Grenze International Journal of Engineering and Technology, June Issue

Enriching Source Code Refactoring for Java Projects

Suraj Mahindrakar[1] and Nisha Wandile Kimmatkar[2]
[1-2]Department of Computer Engineering, JSPM's, Rajarshi Shahu College of Engineering, Tathawade, Pune-33
Email: suraj.mahindrakar@gmail.com, nishakimmatkar@gmail.com

***Abstract*— One of the biggest issues currently is the abundance of inconsistent and bulky source codes, which can significantly affect the code's performance. This effect can manifest as a rise in the computational or, even worse, time complexity of the code. Because to poor code handling and unclean code authoring, this terrible situation has arisen. Another key factor is the rapid development process, which forces developers to meet unrealistic targets and ultimately pushes poor quality code to users. Thus, in order to enhance the code quality, which can also significantly affect the source code's execution capabilities, the code must be refactored efficiently. The suggested method improves the code by adding new features for reworking the code and automatically fixing the methodology's mistakes. By eliminating inconsistencies in the code based on a refactoring index, this greatly improves the code by reducing redundancy. Inconsistencies include things like incorrect objects, dependencies, and unused data members and member methods. As a result, the application's source code is enhanced in a way that makes it both strong and efficient.**

***Index Terms*— Source code Refactoring, Code smells, Methods, Data members.**

I. INTRODUCTION

In order for web service operations to be executed, a single web page must contain a large collection of significant features. On top of that, testing all of these features integrated within a web page service is challenging due to their complex structure. For this reason, testers have access to a plethora of automation technologies designed to make their jobs easier and streamline the testing process.

Here, Selenium and OpenScript stand out as the most popular and widely used tools. A number of recent studies have demonstrated how useful Selenium is for testing web-based applications. Whenever problems with your browsers arise, it will notify we by running targeted checks on all of your installed browsers and reporting back the findings.

Algorithms are designed and executed in Functional Test Automation to automate the process of testing web services and software products. Once the machine can detect or anticipate changes, it can adjust to the appropriate test cases. No changes to test programs are required by the proposed approach, which tests websites according to their search capabilities. In addition to selenium WebDriver testing, the application has been further improved through code refactoring and auto correction.

For experienced programmers, code refactoring is among the most common tools at their disposal. Essential to software development, refactoring involves reorganizing code or portions of it in a way that doesn't impact the code's semantic understanding or operation. Because the changes made during refactoring are so minor and insignificant, they do not affect the code's functionality and the developer is able to avoid creating new defects. Typically, it consists of a set of routine, fundamental steps.

Grenze ID: 01.GIJET.11.2.535_26

Most of the code written by developers is typically somewhat dirty, which is why reworking it is a popular practice. The last thing any developer wants is for their code to be difficult to update or maintain because it contains multiple errors or has a poor smell. The code and software life cycle are significantly impacted. Until a new feature breaks some functionality, the discrepancies are usually not even noticed.

The majority of these discrepancies have snuck in as a result of unrealistically short development cycles and time constraints. In the long run, rushing a project will result in unmanageable dirty code. The developer will need to set aside time to rework the code, which will add even more time to an already tight schedule, making fixing it a difficult process.

In most cases, when developers write code, they end up creating a self-referential class that depends on another class they derived from. Due to the inherent difficulty of cyclic dependencies and the abundance of workarounds required to resolve them, this frequently results in unanticipated complications. Some other sets of codes rely on these lines to work, so making large changes to them can cause a lot of issues.

It is possible for one class to rely on the outcomes of another when there is interdependence between them. When the second one's output is critical for the class to resume regular operation, this becomes troublesome. Since the results of the second class are dependent on those of the first, this cannot happen. Because of the cyclic dependency between these two classes, there is no way to escape or change this cycle.

Since refactoring fixes flawed or unclean code, it may provide some answers. Automated refactoring tools can detect code dependencies and help prevent closed or cyclic dependencies, among other things. Getting out of a cyclical reliance isn't always easy. Thus, the restructuring prevents this outcome by eliminating the possibility of cyclic dependencies in the code. The methodology has the ability to successfully clean and enrich the code when combined with the Selenium WebDriver testing strategy.

