EasyChair Preprint

№ 6204

# Enhancing Software Quality Using Artificial Neural Networks to Support Software Refactoring

Parveena Sandrasegaran and Sivakumar Vengusamy



August 1, 2021

# Enhancing Software Quality using Artificial Neural Networks to support Software Refactoring

Parveena Sandrasegaran, V. Sivakumar
Asia Pacific University Malaysia
parveena6@gmail.com, dr.sivakumar@staffemail.apu.edu.my

***Abstract*— Current trends of software refactoring involve tools and techniques to eliminate code smells that hinder the software from achieving quality goals. This is carried out manually as the developer is required to analyse the system in order to identify how a particular quality attribute is being affected. This approach to software development is inefficient as a majority of software engineers lack this skill and it prolongs the time allocated for the software's implementation and maintenance. This dissertation outlines the need for Artificial Neural Networks to support software refactoring in order to enhance the system's quality. This justification is emphasized by means of illustrating the issues that arise when software quality is affected by the presence of codes smells that have been overlooked by the developers. By adhering to a research methodology that comprises of SEVEN major phases, an ANN model is able to measure software quality in terms of efficiency, maintainability, and reusability. This calculation is based off inputs that are generated through SciTools whereby an application is decomposed into metric parameters such as Cyclomatic Complexity (CC). The results of the quality of ELEVEN JAVA projects were quantified in order to further analyse patterns of code smells; this provides an insight on how the model may be utilized to enhance software quality. Furthermore, the performance of the model is evaluated relative to other Machine Learning (ML) models.**

***Index Terms*—Software Quality, Software Refactoring, Neural Networks, Code Smells, Refactoring Techniques**

## 1. Introduction

Software products that are produced by traditional or agile methodologies often require a prolonged implementation, testing, and maintenance phase. This is due to the coding style that several developers adopt by which unnecessary Lines of Code (LOC) are present in the system. This imposes a delay in the software's start-up and may degrade its performance. In addition, such systems do not meet the quality criteria based on McCall's Factor Model by which a system should attain specific quality standards that are categorized as product operation, product revision, and product transition factors. Ideally, the term quality is defined based on the user's perception of a quality system. However, general features influence this ideology of a quality system; such aspects include each classification of McCall's Factor Model respectively: efficiency, maintainability, and reusability. The core of degraded quality commences with programming concepts that have been implemented incorrectly.

Software refactoring is the process of reorganizing the code that makes up a software, it involves improving the internal structure of the system to enhance its non-functional properties without any modification to its behaviour (Satwinder Singh, 2018). This process is supported by a series of techniques that identifies defects that hinder the system from executing. Although current trends of software refactoring assist developers to identify and fix bugs and errors within the system, these techniques are applied manually which leads to the excessive consumption of time, effort, and resources (Alotaibi, 2018).

Developers are limited to identifying errors and defects with regards to the desired output from the code during its execution. Code smells are characteristics of a program that is a result of design issues which hinders the program from achieving particular quality goals (M. Mohan, 2019). Such characteristics continue to exist among the various classes and methods of the system regardless of whether manual refactoring has been performed. Artificial Intelligence (AI) is an approach considered to automate refactoring. Various studies on an AI subset known as Machine Learning (ML) have been conducted to critically analyse its effectiveness towards addressing issues in refactoring. Specifically, Artificial Neural Networks (ANN) have been designed to execute aspects of software refactoring. The following section aims to discuss problems that are not addressed by current software refactoring techniques.

### 1.1 Problem Statement

The current use of software refactoring improves the design of an application. However, it does not eliminate code smells; *such as data clumps, long method, refused bequest, and God class*; that have an adverse effect on software quality goals. These code smells violate the SOLID design principles resulting in an increased time, cost, and effort to cater novel requirements while simultaneously failing to achieve quality standards. The support of ANN specifically to measure software quality may be investigated to provide the developer with appropriate refactoring options to achieve quality goals that are hindered by code smells.

### 1.2 Aim

The aim of this research is to propose an ANN model that measures the quality of an application in order to manually

identify code smells affecting a specified set of quality goals. This assists in justifying the model's support for refactoring as this enhances software quality.

