DEVELOPING A COMPLEXITY METRIC FOR INNER CLASSES

[1]SIM HUI TEE, [2]RODZIAH ATAN, [3]ABDUL AZIM ABD GHANI
[1]Faculty of Creative Multimedia, Multimedia University, Cyberjaya, Malaysia
[2,3]Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Malaysia
E-mail: shtee@mmu.edu.my

ABSTRACT

In some of the object-oriented programming languages such as Java, a regular class could contain one or more nested classes. These nested classes are called inner classes which are supposed to carry out more specific tasks for their outer class. However, extensive use of inner classes would result in increasing program complexity and costly maintenance. We propose a complexity metric to measure the complexity extent of inner classes. Our metric is able to measure the complexity of inner classes in term of breadth and depth.

Keywords: *Inner class, outer class, object-oriented programming, program complexity*

1. INTRODUCTION

Inner classes are special type of classes which are defined within the body of a regular class [1]. They are used as assistant classes in certain programming languages such as Java. The role of inner classes is to assist the containing outer class in performing a specific task. An inner class is an element of its outer class. Defining inner classes in an outer class may reduce the total number of outer classes in a software application. Large amount of predefined inner classes in Java Swing [2] is an indication of its significance in modern programming.

An inner class shares all the features of a regular class. It can contain attributes, methods, constructors, and data type. In addition, an inner class is able to inherit or to be inherited. Interface implementation is also allowed at the level of inner classes. Furthermore, an inner class can be defined as abstract or final, as a regular class does.

However, extensive use of inner classes in an outer class may lead to program complexity. It increases the difficulty in program comprehension and maintenance. Furthermore, extensive use of inner classes tends to result in low class cohesion.

To date, the significance of the complexity of inner classes has not been recognized by software practitioners. In this research, we investigate the complexity of inner classes from the perspective of breadth and depth. We propose a new complexity metric to evaluate the extent of complexity of inner classes.

2. COMPLEXITY OF REGULAR CLASSES

A regular class consists of attributes, methods and constructors. The complexity of a regular class tends to increase when its elements increase. One of the common ways in evaluating the class complexity is by probing into the size of class. Line of code (LOC) is a common measure for the class size that is used in software estimation and maintenance. Code complexity correlates with program size measured by lines of code [3][4][12]. The higher level of complexity requires more efforts in maintaining the software [5][8][9][10].

There is no consensus on the idea that class size is necessarily resulting in class complexity. Grimstad and Jørgensen hold that size and complexity are two distinct parameters of a program element [6]. Furthermore, size-based models (especially LOC-based) has received strong criticism on their requirement of the class size estimation for future system. The critics hold that the lines of code of the future system are unpredictable [7]. They urge software practitioners to estimate effort than size [7].

The efforts in maintaining a software application can be reflected in the degree of complexity of classes [11]. The complexity of classes is not wholly relying on the class size, but also determined by the structural and functional relationship among class elements.

Figure 1 provides a complexity comparison between three classes in term of internal class structure.