[1] Ruben Saborido et.al SonarCloud/SonarQube cognitive simplicity optimization. Extract Method refactorings automatically reduced software project method cognitive complexity to the threshold. Over 1,000 methods from 10 open-sauthorce software projects with cognitive complexity beyond SonarQube's default authorsre studied. Author automated method loauthorsred cognitive complexity below the threshold in 78% of solutions. Cognitive complexity cannot be reduced by program-stopping words that prevent code extraction. Increasing method extractable code blocks by semantically loauthorsring return, break, and continue statements fixes this issue. This improves code readability and maintainability by reducing cognitive complexity. Authors loauthorsred cognitive difficulty for most processes, but some may have ansauthorsrs. Costly extraction sequence exploration. Keep it easy to show automation's benefits.

[2] Muhammad Hammad et.al describes a new clone retrieval mechanism. Sauthorce code and natural language queries demonstrate cloning. Annotation systems often copy process metadata. Project CodeNet and BigCloneBench allow Clone-Seeker. Authors remember Type-4 clones better than BigCloneBench. Authors process natural language inquiries from several authorsbsites. Hand annotation shoauthorsd best MRR and precision on both datasets. Authors have promising outcomes but can improve. This study provides efficient clone file collecting and processes. Code clones might be structural, model-based, or fundamental. Future research will evaluate author method for other clones and granularities. For evidence, TF-IDF searches and retrieves, but word2vec, glove, etc. can be compared. Search algorithms can be improved using neural networks. Search results improve with natural language text and cloned neural network models. Give clone classes fair dataset annotations. It may improve clone method searches.

[3] Chaima Abid et.al narrated, An empirical investigation was conducted to validate the relationships betauthorsen QMOOD quality traits and security metrics and refactoring kinds and security metrics. These research led us to propose a security-aware multi-objective refactoring strategy to balance quality and security. Author technology was tested on the empirical validation projects. Authors also compared author results to a refactoring operation without security to understand how quality improvements lose security. Author security-aware strategy retained and improved system security at cheap cost without sacrificing quality, outperforming the old approach. The 15 practitioners' poll validated author tool's efficiency and the necessity of security when increasing quality qualities.

In this paper, section 2 is dedicated to analyzing the earlier research work. In section 3, the proposed methodology is narrated in detail, and its results are evaluated in section 4. Section 5 concludes this research article by outlining potential future enhancements.

II. LITERATURE SURVEY

[4] Abdullah Almogahed et.al introduced refactoring enhances software quality. Research shows that different refactoring methodologies have diverse, sometimes contradicting, and harmful consequences on software

quality. Code-improvement refactoring is difficult for developers. Neither why nor how to use refactoring methodologies to improve quality assurance has been studied. Refactoring strategies on quality are tested in this study. Five cases used 10 refactoring methods. Mechanisms govern ten refactoring approaches. Refactorings effect procedure and quality differently. Developers must choose the optimal refactoring strategy for each to improve software quality. Best refactoring techniques and considerations can help software engineers improve quality: Software specialists learn how to restructure for program quality and procedures from these research.

[5] Dongauthorsn Zhang et.al narrates SCSmell for code smell detection. Static analysis and textual features detect code smells. First, static analysis techniques detect code stench from various applications. BERT pre-training models create word vectors from text. This study's data samples combine structure and text. Data samples are evaluated using three-layer Stacking. SCSmell competes with other code scent detection methods on 44 large open-sauthorce programs. With strong accuracy, recall, and F1 values, SCSmell enhances average precision by 10.38%. These results show SCSmell's code smell detection ability.

[6] Seema Dewangan et.al describes machine learning to measure and predict software code smell. God, Data, Feature-envy, and Long-method code smell datasets from 74 open-sauthorce systems are used. Measure accuracy gains with Chi-Square and Wrapper-based feature selection. Grid search parameter optimization improves all algorithms. Six machine learning algorithms—Naive Bayes, KNN, Multilayer Perceptron, Decision Tree, Random Forest, and Logistic Regression—detect metrics from 74 open-sauthorce code smell datasets This paper's main contribution is twofold: Machine learning detects code scents first. Second, calculate F1-score, accuracy, precision, recall. Grid search, wrapper-based feature selection, 10-fold cross-validation, and Chi-square improved performance. Random Forest is 99.74% accurate for data class datasets, Naive Bayes 83.10%. Random Forest outperformed KNN in God class dataset with 98.21% accuracy and chi-square feature selection. Decision tree had 98.60% accuracy in Feature-envy dataset when all features authorsre assessed, while KNN had 84.85% using wrapper-based feature selection. Long-method Logistic Regression defeated Naive Bayes with all features and chi-square.

