The Impact of Design-level Class Decomposition on the Software Maintainability

Bayu Priyambadha[1], Tetsuro Katayama[2]
Interdisciplinary Graduate School of Agriculture and Engineering, University of Miyazaki, Miyazaki, Japan[1,2]
Faculty of Computer Science, Universitas Brawijaya, Malang, Jawa Timur, Indonesia[1]

***Abstract*—The quality of the software's internal structure tends to decay due to the adaptation to environmental changes. Therefore, it is beneficial to maintain the internal structure of the software to benefit future phases of the software life cycle. A common correlation exists between decaying internal structures and problems like software smell and maintenance costs. Refactoring is a process to maintain the internal structure of software artifacts based on the smell. Decomposition of classes is one of the most common refactoring actions based on Blob smell performed at the source code level. Moving the class decomposition process to the design artifact seems to affect the quality and maintainability of the source code positively. Therefore, studying the impact of design-level class decomposition on source code quality and software maintainability is essential to ascertain the benefits of implementing design-level class decomposition. The metrics-based evaluation shows that the design-level class decomposition positively impacts the source code quality and maintainability with the rank biserial value is 0.69.**

Keywords*—Design level refactoring; class decomposition; class diagram decomposition; software quality; software internal quality; software maintainability

I. INTRODUCTION

Internal structure quality maintenance is an essential task in every phase of the software development process. Due to software changes, software's internal structure or design tends to decay or decrease quality [1]–[3]. Some software is changed without using a modern software engineering approach. As a result, it may never be well structured and optimized for understandability. A software developer often changes the software artifact for the short-term goal. The changes are made mostly not followed by a comprehensive analysis of the existing artifact structure. Sometimes, the changes in one unit of software artifact require adjustment at the other unit, so the structure is always maintained. It is not uncommon that the decaying condition of the software structure will have other impacts, such as the immerging of smell on the software artifacts.

Finding and solving the smell problems in the software artifact is a software research field that continues growing until now [2], [4]–[7]. Many approaches are immersed in detecting the smell in the source code artifact [2]. Also, several researchers have defined how to solve every type of smell in source code without altering the outside behavior, known as Refactoring. The smell term was also started to understand at the level of design phases (software developing process) [4]–[6]. Design smell is defined using the knowledge that lies in the design artifact, for example, the class diagram or the other architectural diagram that expresses the software architecture. But it is still a lack of refactoring process at the design-level artifact using the terms of design smell due to the abstract information in the design artifact [1], [8].

A previous study was conducted to do the refactoring process at the design artifact. The studies focused on the Blob smell that was identified in the class based on the data from the class diagram. Class decomposition is the process of refactoring to solve the Blob smell in the class diagram. Using the threshold-based agglomerative hierarchical clustering algorithm that is based on the class diagram metrics the class decomposition shows promising results [8]. The class diagram metrics, in this case, are syn and sem, representing syntactic and semantic metrics, respectively. Those metrics measure the relationship between class elements. Then, the approach was enhanced by using the evaluation process to solve the misplaced element and unusable class [9]. The research has shown promising results and is worth continuing to the other pathway solution than the Blob smell.

Before it is continued to the other pathway refactoring solution in the design-level artifact, it is better to know the impact of existing approaches to software maintainability. The existing refactoring process at the source code level has already been proven to have a good impact on software maintainability [10], [11].

This research examines the impact of design-level class decomposition on software maintenance. The decomposition process will be carried out using class diagrams. Once the decomposition recommendations are generated, it will be implemented into the source code. After the code has been implemented (class decomposition), several software metrics are used to measure the quality of a piece of source code after its implementation. Measurement results are compared, before and after decomposition, to determine whether there are any differences or effects on software maintainability.

The following section will describe the literature study (Section II) and the whole method of the design-level class decomposition. The class decomposition process on the class diagram is described in Section III. Section IV describes the current experiment scenario to know the impact of the design-level class decomposition on software maintenance. Finally, Sections V and VI represent the result of the experiment, the analysis using the statistical approach, and the overall conclusion of this experiment.

II. LITERATURE STUDY

Maintaining the software's internal structure also benefits the future phase of the software life cycle. The software life cycle is not only one cycle and finish. Mostly it will continue cycling as long as the user and environment need the software. During the cycle, the software experiences change due to user needs and environmental changes. The changes must be applied to the software to prevent the existing software in the specific environment. The software changes require costs we must pay [12], [13]. Therefore, the software engineer must maintain the internal structure to make changes easier and not costly. The bad structure makes the artifact difficult to understand, change, and maintain.

Every effort has been made to maintain the software's internal structure's quality, starting from the source code. Therefore, shifting from the source code to the design level of Refactoring is considered worth doing to increase the quality awareness of the design artifact as early as possible. But, refactoring activity at a higher level of abstraction has a specific problem [1], [8]. The design artifact contains less information than the implementation artifact. Therefore, excavating or mining the design artifacts and analyzing their in-depth information is necessary.

Generally, design artifacts are only written (text-based) with information that is contained within them. However, information may sometimes contain hidden meanings that require further analysis to be understood. The use of natural language processing (NLP) or semantic analysis (SA) is one approach that can provide functionality for understanding the meaning of information [14].

In contrast, source code-level information clearly provides complete information about the source code profile [1] for example, the number of operands or operators in the source code can be measured to determine the complexity of the source code by the software engineer. In addition, by reviewing the internal source code, the developer will be able to understand the relationship between attributes and methods. They can review the assigning value statement to determine the relationship between the method and attribute.

On the side of design, a review and assessment of the quality of the artifact design can be carried out by utilization of NLP and SA. Furthermore, the refactoring activity, such as class decomposition, using the design artifact is also possible using the NLP and SA analysis [1], [8].