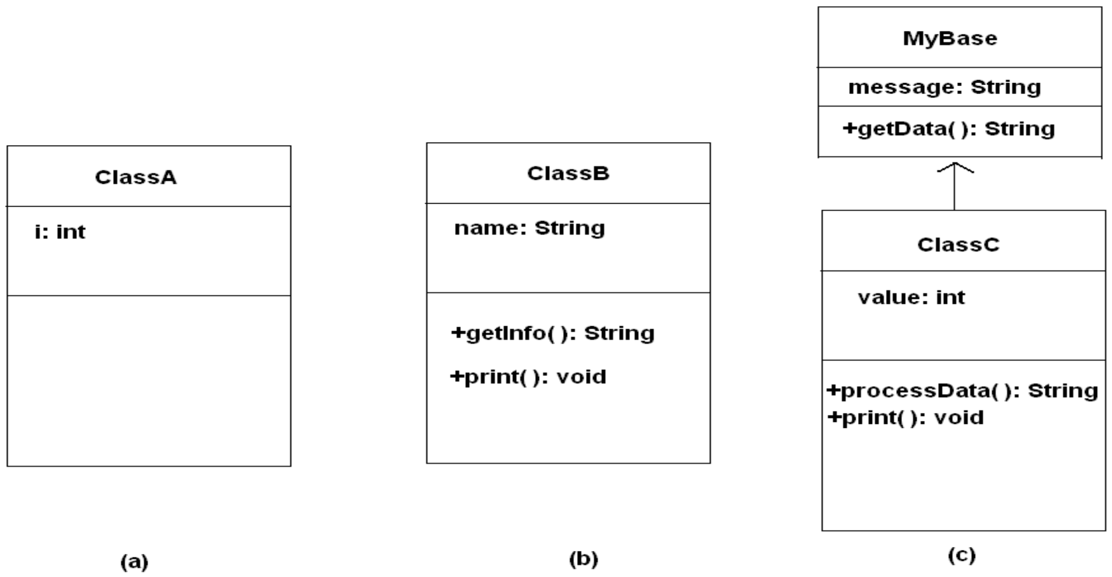


Figure 1: Structural complexity of three classes in UML

Figure 1 (a) is a regular class named *ClassA* which consists of an attribute. It is less complex as compared to *ClassB* in Figure 1 (b), as the latter class has two additional methods which are not possessed by *ClassA*. However, *ClassC* in Figure 1 (c) is more complex than Figure 1 (a) and 1 (b) because it inherits from a superclass named *MyBase* and contains three elements. The UML representation of three classes in Figure 1 depicts the class complexity from the perspective of class structure, which does not take LOC into consideration.

Apart from the structural complexity of classes, the complexity of a class may be evaluated from the perspective of internal function. Functional complexity of a class is evaluated based on the functional relatedness among class elements. It exhibits the ways of functional relatedness among attributes, methods, and constructors.

In Figure 2, two distinct classes are given for the analysis of their functional complexity.

```
class Zoo{

  private int animal_no;

  Zoo(int animal_no){

    this.animal_no=animal_no;
  }

  public int getAnimalNumber(){

    return animal_no;
  }

}
```

(a)

```
class Scores{

  int low,medium,high;

  Scores(int low,int medium,int high){

    this.low=low;
    this.medium=medium;
    this.high=high;
  }

  public int getLow(){
    return low;
  }

  public int getMedium(){
    return medium;
  }

  public int getHigh(){
    return high;
  }
}
```

(b)

