ISSN No:-2456-2165

Enhanced Class Normalization Rules for Refactoring Large Class Smell

Marwan Ahmed Lardhi
Department of Information Technology
Faculty of Engineering & Information Technology
Al-Rayan University
Hadhramout, Yemen

Saeed Mohammed Baneamoon[1,2]
Department of Computer Engineering
[1]College of Engineering & Petroleum
[2]College of Computers & Information Technology
Hadhramout University
Hadhramout, Yemen

Abstract:- This paper proposes an effective method for optimizing extraction large class smell using enhanced class normalization rules in order to ease maintenance and improve the quality of software by creating new classes with strongly and similarity attributes and shared behavior. The proposed method introduced a technique to extraction a class with many responsibilities that is chosen by the developer or automatically, where is produced an access-set table of attributes, then is calculated the Jaccard similarity measure to create a similarity matrix for attributes. After that is designed the structural similarity matrix of each extracted class to calculate the cohesion of each class. Experimental results show that applying the proposed method for dividing the large class into many cohesive classes provides better performance in software evolution compared to existing methods.

Keywords:- *Extract Class Refactoring, Large Class smell, Class normalization, Cohesion.*

I. INTRODUCTION

Maintenance of software is a component of the life cycle of software development that the primary aim is to modify and update software application after delivery to correct faults and enhance system efficiency where could be easy modifications to correct coding mistakes, more comprehensive modifications to correct design mistakes, or to accommodate new requirements [1]. They're things that impair software quality and make them hard to maintain and evolve like code smells. Code smell is signs inside the code that indicate that there is a design flaw and is not in a software error, where may find codes full of these odors but they work just fine without any problems.

To improve software maintainability, there several refactoring techniques that may apply to source code. Refactoring is a change made to the software's inner structure to make it simpler to comprehend and cheaper to change without altering its behavior such as Move Field, Move Method, Extract Method, Pull Up Field and Extract Class. First, refactoring improves software design where changes to realize short-term goals or changes made without a full understanding of the code's design the code loses its structure, making it more difficult to see the design by reading the code and the poorly designed code which usually requires more code to do the same things. Second, it makes software easier to understand, programming is a write code conversation with a computer that informs the computer what to do, and it reacts by doing precisely what you say and programming in this mode is all about stating precisely what you want, but somebody will attempt to read this code. But there's another user of this source code which in a few months' time someone will attempt to read code to create some changes, which means that additional code user can readily be forgotten. Third thing, refactoring helps to find bugs, since understanding the code can help identify bugs that some can read a bunch of code and see bugs. Lastly, it helps with programming quicker where the whole point of getting a good design is to enable fast development and without a good design can the progress rapidly for a while, but soon the bad design starts slowing down the developer and thus spend time finding and fixing bugs instead of adding a new feature where modifications take longer as an attempt to comprehend the system and discover the duplicate code [2].

Classes usually start small, but over time they become larger as the software expands. As is the case for long methods, programmers usually find it less exhausting mentally to put a new feature in an existing class than to create a new class for the feature and important to improve any program's structure, maintenance and improve performance where refactoring is key to improving both the quality of the code. The extract class refactoring method will help maintain adherence to the single responsibility principle and classes are more reliable and tolerant of changes.

Class normalization techniques are not yet as popular as refactoring or pattern application. Class normalization is a process through which object schema structure is reorganized in such a way that class cohesion is increased with coupling is minimized between classes. The Repeating data structures are refactored into their own class to place a class in the first object normal form (1ONF). When encapsulating the shared behavior required by multiple entities within its own class, a class is in the second object normal form (2ONF). A class is in the third object normal form (3ONF) when implementing a single, cohesive set of behaviors [3].

II. RELATED WORKS

Marcus et al. [4] proposed a new Object-Oriented (OO) software class cohesion measure based on the analysis of unstructured information embedded in the source code, which called the Class Conceptual Cohesion (C3) such as comments and identifiers, where structural and conceptual metrics are combined to provide better models for classes faults prediction than combinations of structural metrics alone. In this approach was noted that does not take into account polymorphism and inheritance, and its reliance on the existence of naming's conventions for the relevant identifiers and comments, and when these are missing, the effect on measuring the coherence of the class appears.

Bavota et al. [5] proposed a method based on graph theory that exploits structural and semantic relationships between methods in a class to be refactored that to build two new classes having higher coherent than the original class. Bavota et al. [6] came back to update of the previous work, where they presented a method chains used to define new classes with a higher coherent than the original class, while preserving the overall coupling between the new classes and the classes that interact with the original classes. This study distinguished its ability to increase the strength of cohesion of classes without a significant increase in coupling, but relying on generalized results from master's sample experience poses a threat.