Shifting the refactoring activity to the design artifact is expected to contribute and positively impact software maintenance [11], [15]. There are two fundamental theories of the research of design-level software refactoring. First, the theories of Software Evolution (specifically in software refactoring) [2] and the second is Model-Driven Software Engineering (MDSE) [16] Software refactoring preserves the quality of the software's internal structure in relation to the software evolution, specifically in the case of software maintainability. On the other hand, MDSE provides the concept that software development is oriented on a model. Therefore, the model acts as the core of action and the guidance of the implementation phase. In other words, the model is the bridge between requirement analysis and implementation. The previous research on design-level Refactoring aims to combine the theories (Refactoring and MDSE) to propose a better approach for refactoring for better software quality [1]. It also has the aim to increase the awareness of internal quality as soon as possible. The other reason for the shift to the design phase is to gain the benefit of MDSE. Fig. 1 shows the thinking schema of the approach according to the justification or rationale. Furthermore, the impact of the Refactoring on the design phase is needed to be investigated.

MDSE uses a software model as the primary artifact of software development [16]. Compared to the implementation artifact (source code), the software model is closer to the problem domain. The model transformation is the main process of the MDSE since the MDSE aims to generate the source code from the models. On the other hand, there is another approach to the development of software called Code-centric Development (CcD). A comparison study between MDSE and CcD has been done for over a decade. From the review paper by Domingo et al., many researchers have been evaluating the benefit of the MDSE [17]. Some works said that MDSE decreases development time (up to 89%) relative to Code-centric Development (CcD). The other works suggest that the MDSE is suitable for academic exercise. Furthermore, the other works assert that MDSE is also suitable for inexperienced developers. Finally, Domingo et al., based on their review of the MDSE, conclude that the MDSE is suitable for academic exercise and inexperienced developers. It would be beneficial to move refactoring activities to the design artifact utilizing the benefits of MDSE.

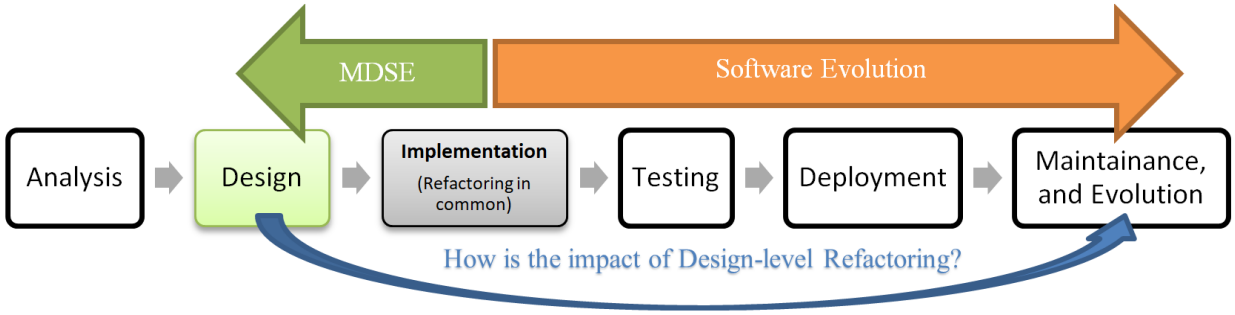

Fig. 1. Thinking schema of design-level class decomposition.

On the other side of view, the impact of the design-level refactoring process on maintenance is questionable and needs study. There are several methods to measure software maintainability. Software maintainability, the ease with which a software system can be understood and modified to accommodate bug fixes, new features, and improvements in general, plays an important role in software quality. Since it is difficult to measure software maintainability without measuring the actual maintenance process, researchers and practitioners often use product metrics as indicators. Some practitioners sometimes face difficulty in finding suitable metrics, specifically software maintainability metrics, according to the practitioner's scope or subjective. The paper of Saraiva et al. has aimed to propose Object-Oriented Software Maintainability (OOSM) metrics categorizations. It aims to make it easy to find suitable maintainability metrics [18]. Maintainability is related to external quality attributes such as analyzability, changeability, stability, and testability. All external attributes are expressed as maintainability characteristics that are "easy to adapt." To achieve this maintainability objective, most researchers measure from the internal quality attributes at least the system's size, complexity, coupling, and cohesion [18].

The other metric that points to software maintainability is named Maintainability Index (MI) [19]. MI is a software metric that is used to measure the level of software, whether it is easy or difficult to maintain in the future. The MI is calculated by considering the Line of Code (LOC), Cyclomatic Complexity, and Halstead Volume (HV). MI is measured using the information of software source code.

III. Design-Level Class Decomposition

This section describes how the refactoring activity can perform in the design-level artifact. This research uses the class diagram as the main object in the refactoring process. The refactoring activity on the class diagram utilizes the class information extracted from the class diagram. One of the challenges in this research is using the existing information to do the refactoring activity. Fig. 2 explains the proposed design-level class decomposition approach to solve Blob smell in class (class diagram).

A. Design-Level Information Extraction

The first task in this approach is to collect or extract information from the class diagram. The class diagram is a notation-based diagram that is expressed as an image showing the software's architecture in case of a class arrangement, and it is a static diagram. The information in the class diagram is important to collect and analyze to support the process. The information extraction needs a specific strategy to make it easy to implement.

The first step in this process is converting the class diagram to an XML formatted file. The conversion aims to transform the notation based to text-based information. Then, the information (text-based) is extracted from the XML syntax to get important data for smell detection (the next process). The extraction uses a specific search algorithm called Tree-based keyword search [20].

