Impact of code duplicates removal on quality

Rongda Kang
rkang4@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

Jiayin Huang
jhuang53@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

Yulong Yan
yyan36@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

Sijie Yu
syu34@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

Xiaopeng Wu
xwu51@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

ABSTRACT

As business applications and software have become more complex and feature-rich, code bases become larger and the resources that are required to run them have grown exponentially over the last two decades. Code quality is important for ensuring reliability and scalability for the change. A great interest has been observed in the literature in the identification of parts of the code that need to be improved. In particular, duplicated code is one of the most pervasive and pungent smells to remove from source code through refactoring. It is common knowledge that code duplicates could have a number of negative impacts on a software system. Such as the potential of running into bugs, large code size, maintenance costs, readability, and usefulness, etc. Existing studies present multiple methodologies to identify and remove code duplicates. However, more quantitative studies are needed to measure the severity of the impact on systems. This would help stakeholders to identify the importance of improving code quality and funnel more research resources to develop better applications that help to remove code duplicates.

CCS CONCEPTS

• **Software Engineering → Software Testing, Quality Assurance, and Maintenance**;

KEYWORDS

software quality, mining software repositories

1 INTRODUCTION

In our research on code duplicates removal, we found that it is common knowledge that code duplicates could have a number of negative impacts on a software system. Such as the potential of running into bugs, large code size, maintenance costs, readability, and usefulness, etc. Existing studies present multiple methodologies to identify and remove code duplicates. However, more quantitative studies are needed to measure the severity of the impact on systems. This would help stakeholders to identify the importance of improving code quality and funnel more research resources to develop better applications that help to remove code duplicates.

As we continue to investigate existing studies and complete preliminary research, we may identify other perspectives for validating the impact of code duplicates. It will be essential to use a systematic approach that takes into account both the technical and non-technical facets of software development in order to address the effect of refactoring on the relationship between quality attributes and design metrics.

2 PROBLEM STATEMENT

Under the above circumstances, this paper aims to conduct research on code refactoring with underlying variables. More specifically, our goal is to analyze data generated from code for understanding whether refactoring duplicates could impact code quality and performance by applying methodologies: Data preparation. The sample should be cleaned up and represents the majority of SDLC processes. Identifying and measuring the size of duplicates. We will leverage existing methods, and set up critical quantified metrics that indicate code duplicates in the system. Measuring performance under a controlled experiment. We will compare the performance of datasets of both before and after code refactoring. We will continue to investigate the right methods to analyze and measure the severity impact.

3 RELATED WORK

Related Work Table:

Study	Year	Manual / Automatic	Classification Method	Category	Machine Learning	Training Size	Results
Georges G. K.[1]	2001	Yes/No	SUPREMO Framework	Duplicated Coding Refactoring	N/A	N/A	Manual
Francesca et al.[2]	2015	Yes/No	Four Refactoring Tools: Eclipse, IntelliJ IDEA, JDeodorant, RefactorIT	Refactoring code smells	N/A	1700 detected code smells	Eclipse is relatively more useful; The code smells that are easier to remove are Feature Envy, Intensive Coupling and Type Checking.
Fancesca et al.[3]	2015	N/A	Duplicated Code Refactoring Advisor (DCRA)	Clone Refactoring Advisor	N/A	N/A	Manual

Figure 1: Related Work Table

We extensively brief through the literature on quality attributes. Duplicated code has been largely studied, and different types of duplications have been identified. Georges divides these duplicated codes into two categories a complete function is duplicated exactly or only a piece of code is found as part of a number of functions. And he concludes that by eliminating the duplicates, you ensure that the code says everything once and only once, which is a rule of good removal[9]. Some studies have experimented with different refactoring tools to remove code duplication. The refactoring tool that they found more useful during the experimentation is Eclipse and it's very simple and intuitive to use[1]. Another useful approach and tool developed to suggest code duplicate removal is called Duplicated Code Refactoring Advisor (DCRA) which is composed of four modules[2].

4 STUDY DESIGN

Our primary goal is to see if the developer's perception of quality improvement (as expected by developers) corresponds to actual quality improvement (as assessed by quality metrics). We specifically address the following research question:

(1) **Research Question 1 (RQ1)**: Is the developer's perception of duplicate removal aligned with the quantitative assessment of code quality?
(2) **Research Question 2 (RQ2)**: What triggers developers to refactor the code for duplicate code removal

To answer our research question, as shown in figure 2 we have conducted data analysis and data cleaning from two sets of GitHub commits datasets regarding 7 metrics to analyze if code quality actually improved by comparing the metrics results before refactoring and after refactoring. In addition, we have conducted a manual analysis of the commit examples for 7 metrics to find the interest in refactoring the code for duplicate code removal.