Al Dallal [7] proposed a model that was applied to automatically predict the classes that need ESR and present them as suggestions for developers working to improve the system during the maintenance phase, as the models created using studied quality metrics showed high capabilities to separate the classes in those that were in need and those that did not need to resell housing. This model was slow and time consuming, because does a complete scan of the code and analyzed the relationships between the layers to determine the classes.

Fokaefs et al. [8] introduced a method accompanied by tool-based for identifying source code chunks which collaborate to provide a particular job and propose extraction as detach methods. The proposed work identified the design defects with the Eclipse plug-in which affected coupling and cohesion. Suggestions could have been better and more complete if the clustering algorithm was combined with other methods, like code duplication detection techniques.

Dexun et al. [9] suggested that classes that were not functionally related could generate software maintenance problems, hence the detection and refactoring of such classes was necessary. The basic process is to gather the dependence relationships between classes, calculate the invoking rates and compare them with dynamic threshold. But the thresholds in FRC bad smell detection that are preset thresholds decrease the veracity of detection results.

Bavota et al. [10] presented an experiment aimed at investigating the characteristics of code components increasing their changes of being subject to refactoring operations where was verified whether refactoring activities occur on classes for which indicators might indicate to be needed for refactoring, such as quality metrics or the presence of smells as detected by the tools-suggest. Quality metrics have not demonstrated a clear relationship with refactoring in some cases, where metrics may not be per se indicators of smells.

Kaur & Kaur [11] used Eclipse tool to refactor the bad smells and make an easy source code to understand. The complexity of the project was then calculated and compared with the initial complexity, and the results were checked. This study confirmed the importance of refactoring that makes a code easier to understand and improve the quality and reduce the maintenance cost.

Zafeiris et al. [12] proposed a method for automated refactoring to the template method design pattern of certain design flaws related to concrete method overriding, where an overriding method includes in its body an invocation to the overridden method through the super keyword (super-invocation), then applied the Template Method design pattern for the elimination of appropriate Call Super instances from a code base, which introduced an algorithm for the discovery of refactoring opportunities based on a broad set of preconditions for the refactoring. Consequently, the results of this study cannot be generalized to a project or projects written in another programming language other than java.

Morales et al. [13] presented a novel approach for automatically scheduling refactoring operations for correcting anti-patterns in software systems where conducted a case study with five open-source software systems and compared the performance of RePOR with the performance of two well-known metheuristics (GA and ACO), one conflicting-aware refactoring approach (LIU), and a recent metaheuristic based on sampling (Sway). Results showed that RePOR can correct more anti-patterns than the techniques in just a fraction of the time, and with less effort. But was compared with genetic algorithm which is known computationally expensive i.e. time-consuming, so this poses a threat for results of the approach.

Turkistani and Liu [14] designed a method for dealing with the Large Class problem by classifying the causes of the code smell and applying different design patterns to refactor the code to improve the quality of the software, analyzing the causes of the Large Class code smell and classifying them into corresponding types and proposing a design pattern to address each type to refactor the code.

Mooij et al. [15] presented an exploratory case study that aimed to rejuvenate an industrial embedded software component implementing a nested state machine. Where develop and apply a series of small, automated, case-specific code refactorings that ensure the code uses well known programming idioms, then perform model-based

ISSN No:-2456-2165

rejuvenation focusing on the high-level structure of the code. And therefore gives ample opportunity to be validated early in the form of code reviews and testing, since each refactoring is carried out directly on the existing code. Moreover, aligning the code with the type of model simplifies the extraction, making the process less error-prone.

III. METHOD

A. *The Proposed Extract Class Approach*

The proposed method for extracting large class by simulating the three rules for normalizing classes. A class is in 1ONF when specific behavior required by an attribute that is a collection of similar attributes, and when shared behavior required by more than one instance of the class is encapsulated to be 2ONF, lastly in 3ONF when it encapsulates one set of coherent behavior.

The method boils down to take a class with many responsibilities that is nominated for extracting by the developer or automatically, where the parser to produce an access-set table of attributes, then calculating the Jaccard similarity index to create a similarity matrix for attributes as shown in Fig 1. The structural similarity matrix is created to compute the cohesion of each class, thus achieving the third rule for normalization. In the case of a high cohesion ratio, the class is behaviorally coherent.