The candidate data (classes) for smell detection consists of seven pieces of information. The data are the number of attributes, number of methods, number of relations between methods and attributes, number of relations between methods and attributes, number of relations between attributes and attributes, the capacity of the relationships within the class, and the degree of cohesion [21]. The NLP and SA determine the relationship between the class elements. In addition, the degree of cohesion is calculated based on the relationship between the class elements [22].

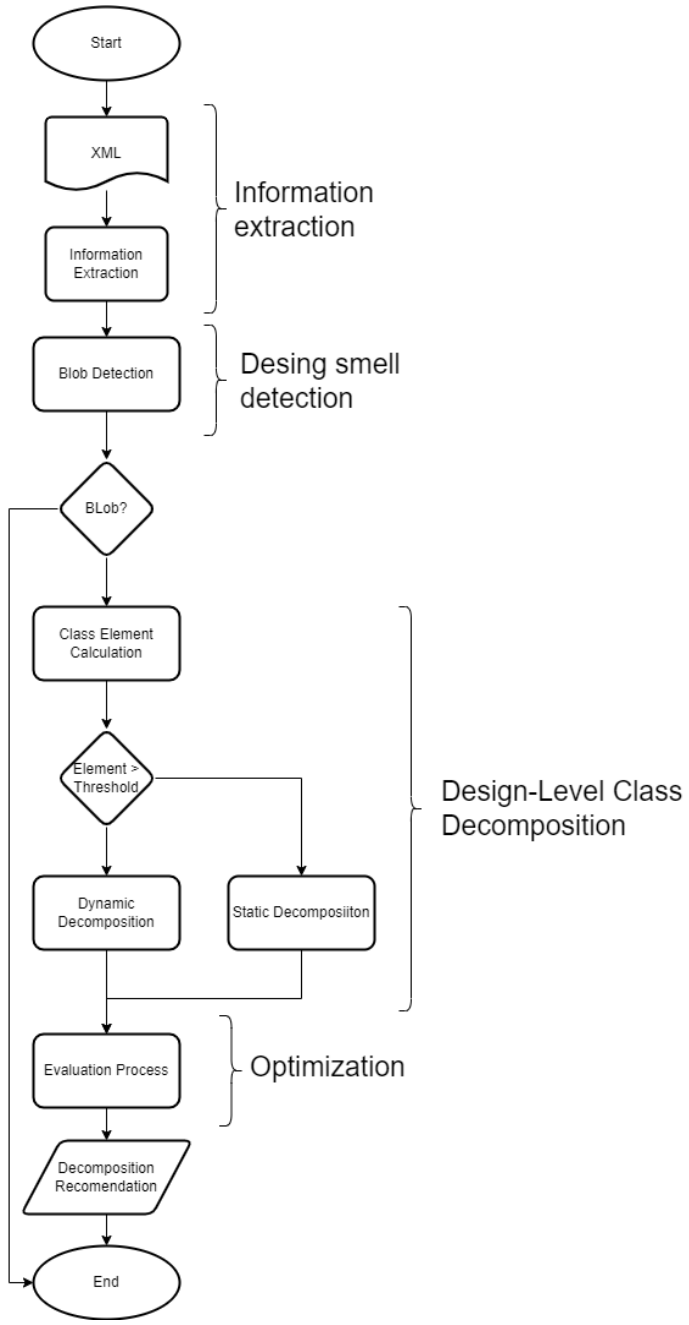

Fig. 2. The design-level class decomposition (design-level refactoring).

B. Design Smell Detection

The classification method will be used to detect the smell. The experiment used three classifiers: the j48, the Multi-Layer Perceptron, and the Naïve Bayes. The experiment uses Weka as machine learning software to solve data mining problems and run using a basic configuration. Classifiers are used to demonstrate that the dataset can be distinguished from bad smells (Blob and Feature Envy) [21]. The experiment result shows an average accuracy of 80.67% for the Blob smell. The accuracy value means that the information data can be utilized for the Blob detection process in a class diagram.

The information (seven pieces of information from the class diagram) contained characteristics that led to the identification of the Blob smell [21]. Hence, the Blob smell is being addressed in the Refactoring.

C. Design-Level Class Decomposition

Once the Blob smell is detected, Refactoring must be performed to solve the smell. Based on the experiences, the Blob smell is solved using the class decomposition or extraction. Mostly the research about class decomposition is done at the level of source code [2], [23]–[30]. Then based on the information in the class diagram [21], the class decomposition is done.

The design-level class decomposition in this research uses threshold-based agglomerative hierarchical clustering. It is divided into static and dynamic threshold hierarchical clustering. Static thresholds differ from dynamic thresholds in defining the threshold before decomposing the cluster. In the static approach, the threshold value is defined at the beginning of the decomposition process (the threshold is defined only once). According to the dynamic approach, the threshold is calculated at every stage of the decomposition process. In this study, Hamdi's algorithm is used, but it is implemented at the level of design [29]. Shifting objects to class diagrams requires defining new metrics for clustering. Syntax (syn) and semantic (sem) aspects of the class element's label are considered in the similarity matrix to do the clustering process (class decomposition) [8]. The two aspects (syntax and semantics) are considered due to the nature of class diagram information, which is more abstract than source code information. To determine the relationship between class elements, it has to determine the closeness meaning of the label name between elements. This seems to be the essential approach in this process.

The process shows the promising result of decomposition (based on the Silhouette value). But, there are still shortcomings to solve. Some elements still have a negative Silhouette value in the decomposition result. Negative Silhouette values indicate that the current element is far from other cluster elements or has the wrong placement. Additionally, the negative Silhouette elements are considered to be the least desirable.