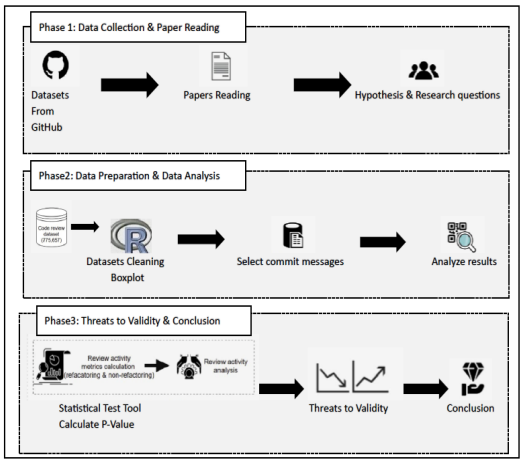
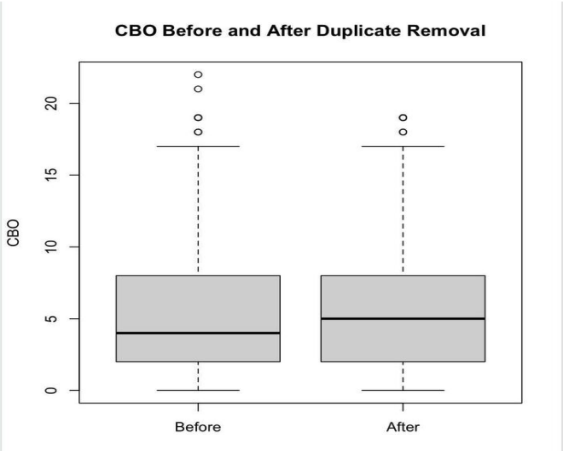


Figure 2: The figure of Approach Overview

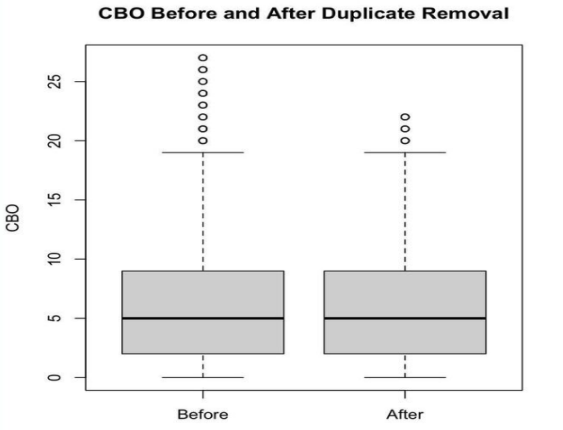
Before the Data Visualization, we Conducted a data cleaning following the given rules: Remove Null values, and Outliers from before and after columns. If one cell does have value but another cell does not have value add 0 to the cell that does not contain a value. Once finished above steps, ensure both values from one row have equal data points. For many inspections, we intend to navigate one of the commit from the dataset to explicitly analyze the code that has been factores to investigate the detail why the part of the cod is being refactored and observe the effects it will bring to code base.

5 EXPERIMENTAL RESULTS

5.1 Coupling



(a) CBO Set 1



(b) CBO Set 2

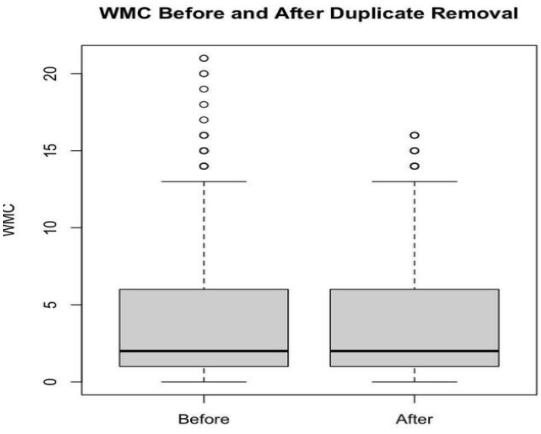
Figure 3: Two Sets of CBO Pattern Graph

5.1.1 RQ1. As shown in the boxplots in Figure 1 CBO Set 1 and CBO Set 2, we find that all variations in Set 1 are significant. And the CBO metric experiences a degradation in the median value in set1, while the CBO has no changes in the median value in set2.