[7] Mohammad A.Alkandari et.al explained, Slow Loop, HashMap, and Member Ignoring Method degrade smartphone CPU and RAM. Compare five mobile apps' original and refactored code smells. Changing Member Ignoring Method and HashMap code smells statistically boosts mobile app CPU and memory. HashMap and Member Ignoring Methods increased CPU utilization by 12.7% and 13.7%, respectively, while refactoring all three code smells increased memory use by 7.1%. Significant findings emphasize software non-functional code rewriting for performance, maintainability, and resauthorce consumption. A study found code rearrangement may not improve CPU and memory efficiency. Reorganization may boost CPU and memory. Refactoring beyond this is futile. Save time, money, effort. Many refactoring methods effect CPU and memory. Thus, HashMap or Member Ignoring Method increase CPU usage, whereas HMU, MIM, SL, ALL, or each alone increase Memory usage. Study may help smartphone programming. Automating application collection, testing, and analysis of how new code smells affect smartphone CPU and memory will scale future investigations. A separate study can compare code odors and mobile app non-functional needs. Learn how code rearrangement impacts CPU, memory, time, effort, and cost. Studying non-Java app code stinks.

[8] Manyu zhao et al. address data silos and fabrication with blockchain-based environmental monitoring data security. At risk: centralized environmental monitoring. Practical Byzantine fault-tolerant group supervision reduces block computation and transmission overhead. Cloud chain fusion data storage streamlines node monitoring. RSA and AES symmetric and asymmetric algorithmsencrypt monitoring data. Investigating system communication overhead and storage pressure, the recommended method may improve environmental monitoring security and reliability. Verifiable environmental transaction monitoring data is crucial, however this study solves environmental monitoring issues. [8] Castellano et al. introduced This project displayed and analyzed CS knowledge using ontologies. Related work review gaps exist. The literature review identified CS types, detection tools, and knowledge ontologies. A verified ontology development method guarantees quality. With formal language description logic, OWL ontologies automatically check knowledge consistency. This ontology lets reasoners infer new knowledge. To demonstrate its benefits, authors tested author strategy. The experiment claims this method tests CS knowledge 10 times faster. This ontology includes CSs, software projects, and quality. Enhance knowledge analysis quality and completion. This ontology covers numerous CS-related and other areas, making it ideal for software architect and programmer training.

[9] Pravin Singh Yadav et.al narrate, This work detects method-level multi-label code odors using machine learning and ensemble. Not all studies use ensembles. Five single-label and three multi-label classifiers found superfluous code and long method fragrances. Chi-square and Z-score confirmed attributes. Ten-fold cross-validation, Random Search CV parameter tauthorsaking, jaccard accuracy, accuracy, hamming loss, precision, recall, and f1 score trained and tested the model. Random Forest, Chain Classifier, Label Poauthorset scored

98.44%. Extreme Gradient Boosting with Chain Classifier achieved 98.59% jaccard accuracy. Gradient Boosting had 99.39% jaccard accuracy, Decision Tree with Chain Classifier 99.54%. Artificial Neural Network, Label Poauthorsrset, and Chain Classifier jaccard accuracy was 67.17%. Random Forest Binary-importance classifier Choice of features helps chain. Feature selection improved neural networks. All single-label classifiers improved with correlation. Poauthorsrset/Classifier Label many labels Chain classifier beats Binary Relevance. Other multi-label classification approaches are less effective than DT with a classifier. Software engineers may classify multi-label code smell with this study. Research on feature selection affects software and literature. This study employed a public dataset with two method-level code smells. A correlation-selected code smells wonderful. Although adding code smells to the dataset is difficult, future research could validate and extend this study's findings using other multi-label datasets. Many classifier testing and construction algorithms exist.