### 1.3 Objective

- To design an ANN model that measures software quality based on parameters that are prone to creating code smells
- To acquire and pre-process the data set using a repository tool known as GitHub
- To analyze the data set for object-oriented metrics using SciTools
- To assess testing strategies that enable the researcher to test the proposed ANN model and measure its impact on software quality
- To evaluate the ANN model relative to existing ML models

### 1.4 Significance of the study

Often coding errors with regard to the software's structure are created during implementation; however, these errors are overlooked and/or unidentifiable. This is costly to amend once the software has been deployed and will require double the effort and man hours to identify and eliminate the error (HaitaoZhao, 2019). Existing studies on ML models concluded a tainted quality as the system's complexity increased. Therefore, these models are applicable to support software refactoring to an extent. The study of ANN models to support software refactoring is significant as it is able to quantify software quality in order to recognize patterns of code smells regardless of a system's complexity (Chen, 2018). This is ideal in the specified context as several patterns of code smells are overlooked due to the system's complexity. Such complexity is commonly experienced in the software industry. Furthermore, the model may act as a tool to alert the developer of a decline in a particular quality in order to perform the corresponding refactor. The selection of ANN minimizes the effort, cost, and time required during software development and achieves software quality goals by embedding quality that is based on each category of McCall's Factor Model into the system.

## 2. Literature Review

Software refactoring is a major role in the implementation phase of the Software Development Life Cycle (SDLC). Existing software refactoring techniques are not automated hence resulting in an excessive consumption of time, effort, and resources (Alotaibi, 2018). Additionally, developers are limited to identifying errors and defects by means of monitoring the desired output from the code during its execution. Numerous experimental researches have been conducted to analyze the ability of a software refactoring technique to eliminate a particular code smell.

The presence of code smells hinders the software from being maintainable and expandable. Current software refactoring techniques do not promote a full elimination of the code smell. This suggests that code smells continue to exist among the various classes and methods of the system regardless of whether refactoring has been performed. Therefore, utilizing a set of algorithms intended for recognizing patterns, *known as neural networks*, may be used to improve the software refactoring process in order to accommodate particular quality attributes.

Using neural networks to support refactoring techniques is vital as it assists in minimizing the time, cost, and effort required to carry out the software refactoring process. Additionally, automating software refactoring has been proven to be complex and may result in added ramification to the code (O.Deryugina, 2019). Therefore, the use of neural networks rather than automating the process would provide the developer with a series of predictions of the software's quality in order to apply the appropriate refactoring techniques to the code.

This systematic literature review aims to identify and critically evaluate commonly utilized refactoring techniques that are applied manually or with the aid of tools. In addition, approaches to how neural networks may be used to support the process shall be analysed. The impact of neural networks used to enhance the quality of the system in the implementation and maintenance phases of the Software Development Life Cycle shall then be assessed. In order to understand the reliability of neural networks and its utilization with software refactoring, the following research questions shall be pursued.

RQ1. How is software quality impacted by the support of neural networks in refactoring techniques?
RQ2. How can neural networks be integrated with existing refactoring tools?
RQ3. What training algorithm can be applied to the neural network to assist software refactoring?

The structure of this chapter is as follows: The following section outlines related works and the search strategy employed to recognize the relevance of the work with consideration of the area of study. Subsequently, a discussion of the various software refactoring techniques shall be outlined to answer RQ1 and RQ2 respectively. Furthermore, a solution to RQ3 shall be explored prior to identifying the research gap and providing a conclusion to this literature review.

### 2.1 Related Works

Studies were carried out to analyse the impact of software refactoring in order to detect and eliminate code smells and anti-patterns. Several systematic literature surveys in the area of software refactoring to eliminate code smells have been carried out. In the discipline of software refactoring, specific milestones have been presented by several authors such as (Alotaibi, 2018), (Satwinder Singh, 2018), and (M. Mohan, 2019). A summary of the results presented by the systematic literature survey gathered by the respective authors is outlined in *Table 1*.

(Alotaibi, 2018) revealed various perspectives of the impact of software refactoring and the challenges that arise during the implementation and maintenance phases due to the current form of refactoring. More than 30 papers were filtered from search engines such as ScienceDirect and Elsevier to extract and analyse the findings of the paper. This was carried out to grasp a deeper understanding of the influence of the current mode of refactoring to the Software Development Life Cycle.

