Visual Indicator and Software Metric Formula to Determine Class Cohesion Problem and Software Design Quality

Djon Irwanto
Department of Computer Science
 BINUS University
 Republic of Indonesia
djonirwanto@gmail.com http://www.djonirwanto.com

Abstract: - Classes is the basic unit in object oriented software, therefore their quality has significant impact on the overall quality of the software. Class cohesion is one of the important factors that determine whether the design quality of a software is good or bad. In software development it is essential for software developers to gauge the degree of cohesion; yet in software development practice, it is rarely found software developers who are concerned with class cohesion and design quality. The lack of concerns resulted in their assembled software being difficult to maintain, understood and reused. Although these days there are many literatures discussing software metric as a tool to calculate the degree of cohesion, in reality the tool is rarely used. To seek an answer why these developers are reluctant to employ these metrics, a survey to several software developers was conducted. The result of the survey reveals that the developers prefer for an easier and more practical measurement. The use of a complicated and less practical measuring method makes the developers unwilling to gauge their design. Through this paper, the researcher is trying to formulate a more practical software metric and visual indicator that are easier to use. By applying visual indicator, the classes that have bad cohesion are able to be detected visually as well as able to map the risk of design outcome from metric gauging. It is hoped that visual indicator and software metric are able to be complementary tools for design software and refactoring; able to detect class cohesion problems fast and pinpoint area refactoring location.

Key-Words: - Class Cohesion, Object-Oriented Software Quality, Visual Indicator, Software Metric, Refactoring

1 Introduction

In object oriented software, classes have an important role in software construction. It can be deduced that the classes are the basic unit in software building. Thus, the quality of any software is determined by those classes.

In Software Engineering, cohesion is one deciding factor that verifies whether the design quality of a software is good or bad. Research in Object Oriented Software Engineering states that if a software design has a low degree of cohesion, then the software will be difficult to reuse, maintain and understood [13]. The effects from these three points are abundant in developing software – among others, they make the process of future software development more difficult, increasing testing effort, making maintenance effort higher, and finally potentially upscaling the software development cost. It is then deemed necessary that in the phase of software development, the software designers measure the cohesion degree - the high and low- on the design software they made.

To measure the cohesion degree of any software, we have to calculate using metric formula. In software engineering we have known many software metrics to calculate the cohesion degree, such as Chidamber and Kemerer with their metric Lack of Cohesion Methods (LCOM) [3], Li and Henry suggesting LCOM using undirected graph that represents every method as node and minimal sharing instance variable as edge [4]. Hitz and Montazeri then redefined Li and Henry’s LCOM by using graph theoretic [5]. Beimen and Kang put forward two class cohesion metrics: Tight Class Cohesion and Loose Class Cohesion [6]. Ertemel et al advise COMIAS [7]. Based on the survey conducted in this research, in the process of software development, there are not many developers or students that apply the previously mentioned models to calculate their designs – especially in Indonesia.

Why are they so reluctant or have an aversion in using software metric to measure their software designs? In this study, a survey was carried on to find out the answer. The result is as follows. First,
their awareness of software quality is very low; they consider software metric as something unimportant. Second, they do not have enough knowledge to understand and interpret the result of metric measurement. Third, they are too indolent to gauge their work. There are many factors that caused this lethargy; one of them is that they have no adequate understanding to comprehend and decode the outcome of the metric measurement.

From further survey (a continuation of my last research), I got some inputs from several software developers that they would like calculating or measuring methods which are easy and practical. For example, to calculate functional cohesion [2] that involves detailed analyzing code program is viewed by the majority developers as a complicated and less practical method of measuring cohesion. Applying complicated and less practical measuring method will no doubt add the developers’ reluctance in gauging their design.

This situation encourages me to make metric formulas to detect cohesion problem whose result can be visually represented via visual indication. This visual indicator will then be able to demonstrate characteristics of the classes so that classes with low degree of cohesion and problematic can be seen clearly with all their risk variants. In turns, it is hoped that visual indicator and this metric formulas can persuade the software developers (or students) to be willing to check their software design.