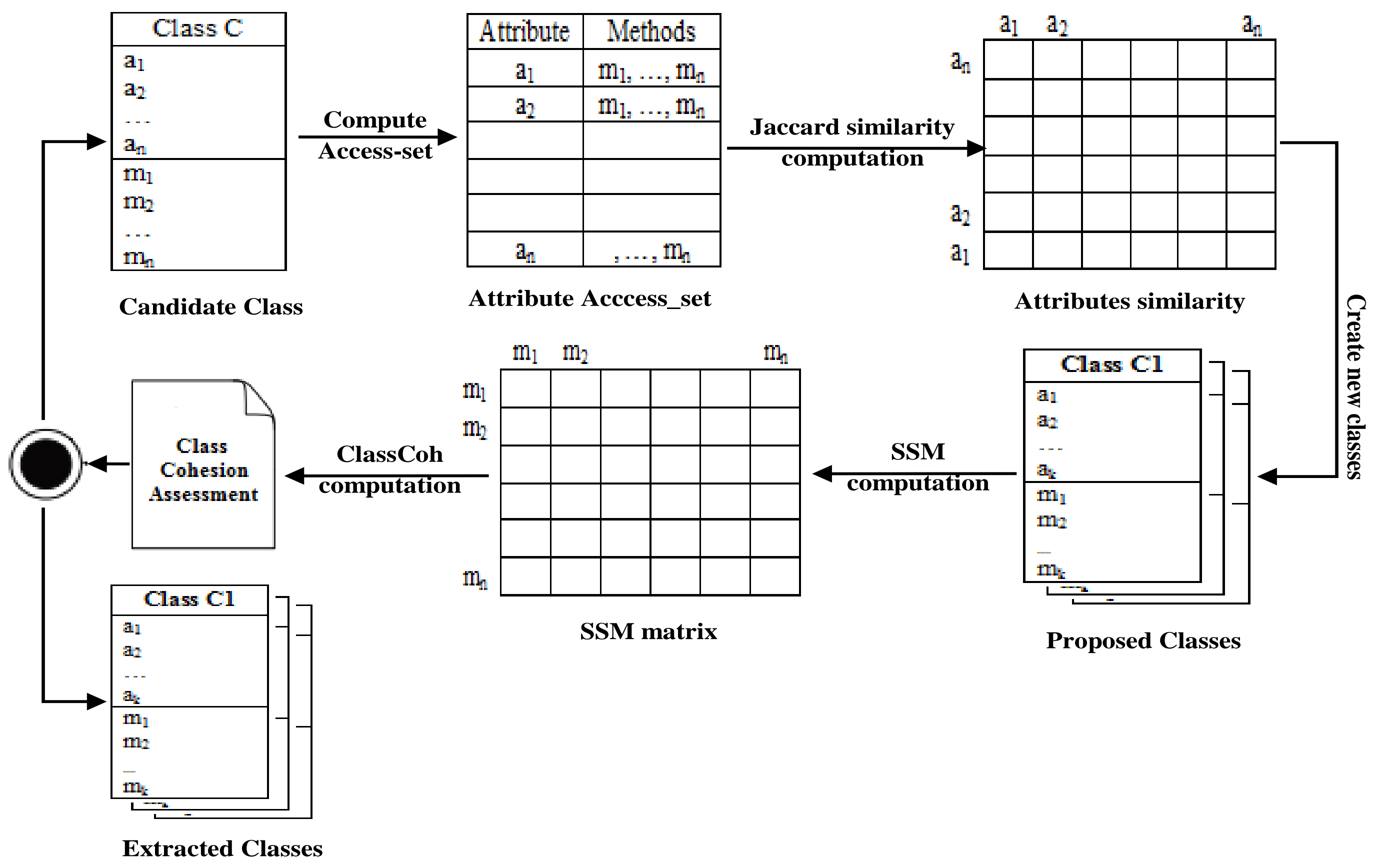

Fig 1:- Process of Extract Class

➢ *Attribute Similarity Matrix*

The attributes similarity matrix is calculated by computing the Jaccard similarity ratio [16] that measures the similarity between two sample sets; it represents a ratio between the sets intersection size and the sets union size. Where access-set is a sample set; hence, computing Jaccard similarity between each access-sets of two attributes until form a similarity matrix for all attributes.

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|} \quad \text{if } |A\cup B|\neq 0 \qquad (1)$$

Similarity matrix has values in [0, 1]; where the value of 1 for a number of attributes indicates that is in the same class with the methods that related to. Consequently, a number of proposed classes are consisted as a result of the original class extraction.