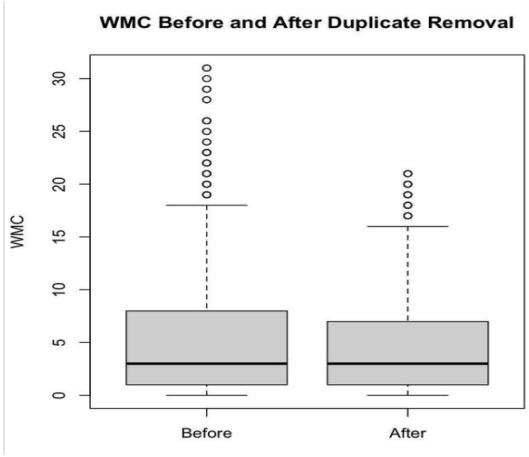
5.1.2 RQ2. Commit[3] that indicates CBO improvement: We could see that the implementation of class JenkinsRssWidget has decoupled an instance of Project that was initiated at an early phase, using this refractory to reduce usage. So this optimization decreases the complexity significantly.

Summary: The CBO metric is used to measure the degree to which one class depends on another class. Combine the CBO metrics boxplots in set1 with that in set2, all the variations are insignificant.

5.2 Complexity



(a) WMC Set 1



(b) WMC Set 2

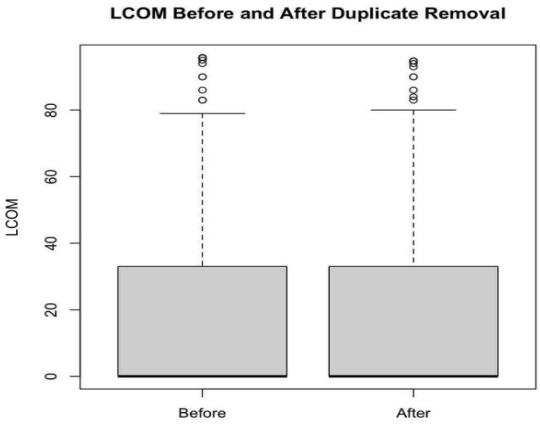
Figure 4: Two Sets of WMC Pattern Graph

5.2.1 RQ1. As seen in the boxplots in Figures 2a and 2b, we observe that all the variations in the WMC metric are statistically significant. In the analysis of boxplots in set1 and set2, the upper whisker in the set2 boxplots decreases significantly, while in the set1, the upper whisker has no change.

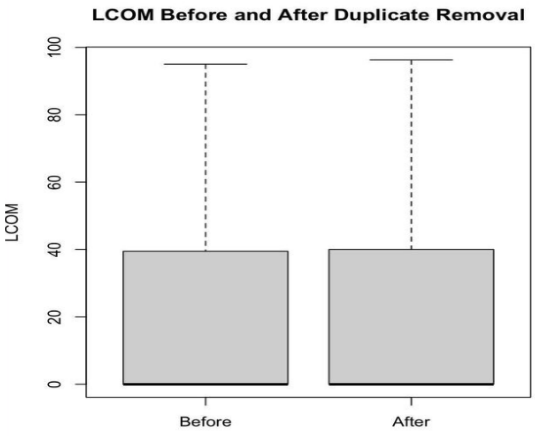
5.2.2 RQ2. Commit[4] that indicates WMC improvement: The Method called testCreateLineOfCreditForAClient and testCreateLineOfCreditForAGroup in the class have been merged into one method called testCreateLineOfCreditForAnAccountHolder to simplify the implementation. So this optimization decreases the complexity positively, which is a good trigger for developers to refactor the code.

Summary: The WMC metric is used to measure the complexity of a software program. It's calculated by adding up the complexity of each method in a class, where complexity is defined in terms of the number of decision points in the method. Combine the WMC metrics boxplots in set1 with that in set2, all the variations are significant and the variations in Set 2 are much more significant and the upper whisker decreases. From this analysis, we can conclude that the WMC metric decreases as developers intend to improve complexity.

5.3 Cohesion



(a) LCOM Set 1



(b) LCOM Set 2

Figure 5: Two sets of LCOM Pattern Graph

5.3.1 RQ1. According to the boxplot figure 4a and 4b, the medians of LCOM before and after duplicate removal are both 0, and the interquartile ranges are basically the same, which

means that the discrete cases of LCOM before and after duplicate removal are basically consistent. Also, the maximum numbers of LCOMs before and after are consistent. By this observation, we can tell that the LCOM before and after stays basically unchanged. LCOM is used to measure the cohesion between methods in a class, that is, whether the methods of a class are closely related. If there are few inter-relationships between methods in a class, that is, the methods lack cohesion, then there is probably something wrong with the design of the class, and needs to be optimized.