This paper is divided into several sections. Section 1 explains the introduction and section 2 will deal with visual indicator models and software metric formulas suggested. Section 3 will explore further on the visual indicator and refactoring. In section 4 experimental results will be discussed and finally conclusion will be dealt with in section 5.

2 Model of Visual Indicator to Detect Class Cohesion Problem & Software Metric

To make detecting process and measuring cohesion be more practical and easy, in this study a visual indicator is made. This visual indicator must be able to interpret the meaning of a metric formula calculation to detect how good or bad the degree of cohesion of a design accurately and fast. The following is a model from visual indicator to detect bad class cohesion.

Focus on figure 1; diagram Cartesian has 2 axes. Axis X explains the degree of delegation of each available class in the design of any software. Axis Y explains the degree of complexity of classes in any software designs.

The element of a class is attribute and method. There are 2 types of method categorized in this paper; they are primary method and secondary method. Primary method is a method that realizes behavior of a class which is closely related to the main responsibility that class carries. Secondary method is a method that realizes behavior of a class whose main responsibility is not carried by that class.

Perhaps the question is why method in class must be classified into primary and secondary? The reason I am inspired to do this was from Dallal’s research which excluded special method [1]. Dallal in his paper claims that several class cohesion metrics are proposed in the literature and the impact of considering the special method (constructor, access method) in cohesion calculation is not empirically studied for most of them [1]. Yet, studying further, it is found that on certain condition, excluding constructor and access method such as setter and getter can end up in a mistake/error. Let me take a simple example. For instance, there is a class Users in software for authentication process. Class Users only provides constructor, setter and getter attribute username and password for login purposes on authentication process and password change. When we state that setter method is a special method that must be excluded then in this case a wrong decision has been made. Why? Because method setUsername() and setPassword() on class Users realizes behavior that is closely related with the main responsibility that is taken by class Users to change username, password for security purposes. Thus, in my opinion, the decision to categorize methods into primary and secondary is far wiser than rigidly deciding constructor, setter and getter as a special method that must be excluded.
The degree of complexity of classes that exists in design software is a unique dimension to be discussed in this paper, because this dimension characterizes a class; whether or not it has a characteristic as an entity class or not. If a class is categorized as an entity then the class has a low degree of complexity. The reasoning for this is because the class has many attributes but only has few (or even none in extreme condition) primary method. Entity class has measuring scale from 0 to 0.5 in this dimension.

If the reverse happens, then the class has a high degree of complexity. Why? Because that class does many control functions, i.e.: functions that control other classes or do a series of algorithm for certain purposes. Class like this is called function class and has a measuring scale bigger than 0.5 to 1 in this dimension.

In general, in architecture component design, entity classes are within component Problem Domain Model (PDM). Instance from entity classes in component PDM will persistence in database (or object database) [16], whereas function classes is in function component. In this study, I propose using the following metric formula to count degree of complexity of a class:

\[ D_C = \frac{T_{PM}}{T_{PM} + T_{ATR}} \]

\( D_C \) = Degree of Complexity
\( T_{PM} \) = Total Primary Method
\( T_{ATR} \) = Total Attribute

Figure 2. Architecture Component Design [15]

Based on the research process done all this time metric formulas degree of complexity has a specialty; that is able to detect the presence of bloated class thru the variable \( T_{PM} \). The bigger the value of this variable, the more careful we need be because we can be certain that inside our design there is bloated class. As an example, the result of \( D_C \) class \( Z=1 \) is acquired from \( 200/(200+0) \), then we can ascertain class \( Z \) as bloated function class which has done too many function control. Class like this will definitely have low cohesion.

The high and low degree of delegation also plays a role in detecting the high and low degree of cohesion in the software design. Why? First, cohesion is stated as the ratio of consistency of the responsibility of a module with each other [7]. According to Larman, class with low cohesion does many unrelated things, or does too much work that should have been delegated to other object [13]. Therefore, in detecting degree of cohesion, it is imperative that we see how focus a class is towards its responsibility and how that responsibility is delegated to each collaborated classes. It can then be deduces that degree of delegation plays a role in deciding whether class cohesion is high or low.