The results also indicate that some clusters are considered unimplementable due to the possibility that they may produce objects that cannot collaborate with each other. A cluster with only one element, particularly if the element has a private modifier, is considered useless. Evaluating the moving mechanism of the negative element is considered important as an optimization process. The next process is the evaluation process to the result of this process.

D. Optimization of Class Decomposition Result

The result of the decomposition from the previous process has to be optimized to solve the negative element and unusable cluster. The elements are evaluated by considering the value of Silhouette ($s(i)$) and class usability ($CUsability$). $CUsability$ value calculated based on the existence of the public method inner the cluster (value one if exists and 0 if not). Then to evaluate the cluster and elements, the following formula is used [9].

$$Eval = a.s(i) + b.CUsability \quad (1)$$

Where a and b are the weight of every factor to adjust during the experiment. Threshold-based agglomerative hierarchical clustering experiment has been optimized by adding an evaluation process. During the evaluation process, a specific element with a negative Silhouettes value in each cluster is intended to be moved to a better cluster.

In comparison to the previous approach, the evaluation process increases the average Silhouettes of the cluster by using a higher or equal to 0.7. There has been an average increment of Silhouettes of about 40% [9]. Based on the results of the previous approach, the evaluation process is also able to solve the unusable cluster.

Finally, the whole process produces a set of clusters that represent the classes as the result of the decomposition. In this step, the result is the recommendation to be implemented at the source code level. Then, the impact of the decomposition implementation on the source code level will be described and analyzed in the following section.

IV. Experiment Scenarios

Understanding the impact of the refactoring result on the software maintainability is essential. The impact of class decomposition recommendation result (design-level class decomposition) on the quality of source code is the way to know how is the performance of the design-level class decomposition approach.

A. Overview of Scenario

The decomposition of the class on the source code level based on the recommendation from class decomposition on the design level aims to compare the quality. The source code quality before and after implementing the class decomposition recommendation will lead us to conclude how the design-level decomposition will impact the source code. The quality of code is measured using the source code quality metrics. Fig. 3 shows how the experiment is held to study the impact of design-level class decomposition. The following five internal quality attributes are associated with software maintainability: size, complexity, coupling, cohesion, and constraints associated with software architecture [18]. Four of the internal quality attributes are expressed in the 18 metrics. Table I shows the list of the 18 metrics used in this experiment to compare the before and after decomposition process. The purpose of this experiment is not only to compare the quality of source code

based on the 18 metrics but also to compare the MI [19] as the final proof of this research.

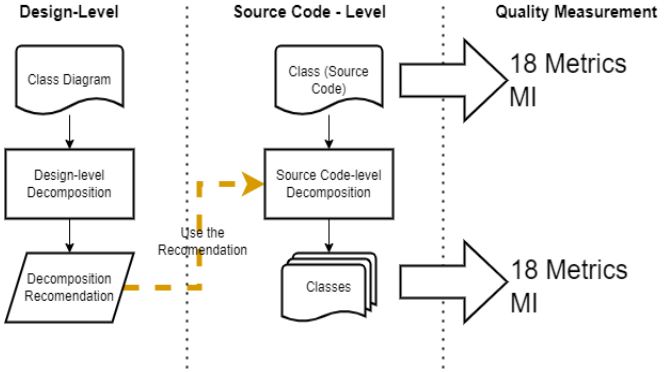

Fig. 3. The experiment scenario.

TABLE I. THE LIST OF THE 18 METRICS

No.	Name	Description
1.	CBO	Coupling Between Object Classes, the number of coupled classes (Coupling)
2.	RFC	Response For a Class, the number of methods that can be potentially invoked in response to a public message received by an object of a particular class (Complexity)
3.	SRFC	Simple Response For a Class, the number of methods that can be potentially invoked in response to a public message received by an object of a particular class (Complexity)
4.	DIT	Depth of Inheritance Tree, the position of the class in the inheritance tree (Complexity)
5.	NOC	Number of Children, the number of direct subclasses of a class (Coupling)
6.	WMC	Weighted Method Count, The weighted sum of all class' methods and represents the McCabe complexity of a class (Complexity)
7.	LOC	Line Of Code (Complexity, Size)
8.	CMLOC	Class-Method Lines of Code, Total number of all nonempty, non-commented lines of methods inside a class (Complexity, Size)
9.	NOF	Number Of Fields, the number of attributes in class (Complexity, Size)
10.	NOSF	Number of Static Fields, the number of static attributes (Complexity, Size)
11.	NOM	Number of Methods (Complexity, Size)
12.	NOSM	Number of Static Methods (Complexity, Size)
13.	NORM	Number of Overridden Methods (Complexity)
14.	LCOM	Lack of Cohesion of Methods, measure how methods of a class are related to each other (Cohesion)
15.	LCAM	Lack of Cohesion Among Methods (1-CAM), CAM metric is the measure of cohesion based on parameter types of methods (LCAM = 1-CAM) (Cohesion)
16.	LTCC	Lack of Tight Class Cohesion, The Lack of Tight Class Cohesion metric measures the lack of cohesion between the public methods of a class (Cohesion)
17.	ATFD	Access to Foreign Data, the number of classes in which the attributes are directly or indirectly reachable from the investigated class (Coupling)
18.	SI	Specialization Index measures subclasses override their ancestor's classes (Complexity)

All the measurement results are collected to be analyzed in the following step. For the result of measurement using the 18 metrics, the data will be recapped and show the trend of comparison before and after decomposition.

Lastly, for measuring MI, statistical analysis is needed to determine the impact of the decomposition recommendation on the source code quality.