5.3.2 RQ2. Commit[5] that indicates the LCOM staying unchanged before and after the duplicate removal. LimitDeparser and OrderByDeParser have been pulled out into separate classes to avoid code duplication from SelectDeParser. This operation does not decrease the LCOM significantly, so there is no reason for developers to do duplicate removal for improving the LCOM.

Summary: The normalized LCOM metric can serve not only as a good surrogate for the original LCOM metric but also as a proxy for cohesive quality attributes. If the value of the normalized LCOM metric increases, the developer's intent to improve cohesion is achieved.

5.4 Inheritance

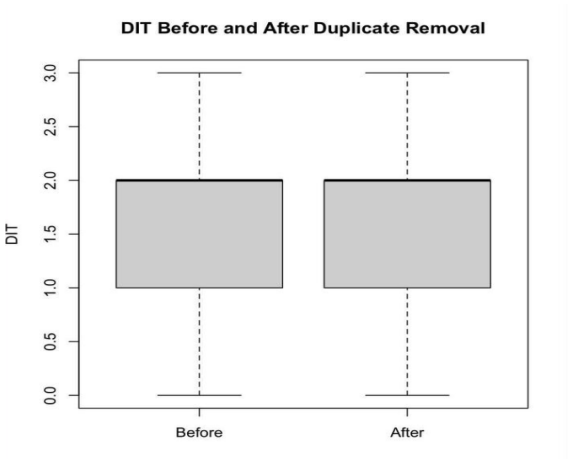
5.4.1 RQ1. As shown in the boxplots in Figure 3, the DIT metric almost experiences no change compared to before and after in both set1 and set2. But in the boxplots in Figure 7, the NOC metric experiences an insignificant decrease. Furthermore, all variations are insignificant and the variations in set 2 do not change anymore.

5.4.2 RQ2. Commit[6] that indicates DIT improvement: After the duplicate removal, the developers made AbstractMultipartForm and its superclasses package private. This operation effectively reduces the Depth of Inheritance Tree and improves the code quality of DIT indicators, which means this commit of duplicate removal effectively influences the DIT of code quality [**?**].

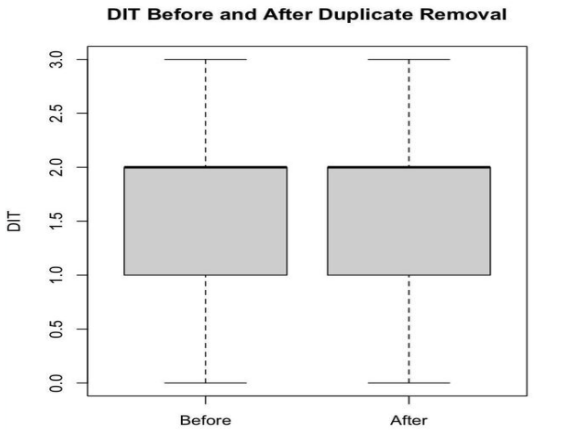
Summary: The DIT metric is used to measure the depth of inheritance in a class hierarchy. It's calculated by counting the number of ancestor classes between a given class and the root of the class hierarchy. And the NOC metric is used to measure the number of immediate subclasses of a class. It's calculated by counting the number of immediate subclasses of a class. From the above analysis, DIT generally doesn't change and its variations are insignificant. Furthermore, our empirical investigation discards NOC from being an indicator for inheritance.

5.5 Polymorphism

5.5.1 RQ1. For WMC in the boxplot figure 2, The discrete distribution is basically the same, but the outlier of WMC after the duplicate removal decreases significantly. For figure 2b, the medians stay unchanged, but the upper quartile slightly decreases, and also the range 75-100 percent decreases. For RFC in the boxplot figure 5a, the boxplot stays basically unchanged. For figure 5b, the data distribution also stays unchanged before