Classes with low cohesion in general have bad delegation. These classes bear excess responsibility than what should have been delegated to other class and vice versa. A typical trait of classes that have excess responsibility is that usually these classes have many primary methods. Most of these methods were supposedly be delegated to other classes. In the meantime, other classes that interact with it (they) have fewer primary method or sometimes none at all. Larman also warned that classes with higher cohesion have relatively fewer methods with highly related functionality and do not do many jobs. [13].

To calculate degree of delegation in this paper, I am proposing the following metric formula:

\[ D_D = 1 - \left( \frac{C_i}{C_c - 1} \right) \]

\( D_D \) = Degree of Delegation
\( C_i \) = Amount of Class that interacts with class X
\( C_c \) = Amount of Collaborated Class

A combination dimension \( D_D \) and \( D_C \) in visual indicator can map the risk of a design. Let us look at figure 1, where there are 4 characteristic classes. Extreme data class point has a metric calculation of (0,0). In this situation, class does not have primary method at all. If the metric calculation \( D_D \) and \( D_C \) states that a class falls into coordinate (0,0) or be near that coordinate in quadrant III, then you should re-examined your design. Why? Because in reality, extreme data class rarely exists. Even product catalog (as an entity class) still has at least one primary method, that is setUnitPrice(). If there is extreme data class in software design, we have to suspect that there might be other class around or in the vicinity that take over the responsibility of that
class. So, we can conclude that behind the existence of extreme data class, in general there are other classes with low cohesion in the vicinity. The more extreme data class you have in your design the worse your software design is. To fix it we can redesign or do refactoring.

If a metric calculation $D_D$ and $D_C$ places a class in coordinate $(1,0)$ then this is a thing of the impossible. The reason is that there is no class that does not have primary method which does not interact with other classes. If metric calculation against a class is in quadrant IV approaching point $(1,0)$ then you need to re-examine your design and ask why there are so many extreme data class collaborating there? Odd, is it not? The next question is where those primary method classes go? If this happens to your design, in my opinion, redesigning it is the only solution to fix it. Based on my experience, in this situation, refactoring against source code to improve design probably will not work – especially on big and complex problem domain.

Rare point has metric calculation coordinate of $(1,1)$. The existence of rare class seldom happens. Hence, if this class is presence in your design then you need to double check carefully and decide whether or not it is right. Based on the experience and study I did, characteristics of rare class that is acceptable in design is when the value of variable $T_PM$ is not big (or the sum does not exceed class context). Some parts of rare class that I have come across have static methods. Examples of rare class we can come across are Math class having the function of $Pow()$, $Sqrt()$, $Sin()$, $Cos()$, etc in it. If the value of its variable $T_PM$ is big then that class is claimed to be bloated class that put up too many functions; functions that are not supposed to carry so that they have low cohesion.

The uniqueness of metric calculation in quadrant II exists when the calculation approaches point $(1,1)$. When this happens, then we should be extra careful. The amount of primary methods are not the only problem indicator. We need to ask our self why there are so many collaborated classes present. There is a possibility that the design is at fault. The bottom line is when inside quadrant II and approaching point $(1,1)$ we need to exercise a lot of evaluation.

Metric calculation point $(0,1)$ can be categorised within normal boundary as long as the $T_PM$ variable is small (not big). If the $T_PM$ is small, metric calculation places a class in the coordinate $(0,1)$ or nearby as safe. However, if the $T_PM$ variable is big, then classes in coordinate $(0,1)$ or near to that coordinate are stated to be Bloated Class. That is why coordinate point $(0,1)$ is said to be conditional point. Usually, bloated class on conditional point has too many interactions with other classes and has low cohesion. To fix this mistake, you can do refactoring or redesign your software. The bigger the amount of $T_PM$ variable, the more refactoring effort is needed.