➢ *Structural Similarity between Methods Matrix*

The structural similarity matrix of constituent classes is formed using the structural similarity calculation between methods (SSM) [5]:

$$SSM(mi,mj)=\begin{cases}\frac{|Ii\cap Ij|}{|Ii\cup Ij|} & \text{if } |Ii\cup Ij|\neq 0 \text{ and } i\neq j \\ 0 & \text{otherwise.}\end{cases} \qquad (2)$$

ISSN No:-2456-2165

The SSM of m_i and m_j is calculated as the ratio between the number of reference attributes that are shared by mi and m_j methods and the total number of attributes that are referenced by both methods.

➢ *Compute & Assessment Class Cohesion*

SSM a measure exploited to compute the cohesion metric ClassCoh, by summing the similarities of all method pairs and dividing by the total number of such pairs [17]:

$$\text{ClassCoh}=\frac{\sum_{i,j=1}^{m}\text{SSM(mi,mj)}}{m^2\text{-m}} \quad (3)$$

where *m* is the number of class methods. According to results of calculating this metric, extracted classes is evaluated so that the value between [0.50-1.00] indicates the coherence of classes and less than that mean its incoherent and requires re-extraction.

B. Test of Proposed Approach

Fig. 2 shows part of the UserManagement class and from its name and set of methods, this class was probably originally responsible for implementing a set of operations that would allow the user entity to be manipulated in the database. However, this class has had two new responsibilities added, i.e., the Teaching Entity management and the Role Entity management. The task is to separate this class so that each entity becomes in a separate class and with a specific responsibility by defining single responsibility methods in the class. The question here, do proposed approach able that?. The names of methods in the class have been have been abbreviated, as follows: inserUser (IU), updateUser (UU), deleteUser (DU), existsUser (EU), checkMandatoryFieldsUser (CU), inserTeaching (IT), updateTeaching (UT), deleteTeaching (DT), checkMandatoryFieldsTeaching (CT), inserRole (IR), updateRole (UR), deleteRole (DR) and checkMandatoryFieldsRole (CR).

```
public class UserManagement
    {
        public void insertUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "INSERT INTO tblUser ...";
            ...
        }
        public void updateUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "UPDATE tblUser ...";
            ...
        }
        public void deleteUser(User pUser)
        {
            string sql = "DELETE FROM tblUser ...";
            ...
        }
        public void existsUser(User pUser)
        {
            string sql = "SELECT FROM  tblUser ...";
            ...
        }
        public bool checkMandatoryFieldsUser(User pUser)
        { ... }

        public void insertTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "INSERT INTO tblTeaching ...";
            ...
        }
        public void updateTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "UPDATE tblTeaching ...";
            ...
        }
        public void deleteTeaching(Teaching pTeaching)
        {
            string sql = "DELETE FROM tblTeaching ...";
            ...
        }
        public bool checkMandatoryFieldsTeaching(Teaching pTeaching)
        { ... }

        public void insertRole(Role pRole)
        {
            bool check = checkMandatoryFieldsRole(pRole);
            string sql = "INSERT INTO tblRole ...";
            ...
        }
        public void updateTeaching(Role pRole)
        {
            bool check = checkMandatoryFieldsTeaching(pRole);
            string sql = "UPDATE tblRole ...";
            ...
        }
        public void deleteTeaching(Role pRole)
        {
            string sql = "DELETE FROM tblRole ...";
            ...
        }
        public bool checkMandatoryFieldsRole(Role pRole)
        { ... }
    }
```

Fig 2:- User Management Class

➢ *Step 1.* Create Access-set table for all attribute in the class

Attribute	Access-set
pUser	IU, UU,DU, EU, CU
PTeaching	IT, UT, DT, CT
pRole	IR, UR, DR, CR

Table 1:- Attributes Access-Sets

ISSN No:-2456-2165

- *Step 2.* Calculate the Jaccard Similarity Index of attributes to build similarity matrix

	pUser	pTeaching	pRole
pUser	1	0	0
pTeaching	0	1	0
pRole	0	0	1

Table 2:- Attributes Smilarity Matrix (Jaccard)

- *Step 3.* According to the values in Table 4.2, there are three proposed classes, where each class contains attributes and methods belonging to.