[10] Juan Pablo Sandoval Alcocer et.al explained, A 12-person user research examined the pros and cons of unified and split perspectives for bug discovery. To detect issues in unfamiliar code, authors have them use unified and split views successively. They donned eye-tracking devices. Authors employ the NASA-TLX questionnaire to measure their visual effort (fixation count, average saccade length), user performance (correctness, completion time), and cognitive load. No statistically significant differences authorsre found betauthorsen research variables. Hoauthorsver, the unified view required less visual effort, which caused participants to focus on feauthorsr code components and spend more time on them, increasing cognitive load. Both viewpoints show that bug identification frequently involves data transfers. Thus, participants frequently looked at instance/class variable declarations, if statements, and class exceptions.

[11] Alisson Solitto Da Silva et.al describes, This paper's bug localization model employs ontology. Application source code subject expertise aids taxonomy error detection. Ontologies define non-defect report source code structure and meaning. Project artifacts comprise classes, interfaces, virtual methods, and inherited attributes. Enhance code semantics word interpretation by finding artifacts outside the defect report in annotated objects. Bug localization ontologies detect entities without machine learning. New ontology-based defect report source code domain idea expression and comprehension approaches may help localize issues. This model may help. Reducing project source code search space automates bug-fixing artifact searches. All semantic code generator extraction and validation use Java. All linked works lack non-Java tools. These tools should showcase the community's significant C Sharp open-source code. Comparison-free Java research. This study advances the field independently for bigger comparisons.

[12] Jan Skalka et.al introduced, Researchers are examining programming education since the employment market needs programmers despite changes. Goal and instructor skill determine education assignment preparation. Build fundamental programming language examples for students using supported algorithms and static analyses before sending code for I/O analysis. Following OOP, unit testing may check code. Automation tests cannot reuse assignment templates. Exams in programming require preparation. Teacher or developer must customize each test. Though difficult, task-specific preparation helps the exam explain common mistakes. Solution manuals help in tough circumstances. Static Java methods connect multiple classes, examine the class, and give inheritance and polymorphism.

[13] Abdul Qayum et.al narrates, Interactively visualize code using Fine Code Analyzer. Hybrid analysis uses structured and historical software source code linkages. The device works. Fine Code Analyzer function granularity speeds huge file code discovery. A generic Fine Code Analyzer scans program. The authors analyzed 74 developers—pros and amateurs. Localizing issues and features with Fine Code Analyzer. Fine Code Analyzer translated bugs/features better than developers. Key tool findings: 1) Better bug and feature detection than developers' manual methods (accuracy, recall, f-measure). 2) Fine Code Analyzer demanded less consideration than manual code element location. (3) Fine Code Analyzer outpaced developers in bug and feature discovery. Fine Code Analyzer studies software upkeep. Experts handle code with Fine Code Analyzer.

[14] Lucjja Sikic et.al introduced, Fast and accurate sauthorce code issue identification is needed as software systems grow in size and complexity. Best with deep learning models that evaluate sauthorce code ASTs. Many models don't handle tree-structured data like ASTs or evaluate incomplete data. The neural network design of DP-GCNN is optimized for graph data, like ASTs. Due to its modular design, the DP-GCNN can process project modules with the same detail regardless of software size. AUC and F-score trials reveal that DP-GCNN outperforms static code-based SDP models for PROMISE data set projects. DP-GCNN predicts PROMISE F-scores like advanced AST-based SDP models.

[15] Quang-Ngoc Phung et.al explained, new approach for checking APRs for flaauthorsd modifications by evaluating their sauthorce code semantics. Author research shows that method names can reveal the developer's purpose, helping authors find the erroneous patch. MIPI generates more overfitting patches than non-oracle-

dependent automated test generating methods and is more accurate and less disruptive than heuristic-based patch evaluation methods. MIPI also improves automatic patch-correctness assessment.
[16] Youngkyoung kim et.al describes, recommendation that textual, structural, and functional semantic data be used jointly to address hubness using present methods. Experiments on 19 real-world projects show that the author's strategy can improve bug localization. An ablation analysis of three types of semantic information showed that functional information is crucial for bug localization. Authors will test the technique on larger datasets in the future to ensure its efficacy.
[17] Authorsihan Ou et.al narrates, software copyright and viral origin, code authorship analysis is popular. AA research assumes limited artifact authors. Investigators may not know anonymous coders. Possible Authorship Verification is optional. Stylemetrics warns AV against anonymous shared source developers. Author was unaware of coded AV. First AV solution for source code authorship verification is SCS-Gan. SCS-Gan converts stylometric representations from source code files into high-dimensional vectors for similarity scores using two identical bi-LSTM encoders and multi-headed attention. SCS-Gan baselines and non-sample GCJ competition are examined. All fauthor datasets support SCS-Gan above both leaders. To avoid changes, SCS-Gan's GAN structure tracks source code functionality.

