Revised Modularity Index to Measure Modularity of OSS Projects with Case Study of Freemind

Andi Wahju Rahardjo Emanuel
Informatics Bachelor Program
Maranatha Christian University
Bandung, Indonesia

Daniel Jahja Surjawans
Informatics Bachelor Program
Maranatha Christian University
Bandung, Indonesia

ABSTRACT
Open Source Software (OSS) Projects are gaining popularity worldwide. Studies by many researchers show that the important key success factor is modularity of the source code. This paper presents the revised Modularity Index which is a software metrics to measure the modularity level of a java-based OSS Projects. To show its effectiveness in analyzing OSS Project, the Modularity Index and its supporting software metrics are then used to analyze the evolution of Freemind mind mapping OSS Project. The analysis using Modularity Index and its supporting metrics shows the strength and weaknesses of the Freemind OSS Projects.

2. RELATED WORKS
The success and failure of OSS Projects have been the subject of many studies and the software modularity is believed to be the major cause of success in these OSS Projects [1]. In order for a particular OSS Project to be successful, the project must allow a new developer to develop a new module [5], and using four possible approaches [6]. The modularity of successful OSS Projects are believed to be better than their proprietary counterparts [7].

Measuring software and specifically its modularity level presents significant challenge. The first successful attempts to measure object oriented metrics are Chidamber and Kemerer [8]. Other proposed methods are the effort estimation [9], and coupling-based structural metrics [10]. In measuring modularity, analytical approach could be used [11]. The integrated measure of modularity in OSS Projects is the Modularity Index [4]. After some observations and considerations, authors are revising the previous formulation which will enhance better comprehension of this software metrics.

3. REVISION OF MODULARITY INDEX
Similar to the previous formulation of Modularity Index, the formulation of the revised Modularity Index should begin from class level, then progresses into the package (module) level and finally into system level. The Modularity Index is intended as the single number to determine the level of modularity in the whole system level.

3.1 Class Level Modularity
There are three main components that are responsible for class level modularity which are LOC Quality (LOCQ), Function Quality (FQ), and Cohesion Quality (HQ). The first two components (LOCQ and FQ) are already introduced previously and the third component (HQ) is a new measure.

The LOCQ is a normalized value that determines the quality of a class based on the number of Non-Commenting Lines of Code (NCLOC) in the class. This value is based on the observation of selected 50 java-based OSS Projects [4] and the formulation is shown here for completeness purpose:

\[ \text{LOCQ} = 0.0138 \times \text{NCLOC} + 0.310 \quad \text{for } \text{NCLOC} \leq 50 \quad (1) \]

\[ \text{LOCQ} = \frac{1}{(\text{NCLOC} - 50)^{0.166}} \quad \text{for } \text{NCLOC} > 50 \quad (2) \]

Where:
- LOCQ = LOC Quality
- NCLOC = Non-Commenting Lines of Code
Similarly, the $F_Q$ is a normalized value that determines the quality of a class based on the number of Functions (methods) in the class. This value is also based on the observation of selected 50 java-based OSS Projects [4].

$$F_Q = 0.1836 \times F + 0.0820 \quad for \quad F \leq 5 \quad \ldots(3)$$

$$F_Q = \frac{1}{(F - 4.83)^{2.691}} \quad for \quad F > 5 \quad \ldots(4)$$

Where:
- $F_Q$ = Function Quality
- $F$ = Function / method

The third component is Cohesion Quality ($H_Q$) which is the normalized value that determines the quality of a class based on its cohesion value (LCOM4). This Cohesion Quality replaces the value LCOM4 which is directly used in the class Quality formulation [4]. By using the selected 50 java-based OSS Project and using the inverse polynomial square fit, the formulation of Coherence Quality is shown in formula 5.

$$H_Q = \frac{1}{LCOM4^{2.604}} \quad \ldots(5)$$

Where:
- $H_Q$ = Cohesion Quality
- LCOM4 = Lack of Cohesion Metrics 4

Finally, the formulation of $c_Q$ is composed of these three components with the weight of each component are differentiated based on the fact that NCLOC and $F$ are found to have high correlation, and the LCOM4 is independent since it has low correlation with NCLOC and $F$.

$$c_Q = 0.25 \times LOC_Q + 0.25 \times F_Q + 0.25 \times H_Q \quad \ldots(6)$$

Where:
- $c_Q$ = class Quality
- LOC$_Q$ = LOC Quality
- $F_Q$ = Function Quality
- $H_Q$ = Cohesion Quality

The class Quality is a normalized value that determines the quality of a particular class. The maximum value of class Quality is achieved when a class has 50 NCLOC, 5 Functions (methods) and perfect cohesion (LCOM4 equals to 1).