A systematic literature survey was conducted by (Satwinder Singh, 2018) on the study of refactoring techniques to eliminate smells in the code. 1053 research items between 2015 and 2018 were reviewed to understand uniform methods to optimize the code by eliminating code smells. These items were filtered and gathered from IEEE, Elsevier, ACM, and Springer. Approximately 200 articles were refined after applying an inclusion-exclusion criterion to the research items.

Similar to the systematic literature survey conducted by (Alotaibi, 2018), (M. Mohan, 2019) exhibits the opportunities that emerges to counteract the challenges faced in the current utilization of software refactoring. 58 papers dated between 2012 to 2017 were reviewed on the basis of the investigation. These papers were published by leading Journals such as Springer and IEEE. Based on the systematic literature survey, the author concluded that the current trend of software refactoring (manual) affects other aspects of the code. This is exemplified in a scenario whereby a software refactor in one class causes errors to arise within the same class or another class.

| Citation | Databases Searched | Number of reviewed articles | Refactoring Technique discussed | Benefits | Code Impurities Addressed | Comments |
|---|---|---|---|---|---|---|
| (Alotaibi, 2018) | ScienceDirect, Elsevier | 30 | Encapsulate Field | Easier to maintain and develop objects<br>Enables complex operations in order to contact object fields | Data Class | - Refactoring tool available is limited to OO Programming<br>- Developers are more likely to apply this technique manually rather than using the automated tool due to complexity |
| (Satwinder Singh, 2018) | IEEE, Elsevier, ACM, Springer | 200 | Extract Method | Readable code<br>Decreased code duplication<br>Lesser probability of errors | Long Method, Refused Bequest, Data Clumps, Duplicate code | - Refactoring tool available is limited to OO Programming<br>- Decision trees are utilized to evaluate approaches to apply the extract method and eliminate the code impurities addressed |
| (Mesfin Abebe, 2014) | Springer, IEEE | 58 | Pull Up Field | Eliminates redundant fields within subclasses<br>Manages relocation of redundant methods | Long Parameter List, Duplicate code | - Refactoring tool available is limited to OO Programming<br>- Manual code refactoring is used, this results in unnecessary consumption of resources as redundant codes are overlooked whilst utilizing the supported tool |

*Table 1 Summary of systematic literature survey*

## 2.2 Search Strategy

The process of refining research articles commences by specifying essential keywords related to the area of study. Additionally, numerous leading Journals; *such as IEEE and Elsevier*; were identified to commence the search. These papers then went through two levels of refinements in order to filter appropriate papers that added valuable knowledge to the area of study. Initially, 50 papers were filtered based on the relationship between the title of the paper and the research area. The research papers were further refined to 25 papers on the basis of its abstract. The final concentrated research articles were then reviewed and analysed to identify similarities between the papers and limitations to the study. Based on the acquired papers, the researcher was able to justify the use of neural networks to aid software refactoring techniques in order to enhance software quality. The outcome of the papers reviewed is discussed in the following section.

## 2.3 Software Refactoring Technique Analysis

The following section aims to discuss possible solutions to the research questions outlined in the introduction section of this paper. Three software refactoring techniques have been selected to focus on the problems that it addresses and challenges that are faced with the current delivery of the technique. These techniques shall be evaluated on the basis of how the software's quality is influenced as a result of the refactor. Furthermore, the possible integration of neural networks with these techniques shall be critically analysed to justify its ability to enhance the software's quality with focus to improving productivity in the Software Development Life Cycle.

### 2.3.1 Encapsulate Field

A key feature that object-oriented programming comprises of is encapsulation. This feature is used to encourage the utilization of access specifiers rather than allowing all objects

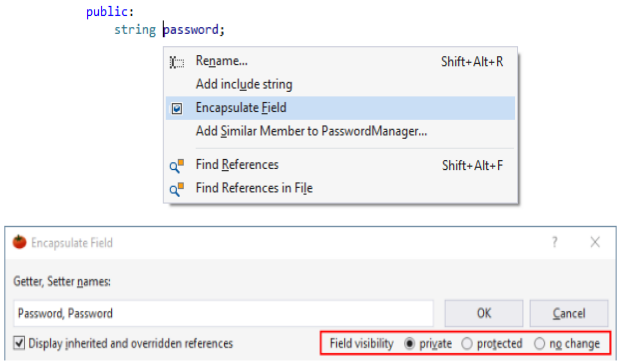



*Figure 1 Refactoring using Encapsulate Field in Visual Assist*

to be public. Code is refactored using this technique by suggesting the use of getter and setter methods for a particular field; the field is then converted from public to private. Additionally, data class is the code smell that this refactoring technique addresses whereby a newly created class comprises of objects containing only data (Giovanni Grano, 2019). This code smell violates the Single Responsibility Principle as a class that comprises of numerous objects containing data to be utilized by other classes would require multiple reasons to change and will involve major reworks to the system to accommodate the necessary changes. The violation of this principle degrades the system's maintainability quality attribute.