Class C1
+IU(pUser: User): void
+UU(pUser: User): void
+DU(pUser: User): void
+EU(pUser: User): void
+CU(pUser: User): bool

Class C2
+IT(pTeaching: Teaching):void
+UT(pTeaching: Teaching):void
+DT(pTeaching: Teaching):void
+CT(pTeaching: Teaching):bool

Class C3
+IR(pRole: Role):void
+UR(pRole: Role):void
+DR(pRole: Role):void
+CR(pRole: Role):bool

Fig 3:- Proposed Extracted Classes

- *Step 4.* Compute SSM for each proposed class

	IU	UU	DU	EU	CU
IU		1	1	1	1
UU	1		1	1	1
DU	1	1		1	1
EU	1	1	1		1
CU	1	1	1	1	

Table 3:- SSM Similarity of Class C1

	IT	UT	DT	CT
IT		1	1	1
UT	1		1	1
DT	1	1		1
CT	1	1	1	

Table 4:- SSM Similarity of Class C2

	IR	UR	DR	CR
IR		1	1	1
UR	1		1	1
DR	1	1		1
CR	1	1	1	

Table 5:- SSM Similarity of Class C3

- *Step 5.* Compute the cohesion of each class by calculate the ClassCoh metric.

Class cohesion of class C1, $ClassCoh=\frac{20}{25-5}=\frac{20}{20}=1.00$

Class cohesion of class C2, $ClassCoh=\frac{12}{16-4}=\frac{12}{12}=1.00$

Class cohesion of class C3, $ClassCoh=\frac{12}{16-4}=\frac{12}{12}=1.00$

The results indicate each class is completely coherent. The candidate class extracted into 3 classes as the following:

```
public partial class UserManagement
    {
        public void insertUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "INSERT INTO tblUser ...";
            ...
        }
        public void updateUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "UPDATE tblUser ...";
            ...
        }
        public void deleteUser(User pUser)
        {
            string sql = "DELETE FROM tblUser ...";
            ...
        }
        public void existsUser(User pUser)
        {
            string sql = "SELECT FROM tblUser ...";
            ...
        }
        public bool checkMandatoryFieldsUser(User pUser)
        { ... }
    }
```

Fig 4:- Extracted Class C1 (UserManagement)

```
public partial class UserManagement
    {
        public void insertTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "INSERT INTO tblTeaching ...";
            ...
        }
        public void updateTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "UPDATE tblTeaching ...";
            ...
        }
        public void deleteTeaching(Teaching pTeaching)
        {
            string sql = "DELETE FROM tblTeaching ...";
            ...
        }
        public bool checkMandatoryFieldsTeaching(Teaching pTeaching)
        { ... }
    }
```

Fig 5:- Extracted Classe C2 (TeachingManagement)

```
public partial class UserManagement
    {
        public void insertRole(Role pRole)
        {
            bool check = checkMandatoryFieldsRole(pRole);
            string sql = "INSERT INTO tblRole ...";
            ...
        }
        public void updateTeaching(Role pRole)
        {
            bool check = checkMandatoryFieldsTeaching(pRole);
            string sql = "UPDATE tblRole ...";
            ...
        }
        public void deleteTeaching(Role pRole)
        {
            string sql = "DELETE FROM tblRole ...";
            ...
        }
        public bool checkMandatoryFieldsRole(Role pRole)
        { ... }
    }
```

Fig 6:- Extracted Classe C3 (RoleManagement)

IV. RESULTS AND DISCUSSION

Based on the results of proposed approach in the preceding example. In Table I, An analysis of the class elements is shown, showing each class attribute or variable and the methods belonging to. By calculating the similarity of attributes by the Jaccard metric to be shown the results in Table II and by taking the attributes that intersection between them and have a similarity value equal to 1, noticed that the *pUser* variable was the result of similarity with itself only and there is no relationship with other variables as well as the rest of the *pTeaching* and *pRole* attributes, so that each attribute in a class with the methods belong to, Fig. 3 shows the three proposed classes arising from the original class division. To measure the cohesion of the one class, the calculation of the measure of the cohesion of the class is applied, so was needful to calculate the structural similarity between methods metric and the result is appeared in Tables III, IV and V. The results of calculating the classCoh metric showed the extent of cohesion of each class where assumed a threshold value 0.5 to be any value less than this, indicates weak the cohesive of class and needs to be refactored. Fig. 4, 5 and 6 show the extracted classes, and the keyword partial was used to maintain class coupling, in the case of inheritance or a recall, with other classes in the system.