B. Experiment Data

TABLE II. MI CLASSIFICATION

MI Value	Classification
>85	Highly maintainable
>65 and ≤85	Moderate maintainable
≤65	Difficult to maintain

This experiment used two study cases, jHotDraw and AgroUML source code. There are 67 classes identified as Blob classes using jDeodorant in both applications. But, after measuring the MI, not all classes are considered problematics in maintenance. The classification of MI value refers to Table II, which explains how the value is classified based on maintainability [19] There are only 33 classes that have moderate and difficult to maintain. Therefore, only 33 classes are used as the object in this experiment. The acquisition of data is shown in Fig. 4.

The Blob classes classified as highly maintainable are not used in this experiment because it assumed not to be included in problematic classes in the maintainability manner.

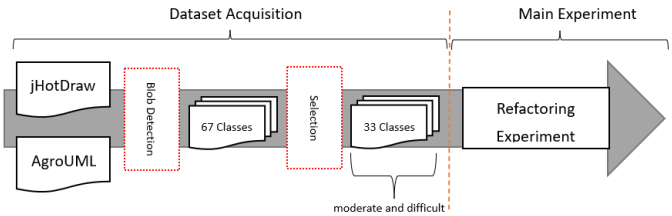

Fig. 4. Data acquisition.

But, it has to solve from the other perspective manner.

C. Tools

There are three tools used in this experiment, the jDeodorant plugin for Eclipse IDE [31], the CodeMR (Code Magnetic Resonance), and the prototype application that implements the MI measurement [19].

The jDeodorant is used in the data acquisition process to select the classes that contain a Blob smell. The CodeMR is the application that has the ability to measure the quality of source code based on the 18 metrics measurement. CodeMR is a static analysis tool for source code. And the last is a custom application that can show the value of the Maintainability index of source code. CodeMR and the custom application used on the before and after decomposition process. The result of measurement is recapped and analyzed to know how the impact of the usage design-level class decomposition recommendation on the source code quality.

V. Experiment Result and Discussion

The class decomposition recommendation is implemented on the source code to get the benefit of it. The result is measured using 18 metrics and MI to know how difference before and after the decomposition process.

A. Measurement using the 18 Metrics

The source code decomposition result is measured using the 18 metrics for the first result. The 18 metrics are grouped into four metrics based on the metrics type. Every value of each metric is calculated by averaging the values of particular metrics in every case study. Then, it differentiated before and after decomposition. The groups are coupling, complexity, cohesion, and size metric. Fig. 5 shows the result of the measurement that is described on the line graph. Another reason for grouping metrics is that each type of metric has a different range of values, so separating each type into groups will clarify trends for each type of metric.

Fig. 5(a) shows the trend of measurement in the type of coupling metric. There are three metrics in the category coupling metric, CBO, NOC, and ATFD. In this result, CBO shows a decrement value from before to after decomposition.

The other metrics, NOC and ATFD, do not show decrement due to the value equality between before and after decomposition.

Fig. 5(b) shows the group of complexity metrics consisting of ten metrics. The metrics are RFC, SRFC, DIT, WMC, SI, NOF, NOSF, NOM, NOSM, and NORM. Those metrics measure the complexity of source code from several sides. For the ten metrics, the graph shows the trend that the values decrease after decomposition. Two metrics show the same value before and after decomposition. The metrics are SI and NORM that has a value of 0 before and after decomposition.

Fig. 5(c) shows the cohesion metric, consisting of three metrics: LCOM, LCAM, and LTCC. All metrics show the measurement of a lack of cohesion in the class. Higher values show a higher lack of cohesion in the source code. The value of those metrics decreases before and after the decomposition process.

Fig. 5(d) shows the size metric, which measures the size of the source code. It seems to be the same trend as the other type of metrics. The value of LOC and CMLOC, before and after decomposition, decreases due to the result of decomposition implemented to the source code.

All metric types show the same trend that, after decomposition, tend to be lower value of metrics. The 18 metrics show the same meaning of the value: the lower value means the better condition of the source code. Implementing class decomposition on the source code seems to make the source code better quality measured by the 18 metrics.