Visual Assist is a refactoring tool embedded within Microsoft Visual Studio that implements Encapsulate Field. *Figure 1* illustrates how this tool is utilized during the implementation phase. However, the use of this tool only enhances the system's security and integrity attributes rather than eliminating code smells that violate the Single Responsibility Principle. Neural networks may be integrated with this tool by assessing the nature of the program in order to provide suggestions to the developer to apply the Encapsulate field. This ensures that unauthorized parties do not have direct access to the field.

### 2.3.2 **Extract Method**

Code fragments that are grouped together into a single method should be separated into a new method. Likewise, the existing method should include a method call to the new method, this is performed using a refactoring technique known as Extract Method. This technique improves the code readability and decreases code duplication and the probability of errors occurring within the class. Extract method addresses several code smells that affects the program's design and implementation such as data clumps, long method, refused bequest, and duplicate code.

| **Code Smell** | **What is it?** | **Quality factors impacted** |
|---|---|---|
| Data Clumps | Code that comprises of identical groups of variables that were the effect of copying and pasting open source code snippets | Flexibility, Reliability |
| Long Method | An oversized method that comprises of too many Lines of Code (LOC) (Gupta, 2019) | Efficiency, Maintainability, Reusability |
| Refused Bequest | Creating inheritance to use objects in a parent class, however, both the child and parent classes are entirely dissimilar | Testability, Reliability |
| Duplicate Code | Redundant code within the same method, class, or program | Efficiency, Maintainability, Reusability, Flexibility, Reliability, Testability |

*Table 2 Impact of code smells to software quality*

*Table 2* summarizes the code smells that Extract method discloses and its influence to software quality. Extract method is supported in a refactoring tool known as Smalltalk Refactoring Browser whereby code is restructured without any modifications to its behaviour. In contrast to Visual Assist, this refactoring tool requires the programmer to select a method's component that should be extracted as a separate method relative to the code smells that are addressed. This tool does not enhance the productivity of novel programmers that are not able to recognize the code smells. Therefore, neural networks many be integrated with this technique by initially recognizing the code smells that exist within the program. Once these smells have been detected, the developer may be notified to apply the extract method to the area where the code smell exists. The application of neural networks will uphold software quality as it assists in immediate detection of the code smells to apply the extract method.

### 2.3.3 **Pull Up Field**

Problems arise when two or more classes comprise of the same field as redundant changes are required when functionalities are added to the system. Pull up field is a refactoring technique that is used to counteract this problem by moving the same field to a superclass that extends the corresponding classes and removing redundant code. Additionally, long parameter list and duplicate code are the main code smells that are addressed by pull up field.

Long parameter list is a common code smell that is encountered by several programmers for several reasons that are dependent on the nature of the class. The primary cause of long parameter lists from arising is due to the necessity of a series of algorithms to be utilized in order to obtain a particular result. This is recurrent when using different algorithms to capture and extract random features of an image. Long parameter lists may also arise when classes are constructed more independently, this decreases the dependency among the various classes and affects the efficiency of passing parameters between classes.

Similar to long parameter list, duplicate code is a code smell that commonly occurs in the programming industry whereby several programmers are working on various segments of a program simultaneously (Sievi-Korte, 2019). Programmers are unaware of the same code being applied within the same program at the same time; this affects the testability, maintainability, and efficiency of the program. Testability is affected as it would introduce redundant errors and bugs which will result in an increase in testing efforts. Similarly, it replicates the need to maintain the same set of code rather than maintaining and applying novel functionalities once. Furthermore, the time taken to execute the program would be prolonged due to the compiling and execution of duplicate code.