Figure 2: Functional complexity of two classes in Java

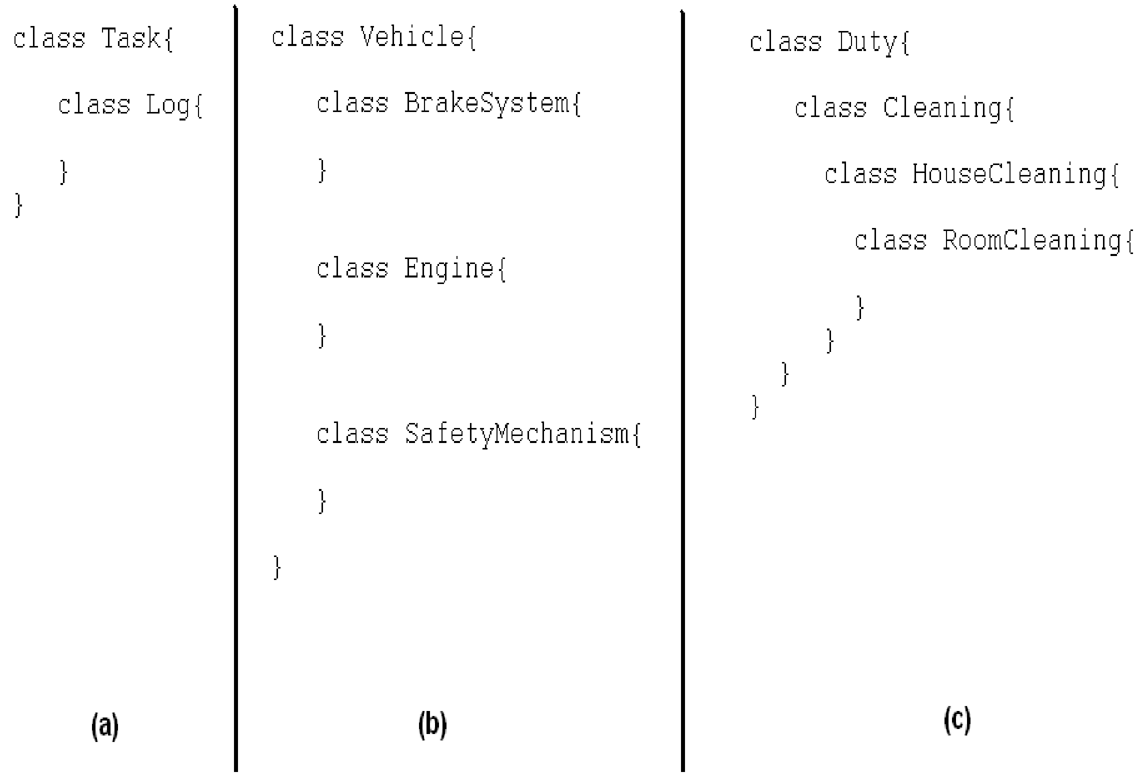
Figure 2(a) is a class named *Zoo* which consists of one attribute, constructor and method, respectively. Class elements are functionally related in the sense that attribute *animal_no* is initialized in constructor and returned in method *getAnimalNumber( ).* Hence, the functional complexity in class *Zoo* involves only the two interactions, which are attribute-constructor and attribute-method. Figure 2(b) depicts a class named *Scores* which consists of three attributes, one constructor and three methods. Each attribute is referenced twice, by constructor and one of the methods. The functional complexity in class *Scores* involves six interactions, which are three attribute-constructor interactions and another three attribute-method interactions. Comparing the functional complexity between class *Zoo* and class *Scores*, it is apparent that the latter class is more complex because it has more functional interactions. It should be noted that the degree of functional complexity is higher in the event that more class element interaction is found in a class.

3. COMPLEXITY OF INNER CLASSES

An inner class is defined in an outer class, as shown in Figure 3.

Figure 3(a) depicts a simplest form of inner class named *Log* which is defined within its outer class *Task*. Inner class *Log* is an element of its outer class. In Figure 3(b), class *Vehicle* contains three inner classes, which are *BrakeSystem*, *Engine*, and *SafetyMechanism* at the same breadth. These inner classes are defined at the same level of depth, which can be represented by Figure 4 (a). Figure

3(c) depicts an outer class *Duty* which contains an immediate inner class named *Cleaning*. *Cleaning* contains an immediate inner class named *HouseCleaning*. Lastly, *HouseCleaning* contains an immediate inner class named *RoomCleaning*. It is to be noted that the inner classes in Figure 3(c) are defined at different level of depth, as represented in Figure 4(b).



```
class Task{

   class Log{

   }
}
```

(a)

```
class Vehicle{

   class BrakeSystem{

   }


   class Engine{

   }


   class SafetyMechanism{

   }

}
```

(b)

```
class Duty{

   class Cleaning{

     class HouseCleaning{

       class RoomCleaning{

       }
     }
   }
}
```

(c)

Figure 3: Examples of inner class

.

