics for JAVA Programs, Science of Computer Programming, 74(5/6), pp. 355-378

[12] Nasseri E. Counsell S., Shepperd M. (2010) Class Movement and Re-Location: An Empirical Study of JAVA Inheritance Evolution, Journal of Systems and Software, 83(2) pp. 303-315

[13] Kaczmarek J. Kucharski M. (2004) Size and Effort Estimation for Applications Written in JAVA, Information and Software Technology, 46(9) pp. 589-601

[14] Pirró G. (2009) A Semantic Similarity Metric Combining Features and Intrinsic Information Content, Data & Knowledge Engineering, 68(11) pp. 1289-1308

[15] Romano D. (2011) Using Source Code Metrics to Predict Change-Prone JAVA Interfaces, In Proc of 27th IEEE International Conference on Software maintenance, pp. 303-312

[16] Sommerville, I. (2004) Software Engineering, 7th Edition, Addison Wesley, 2004

[17] Misra S., Akman I., Cafer F. (2011) A Multi Paradigm Complexity Metric, Lecture Notes in Computer Science, 6786/2011, pp. 342-354

[18] Albrecht A. J. (1979) Measuring Application Development Productivity, Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14-17, IBM Corporation (1979) pp. 83-92

[19] Misra S, Cafer F. (2011) Estimating Complexity of Programs in Python Language Technical Gazette, 18 (1) pp. 1-10

[20] Misra S. (2011) An Approach for Empirical Validation Process for Software Complexity Measures, Acta Polytechnica Hungarica, 8(2), pp. 141-160

[21] Misra S., Akman I. (2010) Unified Complexity Metric: A Measure of Complexity, Proc. of National Academy of Sciences Section A. 80(2) pp. 167-176

[22] Misra S., and Akman I. (2008) Weighted Class Complexity: A Measure of Complexity for Object Oriented Systems, Journal of Information Science and Engineering, 24, pp. 1689-1708

[23] Misra S., Akman I., Koyuncu M. (2011) An Inheritance Complexity Metric for Object Oriented Code: A Cognitive Approach, SADHANA (Springer), 36(3) pp. 317-338

[24] Basci D., Misra S. (2011) Metrics Suite for Maintainability of XML Web-Services' IET Software 5(3), pp. 320-341

[25] Basci D., Misra S. (2009) Data Complexity Metrics for Web-Services, Advances in Electrical and Computer Engineering, 9(2), pp. 9-15

[26] Basci D., Misra S. (2011) Entropy as a Measure of Complexity of XML Schema Documents' Int. A. journal of Information Technology, 8(1) pp. 16-25

[27] Basci D., Misra S. (2009) Measuring and Evaluating a Design Complexity Metric for XML Schema Documents, Journal of Information Science and Engineering, 25(5) pp. 1405-1425

[28] Tonbul G. and Misra S. (2009) Error Density Metrics for Business Process Modeling, In Proc. of the 24th International Symposium on Computer and Information Sciences, pp. 542-546

[29] Lara, P., Fernandez, l. (2008) Test Case Generation, UML and Eclipse, Dr.Dobbs Journal, 22 (11) pp. 49-52

[30] Roger S. P. (2005) Software Engineering – A practitioner's approach, 6th Edition. McGraw-Hill

[31] Briand L. C., Morasca S., Basily V. R.(1996) Property-based Software Engineering Measurement, IEEE Transactions on Software Engineering, 22 (1), pp. 68-86

[32] Gupta V, Chhabra J. K. (2009) Package Coupling Measurement in Object-oriented Software. Journal of Computer Science and Technology 24(2), pp. 273-283

[33] Costagliola G., Ferrucci F., Tortora G., Vitiello G. (2005) Class Points: An Approach for the Size Estimation of Object-Oriented Systems, IEEE Transactions on Software Engineering, 31(1) pp. 52-74

[34] Source Codes World – Chatting Application (last accessed 21.02.2010) Available at: http://www.sourcecodesworld.com/source/show.asp?ScriptID=524

[35] Source Codes World – Microprocessor Simulator (last accessed 21.02.2010) Available at: http://www.sourcecodesworld.com/source/show.asp?ScriptID=849

[36] Misra S. (2011) Evaluation Criteria for Object-oriented Metrics, Acta Polytechnica Hungarica, 8(5), pp. 109-136