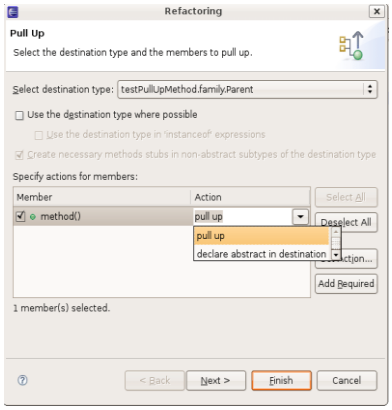



*Figure 2 Pull Up Field implemented in the Eclipse IDE*

The Eclipse IDE comprises of a refactoring tool that implements the Pull Up Field technique by declaring abstract methods in a parent class and/or relocating a field to a parent class, this is illustrated in *Figure 2*. This tool provides the developer with an insight of the refactoring results prior to executing a refactor. This is previewed as a tree that displays each component within the program that will be affected by the refactor. By integrating neural networks, the developer is able to be notified of overlooked code smells such as redundant code as they are required to manually detect code

smells and utilize the tool to refactor the code. The integration of neural networks with the Pull Up Field technique contributes to the software's testability, maintainability, and efficiency.

### 2.4 **Using Neural Networks to detect code smells**

Neural networks can be utilized to support several software refactoring techniques by recognizing patterns of code smells in order to perform a refactor (Ouni, 2017). This will decrease the effort required and improve the productivity of the developers during the implementation and maintenance phases. This is due to the automated detection of code smells rather than developers analysing the code to detect such code smells. Similarly, the utilization of neural networks in software refactoring would assist novel programmers to recognize patterns of code smells and enhance their skills with regards to writing quality code. In order to recognize patterns of code smells, the neural network shall segregate the program into *n* number of sections to conduct a predictive analysis that forecasts when a particular form of code smell will appear. Based on this analysis, the developer is notified of the potential code smell to perform the respective refactor.

#### 2.4.1 **Newton's Method**

Newton's method is an algorithm that uses the Hessian matrix to train neural networks for optimization (Oliveira, 2019). This algorithm searches for a finer training direction by using the second derivatives of a function that plots an event onto a real number in order to represent a "cost" that is linked to the event, this is known as a loss function (Ting, 2019). The premier Newton's training direction is initially acquired in order to augment the performance of the parameters and training rate. This training algorithm is preferred over several training algorithms due its ability to converge to the root promptly (Mujtaba Alshakhouri, 2018). This is essential during the development of the software as it increases the productivity of the developers and improves the quality of the code with regards to catering for efficiency, testability, and maintainability. Furthermore, the expression below depicts Newton's method with regards to training the neural network to recognize code smells whereby $\boldsymbol{a}$ = *the first Newton's training direction*.

$$\boldsymbol{a} = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{(x_k)^2}{2}\frac{f"(n_k)}{f'(x_k)}$$

$x = Vector\ point$
$n = Divided\ number\ of\ sections\ of\ a\ program$
$k = Iteration$

#### 2.4.2 **Variable Learning Rate Backpropagation**

Variable Learning Rate Backpropagation is a training algorithm that iteratively processes a set of training data in order to learn (Ting, 2019). To implement this algorithm, the input pattern $(X_i)$ and target outputs $(X_o)$ are expressed as $(X_i, X_o)$. The input pattern requires specific outputs of each neuron throughout the layer. The output layer is responsible for presenting a certain output based on the inputs; therefore, the actual and target outputs provide an error signal that differentiates the two from each other (Ting, 2019). Furthermore, the value that each neuron and weight carries influence this error signal. The error is propagated backwards after calculating its value; this updates the values of the weights and bias in order to achieve a better accuracy. As outlined in the equation below, the error is continuously adjusted until the validation dataset ensures that the model is underfitted (Davide Arcelli, 2018).

$$\boldsymbol{E_j} = O_j(1 - O_j)\sum_{i=0} E_i W_{ij}$$

$E = Error$
$O = Target\ output$
$W = Weights$
$i = Iteration$
$j = Gradient$

#### 2.4.3 **Levenberg-Marquardt**

By computing a matrix, *known as the Jacobian Matrix*, that contains the first-order partial derivative of a vector valued function, the Levenberg-Marquardt algorithm *(represented in the expression below)* is able to proceed towards a second-order training rate (Almas Hamid, 2018). The algorithm offers dual possibilities to converge in different directions for a single iteration, this enforces a more robust execution. Although, input parameter spaces may be at loss in this algorithm, it is able to identify optimal solutions in order to attain high accuracy (Almas Hamid, 2018). Furthermore, this allows the algorithm to achieve a better performance, speed, and accuracy in comparison to Variable Learning Rate Backpropagation and Newton's method algorithm.