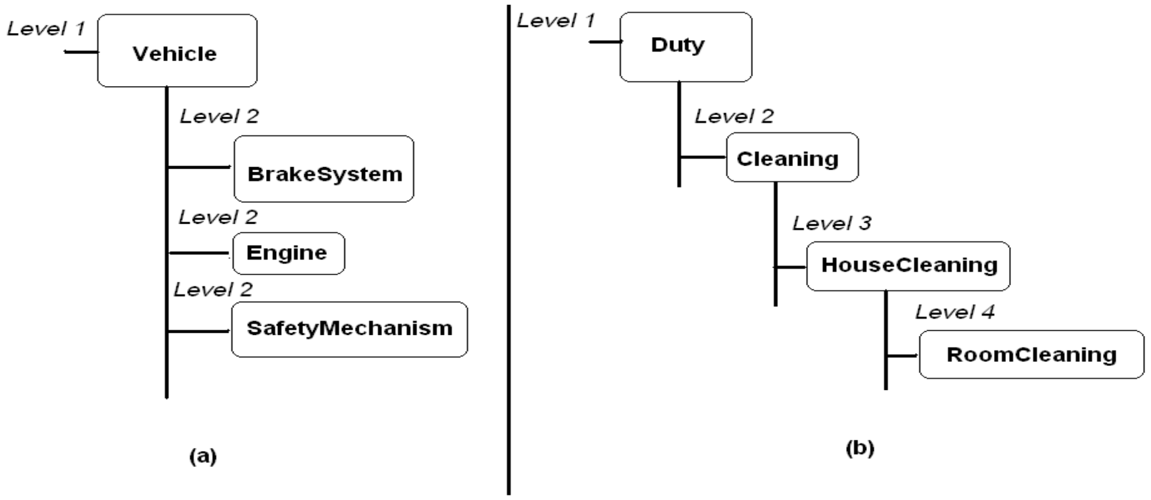


Figure 4: Depth level of inner classes.

Figure 4 is a diagram of depth level of the inner classes. Figure 4(a) illustrates two levels of class depth for the program in Figure 3(b). The outer class *Vehicle* is defined at the first level whereas inner classes *BrakeSystem*, *Engine* and *SafetyMechanism* are defined at level 2. It can also be said that all of the inner classes are defined at the same breadth under class *Vehicle*.

Figure 4(b) illustrates four levels of class depth for the program in Figure 3(c). All inner classes are defined at different breadth and depth. The inner-most level of inner class in Figure 4(b) is *RoomCleaning*, which is immediately defined within *HouseCleaning*. The multi-depth structure of inner classes in Figure 4(b) implies that the farthest level (level 4) is distantly related to the root level (level 1, which is the outer class).

It is a normal practice to define inner classes at different breadth and depth, which is the hybrid of Figure 4(a) and 4(b). Our research considers the complexity of inner classes in term of breadth and depth. Figure 5 illustrates an example of inner classes which are defined at different breadth and depth.

In Figure 5, *Expenditure* is an outer class which contains three immediate inner classes at level 2, which include *LoanPayment*, *LuxuryExpenses*, and *LivingExpenses*. These three inner classes are defined at the same breadth and do not overlap. Inner class *LoanPayment* does not contain further inner class. Inner class *LuxuryExpenses* contains an immediate inner class named *TourExpenses* at depth level 3.

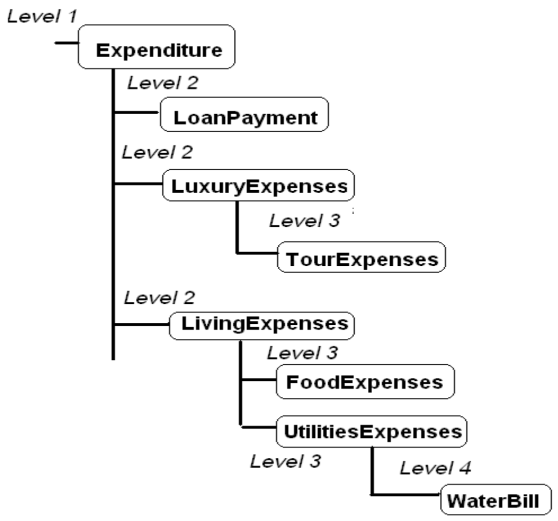


Figure 5: Inner classes defined at multiple breadth and depth

It should be noted that *TourExpenses* is indirectly contains within the outer class *Expenditure* (level 1), but it does not contain within *LoanPayment* and *LivingExpenses* (level 2). Inner class *LivingExpenses* contains two immediate inner classes at level 3, which are *FoodExpenses* and *UtilitiesExpenses*. *FoodExpenses* has no further inner class but *UtilitiesExpenses* contains one inner class, namely *WaterBill*, at level 4.