Table 1 shows the summary of differences between the previous and the revised version of class level modularity.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Previous Version</th>
<th>Revised Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_Q$</td>
<td>unchanged</td>
<td>Composed of LOC$_Q$, $F_Q$ and LCOM4</td>
</tr>
<tr>
<td>$F_Q$</td>
<td>unchanged</td>
<td>Composed of LOC$_Q$, $F_Q$ and $H_Q$ as shown in equation 6</td>
</tr>
<tr>
<td>$H_Q$</td>
<td>not used</td>
<td>introduced to replace the LCOM4</td>
</tr>
</tbody>
</table>

### 3.2 Package Level Modularity

There is no difference for formulation of Package Quality. The Package Quality is the average of class quality in that package:

$$P_Q = \frac{\sum_{i=1}^{j} c_Q}{\sum_{i=1}^{j} c_i} \quad \ldots(7)$$

Where:
- $P_Q$ = Package Quality
- $c_Q$ = $i$-th class Quality
- $c_i$ = $i$-th class

Even though there is no observed similarity of the number of classes in each Package, it can be observed from the 50 selected OSS Projects that most OSS Projects has the 10 to 16 classes per Package.

### 3.3 System Level Modularity

Similar to the previous publication [4], the components for system level modularity are System Architecture ($S_A$) and Package Quality ($P_Q$). The value of $S_A$ determine the quality of the architecture of the system, this value is determined by two parameters which are Package Coupling and Package Cohesion. Package Coupling ($C_{ij}$) is defined as the count of dependency from classes in one package to the classes in other packages in the system, whereas the Package Cohesion ($C_i$) is defined as the count of dependency from classes in one package to classes in that packages including dependencies into the class itself. The formulation of the $S_A$ is based on the formulation of similar value but using entropy measures [12].

$$S_A = \frac{\sum_{i=1}^{d} c_i}{\sum_{i=1}^{d} c_i} \quad \ldots(8)$$

Where:
- $S_A$ = System Architecture
- $c_i$ = Package Cohesion
- $C_{ij}$ = Package Coupling (if $i \neq j$)

The value of $S_A$ is a normalized value from 0 to 1. The 0 value means that the system has none of Package Cohesion, whereas the maximum value of $S_A$ is achieved in a system with have many of Package Cohesion and none of Package Coupling.

### 3.4 Formulation of Modularity Index

The Modularity Index defined as the product of System Architecture and the Average of Package Quality.

$$M_I = S_A \times \frac{\sum_{i=1}^{j} P_Q}{\sum_{i=1}^{j} P} \quad \ldots(9)$$

Where:
- $M_I$ = Modularity Index
- $S_A$ = System Architecture
- $P_Q$ = $i$-th Package Quality
- $P_i$ = $i$-th Package

The Modularity Index is a normalized value with possible value from 0 to 1. The maximum value of $M_I$ is reached when
Sα is one and average P0 is also one. The average P0 of one is achieved when all of the Package Quality in the system are also one.

This formulation of Modularity Index is different from the previous formulation in which the value of M1 was always increasing as the number of Packages increases [4]. This new formulation should increase the comprehensibility of the software metrics and also makes it possible to compare the Modularity Indexes of java-based OSS Project with different sizes (classes and Packages).

Table 2 shows the summary of differences between the previous and the revised version of system level modularity.

<table>
<thead>
<tr>
<th>Table 2. Comparison in class level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package Coupling</td>
</tr>
<tr>
<td>Package Cohesion</td>
</tr>
<tr>
<td>Sα</td>
</tr>
<tr>
<td>M1</td>
</tr>
</tbody>
</table>

4. CASE STUDY: FREEMIND
To demonstrate the effectiveness of the revised Modularity Index, this measure and its components are then used to analyze the evolution process of a java-based OSS Projects that have high number of downloads in sourceforge.net portal which is Freemind mind mapping software. The analysis is based on the measurement of software metrics indicating modularity which are NCLOC, Packages, classes, average class per Package, Functions, average Function per class, average NCLOC per class, Sα, average P0, and finally M1.

Freemind is a Java-based OSS Project used to draw mind mapping. Mind mapping, which is an alternative way to perform brainstorming by using graphical representation in the paper or digital document, in the modern era is popularized by Tony Buzan [13]. The latest version of Freemind is version 0.9 with many of unstable versions before the milestone version 1.0.0 which is not yet reached. The study of the evolution of Freemind should give some insight about the evolution process before an OSS Project reach version 1.0.0 which should be the project's first significant milestone.

The data collection process is using SONAR (http://www.sonarsource.org) which is an application suite to collect many software metrics in Java-based Projects. In order for a particular version of a project to be analyzed by SONAR, the source code should be able to be compiled using ANT in which the measurement using SONAR called within build.xml build script. There are 22 versions that are collected and compiled starting from version 0.4 dated 7 July 2001 until version 1.0.0 Beta 8 dated 7 October 2012. Table 3 shows the detail of the Freemind versions that are collected and measured.