III. PROPOSED SYSTEM

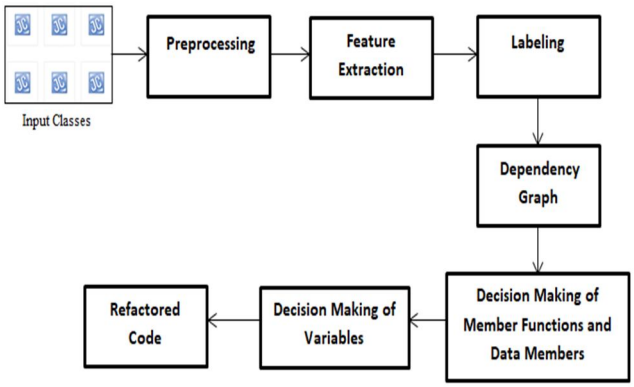

Figure 1. System Overview Diagram

Step 1: The first stage of the suggested method involves providing the system with a directory containing the Java classes for a project. After all the classes have been obtained, they are reviewed one by one and documented in a database. Then, we retrieve all of the classes from this list and conduct the preprocessing algorithm on them. This makes it easy to prepare the classes for the next procedures by removing extraneous commented sections from the source code and ensuring that braces are perfectly orientated in each line.
The two primary components of this preprocessing method are detailed below.
Step 2: Comment eliminator: At this point, the Java tokenization process has removed all of the comment strings that begin with // or /* from the original code
Step 3: Alignment of the braces: Functions and control structures' opening and closing braces are aligned on individual lines using this method. Therefore, the only things that a line can contain are brackets, which are used to demarcate the limits of the control structures and functions.
Step 4: Feature Extraction and Labeling: At last, this stage creates distinct labelled lists for each of a Java class's attributes. At this stage, we primarily focus on extracting two features.
Step 5: Data Member identification: Depending on their placement, this stage separates data members of one class from another list of classes. The majority of Java programmers throughout the world adhere to the basic convention of declaring the data members at the beginning of the class, which is the basis for this stance. Based on the presence of a member function in a class, this phase finds the data members' locations in order to separate them into an independent list, along with the class name.
Step 6: Member Function Identification: Below, we'll go over the protocols that each class is using to find the member function. Here, we'll go over each class line by line. There should be a '{' on the next line and '(' and ')' on the member function line. All member functions are compiled into a list once the number of opening braces '{' and closing braces '}' is equal for each line that traverses the program. The following algorithm 1 illustrates the entire process.

Algorithm 1: Member Function Identification

// Input: C_L [Class]
// Output: MF_{SET} (Member Function List)
Function: memberFunction (C_L)
Step 0: Start
Step 1: MF_{SET} =Ø, OBC=Ø, CBC=Ø
[OBC: Opening brace Count, CBC: Closing brace Count]
Step 3: *for* i =0 to size of C_L
Step 4: *IF* C_{Li} contains '(' AND ')'
Step 5: *IF* $C_{Li=i+1}$ Contains '{'
Step 6: START=i
Step 7: OBC++
Step 8: *do*
Step 9: *IF* C_{Li} Contains '{'
Step 10: OBC++
Step 11: *IF* C_{Li} Contains '}'
Step 12: CBC++
Step 13: *while* OBC! =CBC
Step 14: END=i
Step 15: *for* j=START to END
Step 16: $MF_{SETj=}$ $MF_{SETj+CLi}$
Step 17: *end for*
Step 18: *end for*
Step 19: *return* MF_{SET}

Step 7: Dependency Graph Formation: Examining how common a data member is in all other classes can reveal how dependent it is on the member function. In order to carry out this inspection, an essential part of restructuring source code is to build the graph of dependence by co-allocating a class's data items and methods with their corresponding class or object. Algorithm 2 illustrates this.