In our research, we propose a complexity metric by taking the breadth and depth of inner classes into consideration. We define our complexity metric (C) for inner classes as $\sum \frac{b}{d}$

Where b denotes the breadth of a particular depth level, d denotes the depth level. The complexity value for inner classes is derived from the sum of breadth to depth ratio of the classes. In the event that there is no inner class defined in a regular class, the complexity metric(C) for that class is minimal, which is 1, as illustrated in Figure 6.

```
class X{

}
```

Breadth (b) at level 1 : 1

Depth (d) at level 1 : 1

Complexity for class X (C)= $\sum \frac{b}{d}$

$= 1$

Figure 6: Minimal inner class complexity value.

Class *X* in Figure 6 has no inner class. Thus, its complexity value (for inner class) is the lowest, which is 1. Class X is defined at depth level 1, where there is only one class at this depth. Our complexity metric only measures the complexity caused by inner classes. We do not consider class complexity in term of LOC, method-attribute interaction or other factors.

Figure 7 to 12 provide examples of inner classes which are different in term of breath and depth. Their complexity value is provided based on our proposed metric. As the complexity value increases, the program becomes more complex in term of the inner class.

```
class A{
  class B{
   }
}
```

Breath(b1) at level 1: 1
Depth (d1) at level 1: 1
Breadth(b2) at level 2: 1
Depth (d2) at level 2: 2

Complexity (C)= $\frac{b1}{d1} + \frac{b2}{d2}$

$= 1.5$

Figure 7: Inner class complexity value =1.5

Figure 7 demonstrates an outer class *A* which contains an inner class *B*. There is only one class (b1) at the depth level 1 (d1). The inner class *B* is defined at depth level 2 (d2), where the breadth (b2) is 1. The obtained inner class complexity value, 1.5, is derived from the sum of breadth to depth ratio of the classes. Program in Figure 7 has higher complexity value as compared to program in Figure 6, implying that the former has more complex inner class.

```
class Foo{
  class Goo{ }
  class Hoo{ }
}
```

Breath(b1) at level 1: 1
Depth (d1) at level 1: 1
Breadth(b2) at level 2: 2
Depth (d2) at level 2: 2

Complexity (C)= $\frac{b1}{d1} + \frac{b2}{d2}$

$= 2$

Figure 8: Inner class complexity value=2

Program in Figure 8 consists of two levels of depth. Inner class *Goo* and *Hoo* are defined at the same breadth at d2. The complexity value for inner classes in Figure 8 is higher than the program in Figure 6 and 7. It is because class *Foo* has more inner classes than class *X* and *A*.

```
class CW{
  class CX{ }
  class CY{ }
  class CZ{ }
}
```

Breath(b1) at level 1: 1
Depth (d1) at level 1: 1
Breadth(b2) at level 2: 3
Depth (d2) at level 2: 2

Complexity (C)= $\frac{b1}{d1} + \frac{b2}{d2}$

$= 2.5$

Figure 9: Inner class complexity value=2.5

In Figure 9, class *CW* consists of three inner classes at the same breadth at d2. Its complexity value is higher than that of class *Foo* (in Figure 8) because it has more inner classes.

```
class P{
  class Q{ }
  class R{
     class R1{ }
  }
}
```

Breath(b1) at level 1: 1
Depth (d1) at level 1: 1
Breadth(b2) at level 2: 2
Depth (d2) at level 2: 2
Breadth(b3) at level 3: 1
Depth(d3) at level 3: 3

$$\text{Complexity (C)} = \frac{b1}{d1} + \frac{b2}{d2} + \frac{b3}{d3}$$
$$= 2.33$$

Figure 10: Inner class complexity value=2.33

Class *P* in Figure 10 has equal number of inner classes as class *CW* (Figure 9) does. However, class *P* has three level of depth, as class *CW* has two. The complexity value for class *P* is lower than class *CW* because wider breadth (b2) of class *CW* at d2 implies higher degree of functional irrelevance among inner classes *CX*, *CY*, *CZ.*