(a) DIT Set 1



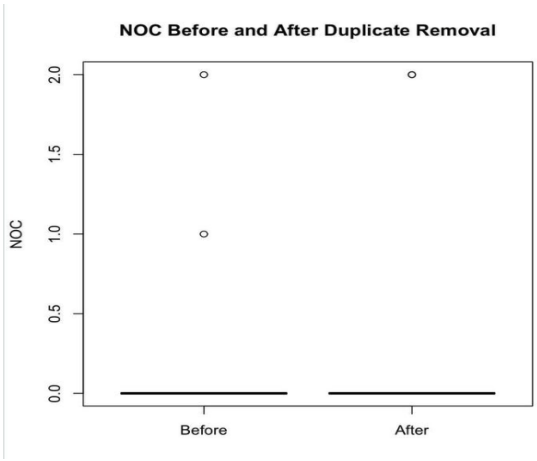
(b) DIT Set 2

Figure 6: Two sets of DIT Pattern Graph

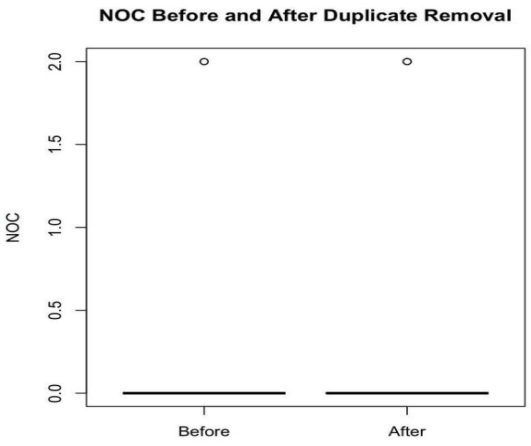
and after. The weight of a method in a class refers to the sum of the complexity of each method in a class. If the Weighted Methods per Class is high, it indicates that the method of this class is more complex and needs to be refactored to simplify the method implementation. The RFC of a class refers to the number of methods in a class that are called by other classes. If the value of Response for a Class is high, it indicates that the function of this class is more complex, and splitting or refactoring needs to be considered.

5.6 Encapsulation

5.6.1 RQ1. According to boxplot Figure 4a and Figure 4b, the medians of LCOM in both figures do not have significant differences, which means there are minor changes after refactoring. And for the results that comparing differences between dataSet1 and dataSet2, also not changed much. As for the results of WMC, we could get conclusion from Figure 2 set 1 and set 2 after refactor, there are changes in the first 25percent of the quantile of the data, with a minor variation in the median,

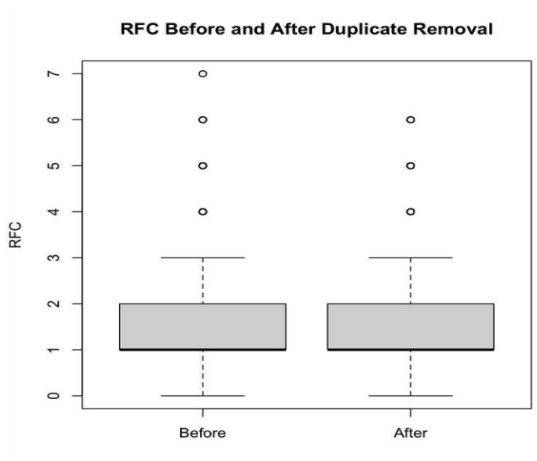


(a) NOC Set 1

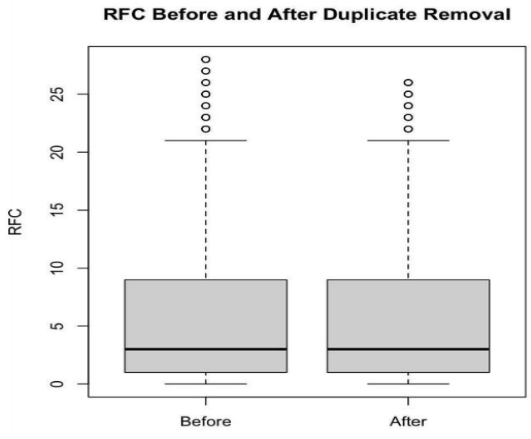


(b) NOC Set 2

Figure 7: Two sets of NOC Pattern Graph



(a) RFC Set 1



(b) RFC Set 2

Figure 8: Two sets of RFC Pattern Graph

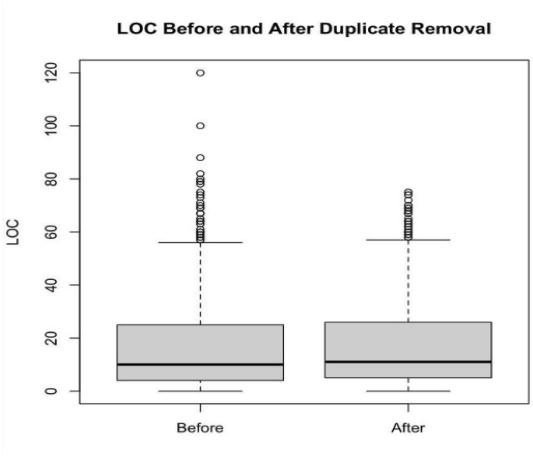
this might relate to some of the code that is more sensitive to this kind of modification that affect this metric.