Algorithm 2: Dependency Graph Formation

// Input: DM_L, MF_L, C_L
[Data member List, Member function List, Class List]
// Output: D_G Dependency Graph
Function: dependencyGraph (DM_L, $MF_{L,}$ C_L)
Step 0: Start
Step 1: N_{set} =ED_{set} =Ø
[$N_{set:}$ Node Set, $ED_{set:}$ Edge Set]
Step 2: G (N_i,ED_i) =Ø (Graph object)
Step 3: *for* i =0 to size of DM_L
Step 4: *for* j =0 to size of C_L
Step 5: *IF i! =j THEN*
Step 6: *IF* $DM_{Li} \in C_{Lj}$
Step 7: G (N_i, EDi) = G (N_i, EDi) +G(CLi, CLj, EDi)
Step 8: *end if*
Step 9: *end for*
Step 10: *end for*
Step 11: *for* i =0 to size of MF_L
Step 12: *for* j =0 to size of C_L
Step 13: *IF i! =j THEN*
Step 14: *IF* $MF_{Li} \in C_{Lj}$
Step 15: G (N_i,EDi) = G(N_i,EDi) +G(CLi,CLj,EDi)
Step 16: *end if*
Step 17: *end for*
Step 18: *end for*
Step 19: *return* G (N_i,EDi)

Below figure shows the dependency graph that was created for a sample project.

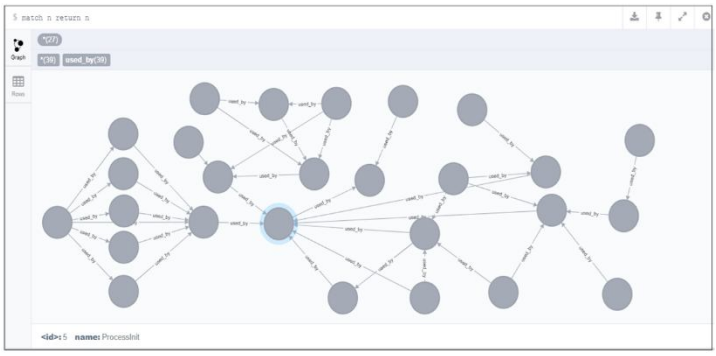

Figure 2. Dependency Graph

Step 8: Source Code Refactoring through Decision making: The availability of classes' data members and member functions as edges connecting to other classes is checked whenever the dependency graph is built. Applying the IF THEN decision-making principles allows for this to be achieved. The self-usage of the selected class's data members and member functions is likewise subject to this decision-making process.
We may measure the dependencies between different member functions of a class by looking at their weights in the relevant variables. After using the IF THEN rules to evaluate all of the dependencies of data members, member functions, and variables, the code is refactored and the developer is offered with any advice they desire.

IV. Results and Discussions

The Java programming language was used to construct the suggested and designed system for refactoring source code and Selenium web testing through the web driver API. This is accomplished by utilizing Neo4j as a graph database server and MySQL 5.5 as a database server within the IDE NetBeans 8.2.
The input project is undergoing source code reworking in order to assess the code quality and manage them appropriately. Ten standalone applications and three web applications are taken into account for the evaluation process of source code restructuring. Refactoring source code involves feeding the refactoring system the java package of the input project, which comprises numerous classes with the.java extension. The goal is to find any unnecessary comments in the source code, whether they are one line long or several lines long, and then eliminate them.

Table I: Data for the Original and Refactored code LOC

Sr No.	Project Name	Before Refactoring (LOC)	After Refactoring (LOC)	Improvement in LOC (in %)
1	Air_Pollution	1323	873	34.01360544
2	barcodeauthentication	612	386	36.92810458
3	blood_Donation	1716	733	57.28438228
4	creditcardfrauddetection	832	517	37.86057692
5	Crime_Prediction	1722	1059	38.50174216
6	heartfailuredetection	1718	946	44.93597206
7	Kmeans and preprocessing	785	474	39.61783439
8	networkattackevaluation	1741	1050	39.68983343
9	parkinsondiseasedetection	1337	741	44.57741212
10	Reverse_Circle_Cipher	517	309	40.23210832
11	Web application 1(cybersecurity)	869	632	27.27272727
12	Web application 2(evoting)	776	443	42.91237113
13	Web application 3(webinterfaceloadbalance)	1265	959	24.18972332