<table>
<thead>
<tr>
<th>Table 3. Lisf of Freemind Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
</tbody>
</table>

4.1 Evolution of NCLOC
The mostly used software metrics showing the size of a project is Non-Commenting Lines of Codes (NCLOC). Figure 1 shows the evolution of NCLOC of 23 versions of Freemind.

![Fig 1: Evolution of NCLOC in Freemind](image-url)
Above figure shows that the NCLOC of Freemind has grown 11-fold from 5835 in version 0.4 to 65,408 in version 1.0.0 Beta 8. The growth of NCLOC from version 0.8.0 to 0.8.1 is the same (other metrics also shows the identical value) showing the vacuum period of the project for about 2.5 years. The growth of NCLOC in the beta versions of Freemind stabilized nearing 65K showing the increasing maturity of the source code.

4.2 Evolution of Packages
Modularity Index defines a 'module' in java-based OSS Projects is a Package. Figure 2 shows the evolution of Packages in Freemind

![Fig 2: Evolution of Packages in Freemind](image)

The number of Packages has grown from 9 packages in version 0.4 to 50 packages in version 1.0.0 Beta 8 indicating the significant feature improvement of the project. The growth of Packages in the beta versions is showing stability in the number of modules indicating the stability in the number of features in the project.

4.3 Evolution of classes
Figure 3 shows the evolution of classes in Freemind OSS Project.

![Fig 3: Evolution of classes in Freemind](image)

The number of classes has grown from 104 in version 0.4 to 898 in version 1.0.0 Beta 8. The number of classes is also showing stability point in the beta versions of Freemind indicating the stability of the source codes.

4.4 Evolution of average classes per Package
Figure 4 shows the evolution of the average class per package in Freemind.

![Fig 4: Evolution of average classes per Package in Freemind](image)

The average classes per package have shown an interesting result. The value had risen significantly from version 0.4 to version 0.7.1. Starting from version 0.8.0 onward, the number of classes per package has risen moderately until it reach a stable value at 18 classes per package in 1.0.0 beta versions. This number is slightly higher than from the common values from our previous observations [4] of between 10 – 16 classes per package.

4.5 Evolution of average Function per class
Figure 5 shows the evolution of average Function (method) per class in Freemind.

![Fig 5: Evolution of Function per class in Freemind](image)

Above figure shows that the average Function per class varies as the version number increases. In version 0.8.0 onward, the average number of Function per class has stabilizes at about 7 Functions per class. The optimal value of Function per class based on our previous observation [4] should be about 5, so that the value of 7 Functions per class is higher which indicated un-optimized coding practices.

4.6 Evolution of NCLOC per class
Figure 6 shows the evolution of average NCLOC per class in Freemind
Above figure shows that average NCLOC per class in Freemind tends to increase as the version number increases. The average NCLOC per class is still in acceptable values which are about 80 NCLOC per class.

4.7 Evolution of average $P_Q$

Figure 7 shows the evolution of average Package Quality in Freemind.

$S_A$ in Freemind are stabilizing at value 0.7 at the latest versions of Freemind. This value is a high value of system architecture showing the dependency structure of the project already implementing the “maximize cohesion and minimize coupling” principle.

4.8 Evolution of $S_A$

Figure 8 shows the evolution of System Architecture value in Freemind.

Finally, figure 9 shows the evolution of Modularity Index in Freemind.

The weaknesses of the Freemind projects are mainly shown in the coding practices. The average value of classes per Package is about 18 classes per Package which is considered too high compared to the standard value of 10 – 16 classes per package. The number of Functions (methods) per class of
about 7 is also too high compared to the standard value of 5 Functions per class.

5. CONCLUSIONS
In this paper, the revised Modularity Index is presented, which is a composite software metrics to measure the modularity level of a java-based OSS Projects. There are several modifications from the previous formulation, such as the introduction of Cohesion Quality (H0) instead of using LCOM4 cohesion metrics directly, and also the new formulation of Modularity Index which is the product of System Architecture (S\textsubscript{i}) and average Package Quality (P\textsubscript{o}). The new formulation makes the Modularity Index value is a normalized value with possible value between 0 (no modularity) to 1 (perfect modularity).

To show the effectiveness of the revised Modularity Index, the evolution of Freemind is analyzed using this software metrics and the supporting metrics. It can be shown that the Freemind OSS Project has a high value of System Architecture (S\textsubscript{i}) but a low number of average Package Quality (P\textsubscript{o}) which causes a low number of Modularity Index. The improvement that should be made to the project such as the reduction of the average Function per class and the reduction of the number of classes per Package while maintaining the high level of System Architecture.

Our future research should include the further analysis of evolutions in other java-based OSS Projects using Modularity Index. The application of Modularity Index in other object-oriented software project such as using C sharp, C++, and OOP PHP are also possible.

6. ACKNOWLEDGMENTS
Authors would like to thank Maranatha Christian University (http://www.maranatha.edu) that provides funding for the research.

7. REFERENCES