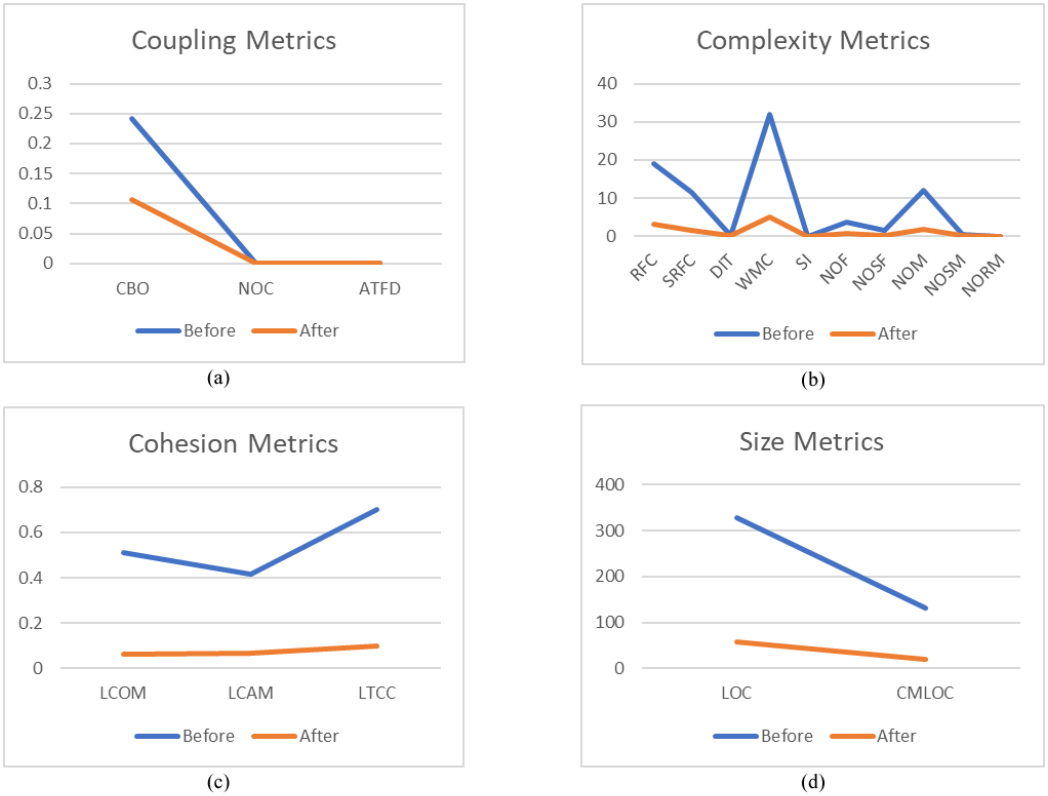

Fig. 5. Result of measurement using the 18 metrics of a) Coupling metrics, b) Complexity metrics, c) Cohesion metrics, and d) Size metrics.

B. Statistical Analysis of the Maintainability Index (MI)

The Maintainability Index (MI) measurement is done using the prototype application. The measurement is applied before and after the class decomposition process based on the design-level recommendation. The MI value for the after-decomposition is calculated by averaging each class's value aims to represent one value. Table III shows the result of the measurement of MI compared before and after the decomposition process.

In this experiment, the differences between before and after are worth calculating to ensure the differences after decomposition. The differences calculation uses the statistical approach, in this case, Wilcoxon signed-rank. Analysis differentiation aims to make sure that there is a difference between before and after decomposition. Differentiation can be used as a sign that the decomposition process causes an impact on the source code in case of maintainability.

Besides the differences, how strong the effect of the usage of design-level decomposition recommendation in the source code level decomposition to the value of MI is also important to know. The Wilcoxon signed rank is able to inform both the differences and the effect size of the approach.

Based on the result of the Wilcoxon signed rank, there are several interpretations based on the test result. Fig. 6 shows the first indicator by the p-value of the result. The significant value of differential analysis is lower than the 0.05 p-value. The current result shows that the p-value is 6.02e-04, lower than 0.05. So, based on the p-value, the result concluded that the MI before and after the decomposition process is significantly different.

The second indicator is the median value from the plot in Fig. 6. Even though the median value cannot act as the main indicator of differentiation and it shows how the differences in spreading data differ. Based on the result, the median values before and after decomposition differ in favor. The median of after decomposition data is increased by 25.03 to the before.

This research aims to know how the impact of the utilization of design-level class decomposition recommendations on the source code level. The p-value and median value only show that the data before and after is different. It does not show how the impact of the design-level class decomposition on the source code quality simultaneously. The other value could be used to know how strong the impact design-level decomposition is rank biserial, as shown in Fig. 6. The rank biserial is used to examine the relationship between dichotomous (binary) nominal data and ordinal (ranked) data. Before running the statistical analysis, the data measurement of MI is calculated to find the data rank based on the differences in the value of MI before and after decomposition. It is one of the Wilcoxon sign rank method's requirements before shown in the plot as shown in Fig. 6. The rank biserial value shown in Fig. 6 is 0.69. Therefore, the higher value is better. Based on Funder's interpretation [32], 0.69 can be interpreted as very large. In other words, the use of design-level decomposition recommendation on the source code level decomposition gives a very large, positive, and significant effect on the MI.

TABLE III. MI BEFORE AND AFTER DECOMPOSITION PROCESS

Differentiation of MI			
No.	Class Name	Before	After
1.	ArgoEventPump	80.78	99.02
2.	ArgoJFontChooser	82.76	100.13
3.	ArgoParser	84.66	116.53
4.	DetailsPane	80.38	97.84
5.	DrawApplet	75.41	120.47
6.	DrawApplication	68.84	122.26
7.	ExplorerPopup	62.42	91.33
8.	FindDialog	66.81	70.96
9.	GenericArgoMenuBar	62.19	89.18
10.	GraphLayout	72.14	72.43
11.	Import	69.4	81.42
12.	MyTokenizer	84.63	98.55
13.	NotationSettings	80.38	107.75
14.	PathItemPlacement	80.93	98.25
15.	PerspectiveConfigurator	63.85	46.43
16.	PerspectiveManager	66.65	88.60
17.	ProfileConfigurationParser	80.97	83.60
18.	ProfileUML	70.41	35.21
19.	ProjectBrowser	57.37	105.78
20.	SettingsTabProfile	53.51	62.48
21.	StandardDrawingView	71.48	110.70
22.	TabConstraints	73.45	104.59
23.	TabStyle	80.92	124.74
24.	TargetManager	71.62	82.97
25.	ToDoList	81.37	125.75
26.	TodoParser	84.07	92.41
27.	UMLActivityDiagram	69.27	101.92
28.	UMLAddDialog	83.8	42.25
29.	UMLDeploymentDiagram	76.53	113.31
30.	UMLStateDiagram	65.81	104.69
31.	UMLUseCaseDiagram	84.26	101.36
32.	UserDefinedProfile	67.57	98.48
33.	WizOperName	77.7	39.25