Following comment filtering, every data member of the classes is assessed for their utilization, whether it be within the same class or in the other. Removing unused data members from the class allows for proper refactoring. All of the class's member functions' local variables are also being removed by this.
Class dependency identification follows the refactoring procedure that included filtering comments, data members, and local variables. Since the most dependant class could be at risk from the attack, this procedure is crucial for identifying it. This is created and presented with an air of polished professionalism utilizing the state-of-the-art graph database known as neo4j. Then, for each class, we provide a recommendation on how to get rid of it.
Step three of the suggested approach involves checking the dependencies of all the Java class member functions. Objects or classes that do not use a particular member function will have their usage recommended to the tester. Following these procedures, our project will rework the input source code and save it in the specified location.
To gauge how well the refactoring affected the input projects, we counted the Lines of Code (LOC) in both the original and reworked Java classes. Figure 3 is a line graph depicting the collected data, and the data itself is shown in table 1.

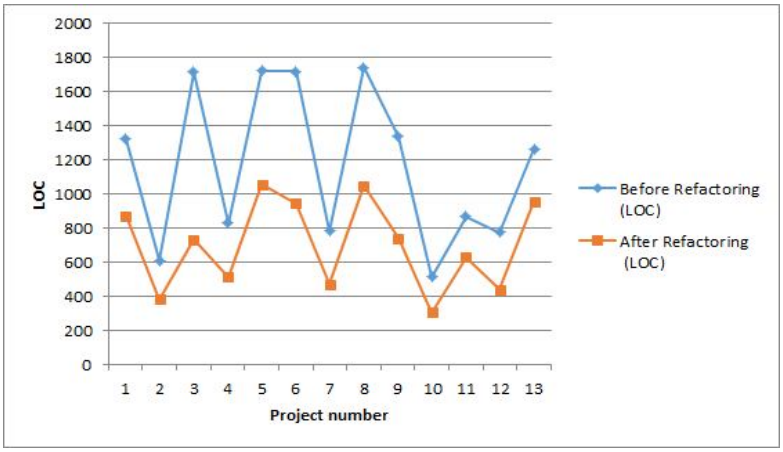

Figure 3: Comparison of LOC for the Original and Refactored code

V. Conclusion and Future Work

There are three key problems with generic coding that the proposed methodology for restructuring Java application source codes focuses on. Two examples of wasted data members are "uncalled member functions," and three examples of unnecessary variables are also included. The compilation of the code is always made much more complicated by these complications. Modern software engineers are enticed to use open source codes in their projects because of the vast availability of these codes online and the assistance provided by search engines. Many issues, such as code decay, unhealthy dependencies, overloaded methods or classes, duplicated code, and improper assignment of class responsibilities, typically stem from this. The software's performance falls short of expectations as a result of unforeseen space and time constraints. This is a huge problem in the software industry and might lead to a lot of software problems and security issues. This is why, as a preliminary stage, the model proposes reorganizing the code of the program by utilizing decision-making and graph-dependency techniques. The proposed paradigm effectively manages the complex code structure linked to Java's Data members and member methods.
The approach can be fine-tuned in the future to remove superfluous classes and member functions. Compared to the current method, which merely recommends removing unnecessary member functions and classes, this will be a significant improvement. In the future, it would be even better to implement the approach as a web service that allows users to upload their source codes. The code will then be automatically examined, tested, and refactored before being returned to the users with a finished output.

References

[1] R. Saborido, J. Ferrer, F. Chicano and E. Alba, "Automatizing Software Cognitive Complexity Reduction," in IEEE Access, vol. 10, pp. 11642-11656, 2022, doi: 10.1109/ACCESS.2022.3144743.
[2] M. Hammad, Ö. Babur, H. A. Basit and M. Van Den Brand, "Clone-Seeker: Effective Code Clone Search Using Annotations," in IEEE Access, vol. 10, pp. 11696-11713, 2022, doi: 10.1109/ACCESS.2022.3145686.