A comparison of what was achieved using the proposed approach with previous literature in obtaining extracted classes with single responsibility and more coherent elements, and with differing the used mechanisms. The approach by Bavota et al. [5] creates a weighted graph for each class under evaluation. Class methods are treated as nodes, and cohesion is assigned as edge-weights between methods. While the presented methodology by Fokaefs et al [18], [8] that computes entity sets for each attribute and method in the target class. All an entity set elements are computed with a distance matrix, and then a threshold value on distance is applied to get the cohesive sets of attributes and methods. However, considering method-calls as a primary means to establish cohesion might not hold good in many cases and hence that may result in inappropriate grouping. Proposed that forming cohesive attribute-set first and then considering method-similarity as a mechanism to establish cohesion.

V. CONCLUSION

This study proposed an approach to extraction large class and improve its cohesion, the approach splits the class to new classes with high cohesion without affecting in the coupling with other classes. The method produces an access-set table of attributes of the class to be needed refactoring, then calculating the Jaccard similarity measure to create a similarity matrix for attributes and by taking by the highest similarity value of intersect attributes new class are created with the methods that related to, then design the structural similarity matrix of each extracted class to calculate the cohesion of each class. Class cohesion metrics, i.e. structural similarity between methods and class cohesion is applied to class normalization rules on source code. The method shows importance of refactoring to enhanced quality of class and simplest the maintenance, where improves the structure of class and makes more organizing, and from the limitation of this increase the size of software to increase the number of classes.

REFERENCES

[1]. I. Sommerville, *Software engineering*, Tenth edition. Boston: Pearson, 2016.

[2]. M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Reading, MA: Addison-Wesley, 1999.

[3]. S. W. Ambler, *Agile database techniques: effective strategies for the agile software developer*. Indianapolis, IN: Wiley, 2003.

[4]. A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, Mar. 2008, doi: 10.1109/TSE.2007.70768.

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

[6]. 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, Dec. 2014, doi: 10.1007/s10664-013-9256-x.

[7]. J. Al Dallal, "Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics," *Inf. Softw. Technol.*, vol. 54, no. 10, pp. 1125–1141, Oct. 2012, doi: 10.1016/j.infsof.2012.04.004.

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

[9]. J. Dexun, M. Peijun, S. Xiaohong, and W. Tiantian, "Functional Over-Related Classes Bad Smell Detection and Refactoring Suggestions," *Int. J. Softw. Eng. Appl.*, vol. 5, no. 2, pp. 29–47, Mar. 2014, doi: 10.5121/ijsea.2014.5203.

[10]. G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–14, Sep. 2015, doi: 10.1016/j.jss.2015.05.024.

[11]. A. Kaur and M. Kaur, "Analysis of Code Refactoring Impact on Software Quality," *MATEC Web Conf.*, vol. 57, p. 02012, 2016, doi: 10.1051/matecconf/20165702012.

[12]. V. E. Zafeiris, S. H. Poulias, N. A. Diamantidis, and E. A. Giakoumakis, "Automated refactoring of super-class method invocations to the Template Method design pattern," *Inf. Softw. Technol.*, vol. 82, pp. 19–35, Feb. 2017, doi: 10.1016/j.infsof.2016.09.008.

[13]. R. Morales, F. Chicano, F. Khomh, and G. Antoniol, "Efficient refactoring scheduling based on partial order reduction," *J. Syst. Softw.*, vol. 145, pp. 25–51, Nov. 2018, doi: 10.1016/j.jss.2018.07.076.

[14]. B. Turkistani and Y. Liu, "Reducing the Large Class Code Smell by Applying Design Patterns," in *2019 IEEE International Conference on Electro Information Technology (EIT)*, Brookings, SD, USA, May 2019, pp. 590–595, doi: 10.1109/EIT.2019.8833851.

[15]. A. J. Mooij, J. Ketema, S. Klusener, and M. Schuts, "Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada, Feb. 2020, pp. 617–621, doi: 10.1109/SANER48275.2020.9054823.

[16]. T. Sharma and P. Murthy, "ESA: the exclusive-similarity algorithm for identifying extract-class refactoring candidates automatically," in *Proceedings of the 7th India Software Engineering Conference on - ISEC '14*, Chennai, India, 2014, pp. 1–6, doi: 10.1145/2590748.2590763.

[17]. G. Gui and P. D. Scott, "New Coupling and Cohesion Metrics for Evaluation of Software Component Reusability," in *2008 The 9th International Conference for Young Computer Scientists*, Hunan, China, Nov. 2008, pp. 1181–1186, doi: 10.1109/ICYCS.2008.270.

[18]. M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, Waikiki, Honolulu, HI, USA, 2011, p. 1037, doi: 10.1145/1985793.1985989.