$$\boldsymbol{S} = argmin_\beta \sum_{i=1}^{m} [y_i - f(J_i, \beta)]^2$$

$S = Sum\ of\ square\ deviatqions$
$\beta = Parameter\ vector$
$f = Gradient$
$J = Jacobian\ Matrix$
$m = Emphiracal\ Pairs$

## 3. **Research Gap and Summary**

Software refactoring is a major role in the implementation phase of the Software Development Life Cycle. This process has not evolved to result in an automated procedure as developers are required to manually analyze the program to recognize code smells that have to be eliminated using software refactoring. However, several code smells remain within the program as they have been overlooked by developers due to the lack of skill or refactoring tool constraints. Neural networks are an agile approach to recognizing patterns of code smells in order to perform a refactor and enhance the quality of the software. This approach to support the refactoring process may be supported by utilizing training algorithms such as Newton's method, Variable Learning Rate Backpropagation, or Levenberg-Marquardt to understand the nature of the program and recognize code smells rapidly. This study is limited to common refactoring techniques and commonly utilized training algorithms that are able to predict outputs based on novel inputs. Future research with regard to automating software

refactoring or utilizing evolutionary computations to enhance the application of software refactoring may be required.

## 4. Research Methodology

This section presents a suitable research methodology that encourages a systematic approach to conducting an experimental research based on quantitative data. The methodology consists of SEVEN distinct phases that produces minor deliverables to document the administration of the research accordingly. The phases include data collection, data pre-processing, design, implementation, testing, validation, and evaluation.

### 4.1 Data Collection

The datasets that are required during the testing phase of the research will be acquired from GitHub. It acts as a public code repository that allows users to upload various projects that varies in consideration of programming languages that were used for its development. FIFTEEN Java Projects that have been uploaded into the repository were collected with varying complexity, LOC, and various legacy systems that comprises of code smells; this assists the researcher in justifying the use of ANN to enhance software quality. The selected projects become the input of the subsequent phase.

### 4.2 Data Pre-Processing

| No. | Project Title | WMC | NCC | CC | MI | RC | ACU | HV |
|---|---|---|---|---|---|---|---|---|
| 1 | Application Performance Monitoring System (APM) | 8 | 1 | 30 | 64.1 | 18 | 30% | 4512.64 |
| 2 | Web Conferencing System (WCS) | 3 | 0 | 42 | 75.2 | 16 | 26.4% | 4512.06 |
| 3 | School Management System (SMS) | 5 | 5 | 35 | 63.24 | 8 | 31% | 4271.95 |
| 4 | Book Management System (BMS) | 4 | 3 | 32 | 52.87 | 7 | 35.1% | 4263.71 |
| 5 | Customer Management System (CMS) | 8 | 2 | 29 | 84.96 | 14 | 25.1% | 4266.12 |
| 6 | Distributed Configuration System (DCS) | 54 | 0 | 56 | 33.42 | 21 | 57% | 5137.45 |
| 7 | Document Management System (DMS) | 14 | 2 | 21 | 82.37 | 2 | 6% | 4249.23 |
| 8 | Library Management System (LMS) | 8 | 3 | 23 | 55.52 | 7 | 38.1% | 4263.53 |
| 9 | Inventory Management System (IMS) | 6 | 4 | 27 | 63.41 | 15 | 31. 4% | 4262.37 |
| 10 | Warehouse Management System (WMS) | 9 | 4 | 34 | 67.3 | 13 | 36% | 4277.02 |
| 11 | Employee Management System (EMS) | 4 | 2 | 19 | 97.72 | 0 | 23% | 4257.41 |

*Table 3 Extracted metric parameters of the refined Java Projects*

Subsequently after the Java Projects have been collected, the data shall be analysed to identify any missing data. The execution of the projects in the NetBeans IDE assists in identifying any missing data that involves absent dependencies between classes and/or loss of values/variables in a method/class. Noisy, missing, and inconsistent data shall be discarded from the acquired data set. The refined projects were then analysed using SciTools in order to identify the independent variables; this is outlined in *Table 3*. The refined and analysed dataset shall be stored in an Excel spreadsheet for the implementation and testing phases of the research.