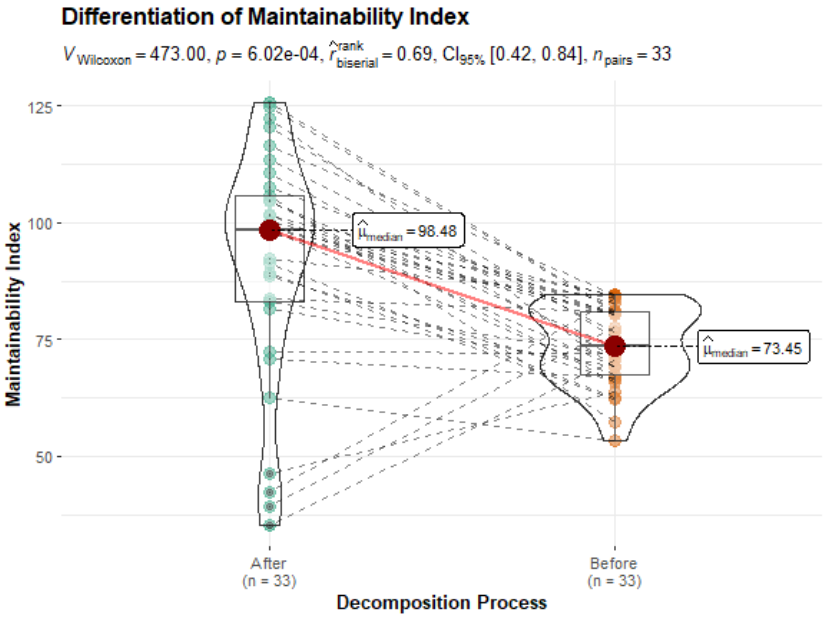

Fig. 6. Differentiation of MI.

C. Research Limitation

This section explains all things that have a possibility to threaten the validity of the experiment result. There are two main limitations to this experiment. First is the dataset used and the manual process conducted in the experiment.

A limited amount of data in the experiment would be the main limitation of this research. The study case is taken from only the two application resources (jHotDraw and jDraw). The dataset collected from both applications is only 33 instant data. It is sufficient for the scope of this experiment, but it is considered better to add the amount of data in the future to increase the result validity.

The process of class decomposition is done automatically. But, the implementation of decomposition recommendations to the source code is done manually based on the location of class elements.

VI. Conclusion

The quality measurement before and after the decomposition process on the source code is by using two approaches. First, the source code was measured using the 18 metrics representing coupling, complexity, cohesion, and size. Those groups of metrics are the type of metrics that are related to software maintainability based on the existing references. There is a trend in all metrics types that after decomposition, the metrics tend to have a lower value than before decomposition. In all 18 metrics, a lower value represents a better condition of source code. By implementing design-level class decomposition on the source code, the source code seems to be of better quality as measured by the 18 metrics.

The second quality measurement uses MI as one specific metric to measure the software maintainability of the source code. The measurement result differentiated before and after the decomposition process. The Wilcoxon signed-rank analysis was applied to the result of measurement to get a deep analysis of the result. A p-value less than 0.05 indicates significant differential analysis in the first test. According to the current results, the p-value is 6.02e-04, which is less than 0.05. Therefore, it is concluded that the MI before and after decomposition is significantly different. The second indicator is the median value from the plot. Regardless of the fact that the median value cannot be used as the primary indicator of differentiation, it does at least indicate how the spread of data differs.

According to the results, the median value before and after decomposition differs in favor of decomposition. After decomposition, the median has increased by 25.03 compared to the before data.

The final test is rank biserial. There is a rank biserial value of 0.69, which can be interpreted as being very large. As a result, using design-level recommendations on source code decomposition has a very large, positive, and significant effect on the MI.

The 18 metrics and MI analysis show the same favorable result. The use of design-level class decomposition recommendation is able to increase the source code quality significantly based on the analysis result.

The shifting refactoring process to the design artifact is still challenging in the future. This is because so many code smell types could detect and refactor from the design artifact. This research only focuses on the Blob smell on the design artifact, only defining the pathway solution based on the existing Blob smell in the class diagram. The research will continue to the other pathway solution than the Blob smell.

This research uses the data collected from the existing open-source application. The limitation on the number of data might be lacking in the meter of data validity. Increasing the number of data is a plan that has been recorded to be carried

out in the future. The complete software documentation will be interesting data to analyze for future research.

REFERENCES

[1] B. Priyambadha and T. Katayama, "Enhancement of Design Level Class Decomposition using Evaluation Process," International Journal of Advanced Computer Science and Applications, vol. 13, no. 8, pp. 130–139, 2022, doi: 10.14569/IJACSA.2022.0130816.

[2] M. Fowler et al., Refactoring Improving the Design of Existing Code Second Edition, Second Ed. United State of America: Pearson Education - Wesley, 2019.

[3] I. Sommerville, Software Engineering, 9th ed. Harlow, England: Addison-Wesley Professional, 2010.

[4] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software Design Smell Detection: a systematic mapping study," Software Quality Journal, vol. 27, no. 3, pp. 1069–1148, 2019, doi: 10.1007/s11219-018-9424-8.

[5] B. K. Sidhu, K. Singh, and N. Sharma, "A Catalogue of Model Smells and Refactoring Operations for Object-Oriented Software," Proceedings of the International Conference on Inventive Communication and Computational Technologies, ICICCT 2018, pp. 313–319, 2018, doi: 10.1109/ICICCT.2018.8473027.

[6] B. Kaur Sidhu, "Model Smells In Uml Class Diagrams," International Journal of Enhanced Research in Management & Computer Applications, vol. 5, pp. 2319–7471, 2016, Accessed: Apr. 10, 2019. [Online]. Available: https://pdfs.semanticscholar.org/bced/a3ff00a0b577007d17abfbb6fd406058def6.pdf

[7] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. in Robert C. Martin Series. Boston, MA: Prentice Hall, 2017.