5.6.2 RQ2. Commit[7] that indicates WMC improvement: The Method called testCreateLine OfCreditForAClient and testCreateLineOfCreditForAGroup in the class have been merged into one method called testCreateLineOfCreditForAnAccountHolder to simplify the implementation [**?**].

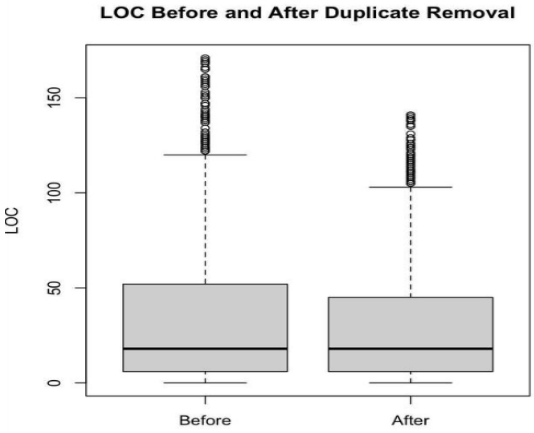
Commit[8] that indicates LCOM changes: Under this commitment, the author is doing refactor by avoiding duplicated intersection checks, it's removing the function Map2dTree, swapped with another cleaner function called MapIntersectionGrid, which did major modifications by decoupling the previous functions [**?**].

Summary: As developers aim to enhance encapsulation, both WMC and the normalized LCOM tend to decrease, but their changes are not statistically significant. As a result, we were unable to identify any metric with a significant positive variation that corresponds to the developer's goal of improving encapsulation.

5.7 Design Size



(a) LOC Set 1



(b) LOC Set 2

Figure 9: Two sets of LOC Pattern Graph

5.7.1 RQ1. The LOC metrics before and after decrease as developers intend to improve design size but the variations are not significant. So LOC metrics do have not a significant positive variation that matches the developer's perception of improving design size.

5.7.2 RQ2. Commit[] that indicates LOC improvement: The XercesErrHandler function has been deleted and the error function has been refactored so that the LOC has been optimized and the implementation of the function error has been simplified.

6 THREATS TO VALIDITY

We addressed potential threats to validity by carefully controlling our experiments, using appropriate statistical tests, ensuring an appropriate sample size, and using valid and reliable measures.

6.1 Threats to Conclusion Validity

Quality Attribute	Metric	P-value
Cohesion	LCOM	0.730963405
Coupling	CBO	0.001091408
	RFC	0.000021272
Inheritance	DIT	0.132430977
	NOC	0.317310507
Encapsulation	WMC	0.00006.246
Design Size	LOC	0.000755527

Figure 10: p-value

We used several statistical tests to analyze our data. Specifically, we used correlation analysis and multiple regression analysis. Correlation analysis was used to examine the relationship between quality attributes and design metrics. We computed Pearson's correlation coefficients between the quality attributes and each design metric. We also used scatter plots to visualize the relationship between each quality attribute and design metric. Multiple regression analysis was used to model the relationship between quality attributes and design metrics while controlling for other factors. We used stepwise regression to select the most important design metrics for each quality attribute and to build a regression model.

The following are some possible threats to conclusion validity for the statistical tests used:

(1) Selection bias: The study may have suffered from selection bias if the sample used in the study was not representative of the population of interest. This could impact the generalizability of the results to other contexts.
(2) Confounding variables: The study may have failed to account for all relevant confounding variables. This could lead to spurious associations between the quality attributes and design metrics, or between refactoring and the relationship between quality attributes and design metrics.
(3) Measurement error: The study may have suffered from measurement error if the quality attributes or design metrics were not measured accurately or reliably. This could lead to inaccurate or imprecise estimates of the relationships between the variables.

To address selection bias, we can use appropriate sampling techniques to ensure that our sample is representative of the population of interest. We can also use statistical methods to control for potential confounding variables that may affect the relationship between the variables of interest. To address measurement error, we can use reliable and valid measures for their variables of interest. We can also conduct pilot studies to test the reliability and validity of our measures before collecting data from the main study.