It is interesting to compare class *J* in Figure 11 with class *P* in Figure 10. Both outer classes have similar structural arrangement of inner classes. There is one inner class at d2 contains an inner class at d3, in Figure 10 and 11. Structural similarity between outer class *P* and *J* implies the same complexity value, which is 2.33.

```
class J{
  class K{
     class K1{ }
  }

  class L{ }
}
```

Breath(b1) at level 1: 1
Depth (d1) at level 1: 1
Breadth(b2) at level 2: 2
Depth (d2) at level 2: 2
Breadth(b3) at level 3: 1
Depth(d3) at level 3: 3

$$\text{Complexity (C)} = \frac{b1}{d1} + \frac{b2}{d2} + \frac{b3}{d3}$$
$$= 2.33$$

Figure 11: Inner class complexity value=2.33

Lastly, class *CA* in Figure 12 has two inner classes (*CB* and *CC*) at d2. Each of them contains one inner class at d3. It is the most complex program as highest complexity value (2.67) is yielded.

```
class CA{
  class CB{
     class CB1{ }
  }

  class CC{
     class CC1{ }
  }
}
```

Breath(b1) at level 1: 1
Depth (d1) at level 1: 1
Breadth(b2) at level 2: 2
Depth (d2) at level 2: 2
Breadth(b3) at level 3: 2
Depth(d3) at level 3: 3

$$\text{Complexity (C)} = \frac{b1}{d1} + \frac{b2}{d2} + \frac{b3}{d3}$$
$$= 2.67$$

Figure 12: Inner class complexity value=2.67

Based on the programs given from Figure 6 to 12, our metric suggests that the lowest inner class complexity value is 1 when there is no inner class defined within an outer class (Figure 6). Our metric also suggests that the breadth of inner class has more impact on greater complexity value than the depth does (Figure 9 and 10). It is because more inner classes defined at the same breadth would increase the number of functionally unrelated inner classes. Lastly, our metric does not assume a ceiling complexity value for inner classes. However, software developers are advised to keep the complexity value for their inner classes as low as possible.

4. CONCLUSION

Increasing complexity and maintenance effort are the inevitable consequences incurred by the extensive use of inner classes in a software application. We have developed a complexity metric for the inner classes. Our metric measures the complexity from the perspective of breadth and depth of inner classes. Software developers may use our complexity metric as a guideline to reduce inner class complexity in their daily job.

REFERENCES

[1] C. Horstmann and G. Cornell. *Core Java 2*. USA: Prentice Hall. 2004.

[2] M. Robinson and P. Vorobiev. *Swing*. USA: Manning Publications Co. 2003.

[3] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, " Early Estimation of Software Quality Using In-Process Testing Metrics: A Controlled Case Study", *WoSQ'05*, May 17 2005, pp. 1-7.

[4] J. Asundi, "The Need for Effort Estimation Models for Open Source Software Projects", *Firth Workshop on Open Source Software Engineering (5-WOSSE)*, 2005, pp 1- 3.

[5] M. Korte and D. Port, "Confidence in Software Cost Estimation Results Based on MMRE and PRED", *PROMISE'08*, May 12-13, 2008, pp 63-70.

[6] S. Grimstad and M. Jørgensen, "A Framework for the Analysis of Software Cost Estimation Accuracy", ISESE'06, September 21-22, 2006, pp 58-65.

[7] J. Aranda and S. Easterbrook, "Anchoring and Adjustment in Software Estimation", ESEC-FSE'05, September 5-9, 2005, pp 346-355.

[8] R.D. Banker and S.A. Slaughter, "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement", *Information Systems Research*, Vol 11, 2000, pp 219-240.

[9] J. Rech and W. Schäfer, "Visual Support of Software Engineers during Development and Maintenance", *ACM SIGSOFT Software Engineering Notes*, Vol 32 No. 2, March 2007, pp1-3.

[10] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring", *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002, pp1-10.

[11] K.K. Aggarwal, Y. Singh, and J.K. Chhabra, "An Integrated Measure of Software Maintainability", *Annual Proceedings of Reliability and Maintainability Symposium*, 2002, pp235-241.

[12] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability", *Sixth International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2007, pp30-39.