[8] B. Priyambadha and T. Katayama, "Design Level Class Decomposition using the Threshold-based Hierarchical Agglomerative Clustering," International Journal of Advanced Computer Science and Applications, vol. 13, no. 3, pp. 57–64, 2022, doi: 10.14569/IJACSA.2022.0130310.

[9] B. Priyambadha and T. Katayama, "Enhancement of Design Level Class Decomposition using Evaluation Process," International Journal of Advanced Computer Science and Applications, vol. 13, no. 8, 2022, doi: 10.14569/IJACSA.2022.0130816.

[10] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Empirical study on refactoring large-scale industrial systems and its effects on maintainability," Journal of Systems and Software, vol. 129, pp. 107–126, 2017, doi: 10.1016/j.jss.2016.08.071.

[11] S. Kaur, A. Kaur, and G. Dhiman, "Deep analysis of quality of primary studies on assessing the impact of refactoring on software quality," Mater Today Proc, no. xxxx, 2021, doi: 10.1016/j.matpr.2020.11.217.

[12] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in Proceedings - International Conference on Software Engineering, 2013. doi: 10.1109/ICSE.2013.6606614.

[13] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in Proceedings - 9th International Workshop on Search-Based Software Testing, SBST 2016, 2016. doi: 10.1145/2897010.2897016.

[14] B. Priyambadha, T. Katayama, Y. Kita, H. Yamaba, K. Aburada, and N. Okazaki, "Utilizing the similarity meaning of label in class cohesion calculation," Journal of Robotics, Networking and Artificial Life, vol. 7, no. 4, pp. 270–274, 2021, doi: 10.2991/jrnal.k.201215.013.

[15] S. Kaur and P. Singh, "How does object-oriented code refactoring influence software quality? Research landscape and challenges," Journal of Systems and Software, vol. 157, p. 110394, Nov. 2019, doi: 10.1016/j.jss.2019.110394.

[16] M. Brambilla, Jordi. Cabot, and Manuel. Wimmer, Model-driven software engineering in practice. Morgan & Claypool, 2012.

[17] Á. Domingo, J. Echeverría, Ó. Pastor, and C. Cetina, "Evaluating the Benefits of Model-Driven Development," Advanced Information Systems Engineering, no. June 2021, pp. 353–367, 2020, doi: 10.1007/978-3-030-49435-3_22.

[18] J. D. A. G. Saraiva, M. S. De França, S. C. B. Soares, F. J. C. L. Filho, and R. M. C. R. De Souza, "Classifying metrics for assessing Object-Oriented Software Maintainability: A family of metrics' catalogs," Journal of Systems and Software, vol. 103, pp. 85–101, 2015, doi: 10.1016/j.jss.2015.01.014.

[19] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using Metrics to Evaluate Software Svstem Maintainability," IEEE Computer, vol. 27, no. 8, pp. 44–49, 1994.

[20] B. Priyambadha and T. Katayama, "Tree-based keyword search algorithm over the visual paradigm's class diagram xml to abstracting class information," 2020 IEEE 9th Global Conference on Consumer Electronics, GCCE 2020, pp. 280–284, 2020, doi: 10.1109/GCCE50665.2020.9291865.

[21] B. Priyambadha, T. Katayama, Y. Kita, H. Yamaba, K. Aburada, and N. Okazaki, "The Seven Information Features of Class for Blob and Feature Envy Smell Detection in a Class Diagram," The 2021 International Conference on Artificial Life and Robotics (ICAROB2021), pp. 348–351, 2021.

[22] B. Priyambadha, T. Katayama, Y. Kita, K. Aburada, H. Yamaba, and N. Okazaki, "The Measurement of Class Cohesion using Semantic Approach," Proceedings of International Conference on Artificial Life and Robotics, vol. 25, pp. 759–762, 2020, doi: 10.5954/icarob.2020.os14-4.

[23] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," ASE'10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 151–154, 2010, doi: 10.1145/1858996.1859024.

[24] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures," Journal of Systems and Software, vol. 84, no. 3, pp. 397–414, Mar. 2011, doi: 10.1016/j.jss.2010.11.918.

[25] G. Bavota, "Using structural and semantic information to support software refactoring," Proceedings - International Conference on Software Engineering, pp. 1479–1482, 2012, doi: 10.1109/ICSE.2012.6227057.

[26] G. Bavota, A. de Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," Empir Softw Eng, vol. 19, no. 6, pp. 1617–1664, 2014, doi: 10.1007/s10664-013-9256-x.

[27] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of Extract Class refactorings in object-oriented systems," Journal of Systems and Software, vol. 85, no. 10, pp. 2241–2260, 2012, doi: 10.1016/j.jss.2012.04.013.

[28] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," IEEE International Conference on Software Maintenance, ICSM, pp. 93–101, 2009, doi: 10.1109/ICSM.2009.5306332.

[29] M. Hamdi, R. Pethe, A. S. Chetty, and D. K. Kim, "Threshold-driven class decomposition," Proceedings - International Computer Software and Applications Conference, vol. 1, pp. 884–887, 2019, doi: 10.1109/COMPSAC.2019.00130.

[30] Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, "Automatic Software Refactoring via Weighted Clustering in Method-Level Networks," IEEE Transactions on Software Engineering, vol. 44, no. 3, pp. 202–236, 2018, doi: 10.1109/TSE.2017.2679752.

[31] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in 2011 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1037–1039. doi: 10.1145/1985793.1985989.

[32] D. C. Funder and D. J. Ozer, "Evaluating Effect Size in Psychological Research: Sense and Nonsense," Adv Methods Pract Psychol Sci, vol. 2, no. 2, pp. 156–168, Jun. 2019, doi: 10.1177/2515245919847202.