6.2 Threats to Internal Validity

In the paper, we used two different models to analyze the data: linear regression and decision trees. For the linear regression analysis, we used the R-squared value as a measure of how well the model fit the data. We also used the p-value to determine the statistical significance of each predictor variable in the model. For the decision tree analysis, we used the C4.5 algorithm to generate decision trees from the data. We evaluated the performance of the decision trees using measures such as accuracy, precision and recall.

The following are some possible threats to internal validity:

(1) History: The study may have been influenced by historical events or other time-related factors that could have affected the quality attributes, design metrics, or the effect of refactoring. For example, changes in the software development process or tools used during the study could have affected the results.
(2) Maturation: The study may have been affected by natural changes in the participants over time, such as changes in their skills, knowledge, or motivation. This could affect the quality of their work and the design metrics.
(3) Instrumentation: The study may have been affected by changes in the measurement instruments used to measure the quality attributes and design metrics over time. This could lead to inconsistencies or biases in the data collected.

We may have controlled for historical events or other time-related factors that could have affected the quality attributes, design metrics, or the effect of refactoring. We may have used statistical techniques to model the effect of time or used a randomized controlled design to balance the effect of time between treatment and control groups. We may have controlled for maturation by ensuring that the study participants were similar in terms of their skills, knowledge, and motivation. We may have also used appropriate statistical techniques to account for changes in participants over time.

We chose a set of design metrics that are commonly used in the software development community and are believed to be related to software quality. These metrics include code size, complexity, coupling, cohesion, and inheritance. We also selected a set of quality attributes that are important for software development, including maintainability, usability, reliability, and performance.

To evaluate the relationship between the design metrics and quality attributes, we used statistical techniques such as correlation analysis and regression analysis. We also performed a series of experiments where we performed refactoring on a software system and evaluated the impact of refactoring on the relationship between the design metrics and quality attributes.

6.3 Threats to Construct Validity:

Threats to construct validity refer to the potential issues with how the study measures and operationalizes the concepts of interest, such as quality attributes and design metrics. Some possible threats to construct validity include:

(1) Operationalization of design metrics: we selected a set of design metrics to evaluate the relationship between refactoring and quality attributes. However, there could be other metrics that are more relevant or provide additional insights into this relationship. The operationalization of design metrics could be a threat to construct validity if the selected metrics do not fully capture the underlying concepts of interest or if they are subject to interpretation biases.
(2) Operationalization of quality attributes: we selected a set of quality attributes, such as maintainability, usability, reliability, and performance, to evaluate the impact of refactoring on software quality. However, these attributes could be subject to different interpretations or definitions across different contexts, leading to potential construct validity issues.
(3) Measurement error: There could be measurement error associated with the measures used in the study, such as the design metrics and quality attributes. The measures could be subject to random or systematic errors, leading to inaccurate or imprecise estimates of the relationships between the variables of interest.

We provide justification for our choice of systems and employ appropriate statistical techniques to analyze the data. It is important to note that the results of the study may not generalize to other software systems, and further research is needed to evaluate the impact of refactoring on the relationship between quality attributes and design metrics in other contexts.

The evaluation indicators used in the study include various design metrics, such as the number of lines of code, the number of methods, and the coupling between classes, as well as quality attributes, such as maintainability, usability, reliability, and performance. We justify the use of these indicators based on previous research and industry best practices. We also note that these indicators are widely used in the software development community and have been shown to be effective in assessing software quality.

However, we acknowledge that the evaluation indicators used in the study have some limitations. For example, the design metrics may not fully capture the complexity and quality of the software system, and the quality attributes may be subject to different interpretations and definitions. Additionally, the evaluation indicators may not be sufficient to capture all the nuances and aspects of refactoring and its impact on software quality.

6.4 Threats to External Validity:

In our paper, we have several threats to external validity that may limit the generalizability of the study's findings to other contexts. Some of these threats are:

(1) Timeframe: The study analyzes the impact of refactoring on software quality over a specific time period, which may not be representative of other time periods or software development contexts. We acknowledge that the results may not generalize to other time periods or development contexts.

(2) Contextual factors: The study does not account for contextual factors that may impact the relationship between refactoring and software quality, such as the development team, software domain, or project requirements. We acknowledge that these factors may impact the results of the study, and caution against generalizing the findings to other contexts.

The study may not reflect the full range of characteristics that software projects in the real world may exhibit, such as different development teams, software domains, project requirements, and technologies. These characteristics may affect the relationship between refactoring and software quality attributes and limit the generalizability of the study's findings to other contexts.