3 Visual Indicator and Refactoring

In software development, early detection software defect is crucial. Reddy et al state in their paper that to detect design defect software metric is applied [10]. I do not argue with Reddy et al’s statement, but my experience says that metrics calculation result do not tell us whether this is good or bad [8]. To understand metrics calculation, we need deeper understanding on metrics measurement [8]. My idea is to use indicator which can visually map the meaning of metrics measurement rather than only metrics calculation results.

Experience tells me that visual indicator is very useful and helpful in design process and refactoring [17]. How is this possible? Facts on past refactoring researches tell that problems faced by software engineer are discovering area where to apply refactoring [9] [11]. That is the reason why Fowler said that the detection of such problematic area inside refactoring is based on human intuition [14] [9] [11]. Fowler even mentioned that “in our experience no set of metrics rivals informed human intuition” [11]. However, the question is ‘are software engineers all over the world as the doer have the same ability? If the answer is no, then what Fowler claimed as human intuition cannot be fully relied on.

If human intuition cannot be fully relied on, then what is the solution for these problems? In this research, the writer is of the opinion that we should go back to software metrics as our standard and anchor. My point of view is in line with several other experts’ opinions which are stated in their papers [11] [10]. Yet, experience and past researches show that metrics calculations result does not state whether it is good or bad, just depicting risks that will be faced [8]. That is why to enable interpret the risks we are going to face, we need deeper understanding from the software developer. To ease interpretation process on the risks, visual indicator that maps out the software metrics calculations is made. In addition to this, similar research shows that the advantage of visualization process is that it will support the decision to apply refactoring [11]. Thus, this visual indicator would
easily pinpoint which class or location is problematic (bad cohesion) and needed to be refactoring or redesigned. Therefore, by using visual indicator in figure 1, we need not use human intuition again to detect the problems and area refactoring.

The relationship between refactoring and design process is closely related. Fowler et al stated that “Without refactoring, the design of the program will decay” [14]. Past researches also testify that design evolution is automatable with refactorings [12]. When visual indicator pinpoints problematic areas in the design that needed refactoring or redesigning, then it will directly give contribution in design improvement and program code.

In iterative and incremental software development, visual indicator can be used to gauge class cohesion from initial components that are present in the initial phase of iteration. If in the initial phase of design we have made bad design, the use of visual indicator can help us detect bad design (low cohesion) in the early development process. Hence, defect removal can be initiated from the beginning. Fowler et al’s experiences inform us that good design is essential for rapid software development. Without good design you can progress quickly for a while but soon poor design starts to slow you down [14].

Because I see there are several positive advantages from using visual indicators, in this opportunity, I would like to suggest that visual indicator in figure 1 be used as a complimentary tool to design software and refactoring.

4 Experimental Result

Bad design greatly contributes in decreasing software quality. In this section, I would like to show a way if detecting cohesion problems on bad design using visual indicator on figure 1.

Let us look at figure 3, class diagram component PDM of a software. Let us now calculate $D_C$ and $D_D$ of each class. Class customer has 5 secondary methods, they are: constructor, getId(), getName(), getAccount() and getAddress(), the rest is primary method. $D_C$ Cust = 6 / (6+4) = 0.6. Class Customer Address does not have primary method, $D_C$ Addr = 0/(0+2) = 0. Class Customer Address does not have primary method, $D_C$ Addr = 0/(0+2) = 0. Class Product has 1 primary method setInterest(), $D_C$ Prod = 1/ (1+3) = 0.25. Class Account has 3 secondary methods which are getNumber(), getOpenDate(), and getTransaction(), the rest is primary method. $D_C$ Acc = 4 / (4+4) = 0.5. Finally class transaction does not have primary method, $D_C$ Trans = 0/ (0+4) = 0.

Next, we will count the degree of complexity of each class. $D_D$ Cust = $1 – (2 / (5-1)) = 0.5$. $D_D$ Addr = $1 – (1 / (2-1)) = 0$. $D_D$ Prod = $1 – (1 / (2-1)) = 0$. $D_D$ Acc = $1 – (3 / (4-1)) = 0$. $D_D$ Trans = $1 – (1 / (2-1)) = 0$. Then we have coordinate Cust(0.5, 0.6), Addr(0,0), Prod(0, 0.25), Acc(0, 0.5) and Trans(0,0). The followings are the envisage result of visual indicator to detect bad cohesion in the design.