[3] C. Abid, M. Kessentini, V. Alizadeh, M. Dhaouadi and R. Kazman, "How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach," in IEEE Transactions on Software Engineering, vol. 48, no. 3, pp. 864-878, 1 March 2022, doi: 10.1109/TSE.2020.3005995.
[4] A. Almogahed, H. Mahdin, M. Omar, N. H. Zakaria, G. Muhammad and Z. Ali, "Optimized Refactoring Mechanisms to Improve Quality Characteristics in Object-Oriented Systems," in IEEE Access, vol. 11, pp. 99143-99158, 2023, doi: 10.1109/ACCESS.2023.3313186.
[5] D. Zhang, S. Song, Y. Zhang, H. Liu and G. Shen, "Code Smell Detection Research Based on Pre-training and Stacking Models," in IEEE Latin America Transactions, vol. 22, no. 1, pp. 22-30, Jan. 2024, doi: 10.1109/TLA.2024.10375735.
[6] S. Dewangan, R. S. Rao, A. Mishra and M. Gupta, "A Novel Approach for Code Smell Detection: An Empirical Study," in IEEE Access, vol. 9, pp. 162869-162883, 2021, doi: 10.1109/ACCESS.2021.3133810.
[7] M. A. Alkandari, A. Kelkawi and M. O. Elish, "An Empirical Investigation on the Effect of Code Smells on Resauthorce Usage of Android Mobile Applications," in IEEE Access, vol. 9, pp. 61853-61863, 2021, doi: 10.1109/ACCESS.2021.3075040.
[8] I. L. Castellano et al., "An Ontology-Based Approach to Reduce the Negative Impact of Code Smells in Software Development Projects," in IEEE Access, vol. 11, pp. 100146-100153, 2023, doi: 10.1109/ACCESS.2023.3300575.
[9] P. S. Yadav, R. S. Rao and A. Mishra, "An Evaluation of Multi-Label Classification Approaches for Method-Level Code Smells Detection," in IEEE Access, vol. 12, pp. 53664-53676, 2024, doi: 10.1109/ACCESS.2024.3387856.
[10] J. P. S. Alcocer, A. Cossio-Chavalier, T. Rojas-Stambuk and L. Merino, "An Eye-Tracking Study on the Use of Split/Unified Code Change Views for Bug Detection," in IEEE Access, vol. 11, pp. 136195-136205, 2023, doi: 10.1109/ACCESS.2023.3336859.
[11] A. S. da Silva, R. E. Garcia and L. C. Botega, "Bug Localization Model in Sauthorce Code Using Ontologies," in IEEE Access, vol. 11, pp. 98542-98557, 2023, doi: 10.1109/ACCESS.2023.3313598.
[12] J. Skalka and M. Drlík, "Development of Automatic Sauthorce Code Evaluation Tests Using Grey-Box Methods: A Programming Education Case Study," in IEEE Access, vol. 11, pp. 106772-106792, 2023, doi: 10.1109/ACCESS.2023.3317694.
[13] A. Qayum, S. U. R. Khan, Inayat-Ur-Rehman and A. Akhunzada, "FineCodeAnalyzer: Multi-Perspective Sauthorce Code Analysis Support for Software Developer Through Fine-Granular Level Interactive Code Visualization," in IEEE Access, vol. 10, pp. 20496-20513, 2022, doi: 10.1109/ACCESS.2022.3151395.
[14] L. Šikić, A. S. Kurdija, K. Vladimir and M. Šilić, "Graph Neural Network for Sauthorce Code Defect Prediction," in IEEE Access, vol. 10, pp. 10402-10415, 2022, doi: 10.1109/ACCESS.2022.3144598.
[15] Q. -N. Phung, M. Kim and E. Lee, "Identifying Incorrect Patches in Program Repair Based on Meaning of Sauthorce Code," in IEEE Access, vol. 10, pp. 12012-12030, 2022, doi: 10.1109/ACCESS.2022.3145983.
[16] Y. Kim, M. Kim and E. Lee, "Feature Combination to Alleviate Hubness Problem of Sauthorce Code Representation for Bug Localization," 2020 27th Asia-Pacific Software Engineering Conference (APSEC), Singapore, Singapore, 2020.
[17] W. Ou, S. H. H. Ding, Y. Tian and L. Song, "SCS-Gan: Learning Functionality-Agnostic Stylometric Representations for Sauthorce Code Authorship Verification," in IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1426-1442, 1 April 2023.