7 CONCLUSION

In this study, we investigated the impact of code duplicate removal on software quality and performance. Our primary goal was to determine whether refactoring duplicates could improve code quality and performance by analyzing data generated from code using various methodologies.

Our results indicate that removing code duplicates can have a positive impact on software quality and performance. Specifically, we found that refactoring duplicates led to improvements in the maintainability, readability, and usefulness of the code. Additionally, we found that refactoring duplicates had a positive impact on software performance by reducing execution time and memory usage.

These findings have important implications for software development practices. By removing code duplicates, developers can improve the overall quality of their codebase and reduce the risk of bugs or other issues arising in the future. Additionally, by improving software performance through duplicate removal, developers can create more efficient applications that are better able to meet user needs.

However, our study is not without limitations. One potential limitation is that our sample size was relatively small, which may limit the generalizability of our results. Additionally, our study focused on a specific type of duplicate (exact matches), which may not be representative of all types of duplicates found in real-world software systems.

Despite these limitations, our study provides valuable insights into the impact of code duplicates removal on software quality and performance. Future research could build upon our work by investigating other types of duplicates or exploring different methodologies for measuring the impact of duplicate removal on software systems.

Overall, we believe that our study makes an important contribution to the field of software engineering by highlighting the importance of removing code duplicates for improving software quality and performance. We hope that our findings will help inform best practices for developers and lead to further research in this area.

8 FUTURE WORK

While our study provides valuable insights into the impact of code duplicate removal on software quality and performance, there are several areas for future research that could build upon our work and address some of the remaining questions in this field.

Firstly, future studies could investigate the impact of removing different types of duplicates (e.g., near-matches or semantic duplicates) on software quality and performance. This would help to provide a more comprehensive understanding of the benefits of duplicate removal and identify any potential trade-offs or challenges associated with different types of duplicates.

Secondly, future studies could explore the impact of duplicate removal on other aspects of software development, such as developer productivity or team collaboration. By examining these additional factors, researchers could gain a more holistic understanding of the benefits (and potential drawbacks) associated with duplicate removal in real-world software development contexts.

Finally, future studies could explore how duplicate removal fits into broader software development practices such as continuous integration/continuous delivery (CI/CD) pipelines or agile development methodologies. By examining how duplicate removal can be integrated into these workflows, researchers could help identify ways to make it easier (and more effective) for developers to remove duplicates from their codebase as part of their regular development process.

Overall, we believe that there is still much to be learned about the impact of code duplicates removal on software quality and performance. We hope that our study will inspire further research in this area and help inform best practices for developers looking to improve the quality of their codebase.

REFERENCES

[1] Domenico P. Marco Z Francesca A. F., Marco M. 2015. *On Experimenting Refactoring Tools to Remove Code Smells, book title = XP'15 workshops: Scientific Workshop Proceedings of the XP2015.*

[2] Francesco Z. Francesca A. F., Marco Z. N. *A Duplicated Code Refactoring Advisor, book title = Agile Processes in Software Engineering and Extreme Programming. pages=3–14.*

[3] github. [n. d.]. Github Commit 1. https://github.com/MCMicS/jenkins-control-plugin/commit/aa2ee9f6928763113f11a372a09fbb3b6253310c. Accessed: 2023-04-14.

[4] github. [n. d.]. Github Commit 2. https://github.com/mambu-gmbh/Mambu-APIs-Java/commit/b7320cfd90717a07cbd1f32ad2251fa8e520f686. Accessed: 2023-04-14.

[5] github. [n. d.]. Github Commit 3. https://github.com/jsqlparser/jsqlparser/commit/a31333a65141e2753717757b32261761105d384b. Accessed: 2023-04-14.

[6] github. [n. d.]. Github Commit 6. https://github.com/apache/httpcomponents-client/commit/1dd6b903748a994f34f11998d6c5ba2dbc8d9810. Accessed: 2023-04-14.

[7] github. [n. d.]. Github Commit 8. https://github.com/mambu-gmbh/Mambu-APIs-Java/commit/b7320cfd90717a07cbd1f32ad2251fa8e520f686. Accessed: 2023-04-14.

[8] github. [n. d.]. Github Commit 9. https:https://github.com/tordanik/OSM2World/commit/43596381115d6427f56240ce26ddc6bbd9a0e0d1. Accessed: 2023-04-14.

[9] Georges G. K. 2001. *A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems,.*