![Figure 3 Simple Banking System Example](image)

**Figure 3 Simple Banking System Example**

![Figure 4 Detect Problem using Visual Indicator](image)

**Figure 4 Detect Problem using Visual Indicator**

Figure 4 visualized that there are many setbacks in design in figure 3. First, there are 2 extreme data class presence. As has been previously explained, behind extreme data class, in general, hides classes having inferior cohesion. Second, Class Customer is in the function class area, yet it is supposedly to be within entity class area because object from customer will persistence into database (or object...
database). Third, even though class account is not in the function class but it and class customer are suspected as class with inferior cohesion behind the presence of two extreme data class. The only class accepted is class product.

To put design in figure 3 right, you can redesign or do refactoring and the result can be seen in figure 5. In figure 5, class Transactions is changed to abstract class. After metric \( D_D \) and \( D_C \) are counted, each class the value of: \( D_C \ cust= \frac{4}{4+4}= 0.5 \), \( D_D \ cust= \frac{1}{1+3}= 0.25 \), \( D_C \ addr= \frac{1}{1+4}= 0.2 \), \( D_D \ addr= \frac{1}{1+2}= 0.5 \), \( D_C \ prod= \frac{1}{1+3}= 0.25 \), \( D_D \ prod= \frac{1}{1+2}= 0.5 \), \( D_C \ acc= \frac{1}{1+4} = 0.25 \), \( D_D \ acc= \frac{1}{1+2} = 0.5 \), \( D_C \ deposit= \frac{1}{1+4} = 0.25 \), \( D_D \ deposit = \frac{1}{1+2} = 0.5 \). The result is Cust(0, 0.5), Addr(0, 0.2), Prod(0, 0.25), Acc(0.25, 0.5), Deposit(0, 0.2). Withdraw is equal to Deposit.

Once metric calculation is obtained, let us put it in visual indicator to see the result in figure 6. It is shown that the problem of low class cohesion is already fixed. There is no more extreme data class and classes with inferior cohesion that are hiding around extreme data have been mended.

In this case, there is an interesting issue. Constructor Deposit() and Withdraw() on class Deposit and Withdraw are primary method. This is an example that I have mentioned before; indeed, that in certain condition excluding constructor and access method such as setter and getter will end up in a mistake.

![Figure 6 Detect Problem using Visual Indicator](image)

Let us look at a different case in figure 7. In that class diagram, class Users are seen as abstract class. Let us calculate its metric.

\[
D_C \ Admin= \frac{2}{(2+3)}= 0.4 \quad D_C \ Operator= \frac{1}{(1+3)} = 0.25 \quad D_C \ Grant = \frac{1}{(1+1)} = 0.5 \quad D_C \ Security= \frac{3}{(3+0)} = 1 \quad D_D \ Admin= 1 \quad D_D \ Operator= \frac{1}{(2-1)} = 0 \quad D_D \ Grant = 1 \quad D_D \ Security= 1 
\]

The result is Admin(0, 0.4), Operator(0, 0.25), Grant(0, 0.5), Security(0.5, 1). Next, let us put in the result of this metric calculation into visual indicator as seen in figure 8.
In figure 8, it is seen that design software in figure 7 does not have any problem. In visual indicator it is noticed that the classes on that design has no problem with low degree of cohesion.

5 Conclusion
Visual indicator and metric formula that are suggested in this paper are significant tools for software designers to make a decision on design and refactoring. Combine use of metric formula and visual indicator can help us detect problems on class cohesion and software design quality easier, faster and more practical. Using visual indicator effectively helps us to isolate refactoring area and no longer need to be dependent on human intuition. Visual indicator enables us to do early or final detection of design defect. Finally, employing visual indicator and metric formula aids us increasing software maintainability because they play a part in making a design and code easier to be understood, modified and tested.

References:
[17] Irwanto Djon, Refactoring pada Object Oriented Software dan Object Database, Andi Publisher, Yogyakarta, 2010.