### 4.3 Design

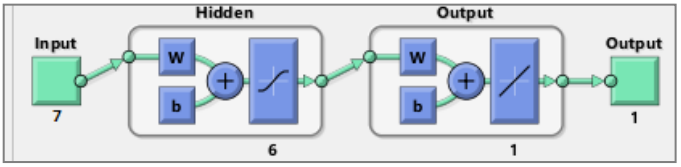



*Figure 3 Single Layer Architecture*

The formulation of ANN specific to enhance software efficiency, maintainability, and reusability is designed during this phase. As illustrated in *Figure 3*, the ANN comprises of a Single Layer Architecture that is embodied by THREE distinct layers: input, hidden, and output layer. This architecture receives metric parameters that are organized in different columns. This input enters the hidden layer and is dependent on the set bias and weights. SIX neurons form the hidden layer; the output it produces is relative to the activation function initiated. The data produced in the output layer is then utilized to quantify the efficiency, maintainability, and reusability of the corresponding application in consideration of the metric parameters that have been extracted.

### 4.4 Implementation

MATLAB is the IDE that shall be used to simulate the ANN model, this is due to the deep learning toolbox that is offered in the platform. The analysed dataset that was stored during data pre-processing shall be imported to IDE in order to act as an input for the ANN model. In order to ensure that the neural network is able to accept the imported inputs, it is essential to convert the numerical values in the spreadsheet into binary values in an array.

As stated in the design phase, the hidden layer comprises of SIX neurons. These neurons act as a mathematical function that receives *n*-number of inputs that are mathematically weighted separately. The sum of this computation subsequently passes through an activation function. The selected activation function comprises of a hyperbolic tangent function, $x(j) = \tanh(\beta_1 + \beta_0 \sum_i j, k_j)$, as it is useful for classification issues. Additionally, the use of this activation function ensures that the probability of the network getting wrapped remains at a minimal as it avoids slowing down the network's classification.

The output layer as a similar design to the hidden layer. However, the further computation with its neuron provides a predicted probability of the software's necessary coefficient. This value is inserted into the respective quality metrics to quantify the software's efficiency, maintainability, and reusability. Furthermore, the results from the output layer undergoes a Variable Learning Rate Backpropagation algorithm that applies a chain rule. This minimizes the network's chance of error by calculating the direction of the steepest descent and executing a backward pass in order to adjust the parameters of the model.

MATLAB comprises of a Machine Learning Toolbox that assists in visualizing the network's architecture, progress, and plots. However, the results of this feature are subject to the

IDE's pre-programmed training algorithm: Levenberg-Marquardt that may result in a different performance and error histogram.

### 4.5 Testing

The results that were gathered from the execution phase involves the predicted and quantified software quality with regard to efficiency, maintainability, and reusability. The metric's parameters are analysed in order to manually identify code smells in consideration of Design Principles. The standard recognition of code smells, *discovered by* (Kim, 2017), is in reference to *Table 4*. These rules are established to discover the effect of the code smell with the corresponding quality attribute. The result of this phase is documented in the following chapter.

| Code Smell | Rules | Affected quality attribute |
|---|---|---|
| Large Class | ACU>70% \|\| RC>20 | Maintainability, Efficiency |
| Lazy Class | RC=0 \|\| CC<3 \|\| WMC<=2 | Usability, Reliability |
| Data Class | HV>4500 \|\| WMC>50 | Flexibility, Testability |
| Parallel Inheritance Hierarchies | NCC>4 | Reusability, Portability, Flexibility |
| God Class | WMC>=47 \|\| HV>4900 | Efficiency, Reliability, Usability, Maintainability, Flexibility, Testability, Portability, Reusability |
| Feature Envy | RC>20 \|\| WMC>50 | Reusability, Efficiency |
| Data Clumps | MI>60 \|\| ACU>50% \|\| CC >=35 | Maintainability, Efficiency |
| Long Parameter List | RC>=15 \|\| MI>70 | Maintainability, Reusability |

*Table 4 Rules for code smell identification*

### 4.6 Validation

Validation of the neural network is carried out by calculating the classification error of the neural network. This is performed by utilizing the weights generated from the model's training *(model training is elaborated in the Implementation phase)*. In a scenario where the classification error is high, a higher threshold is declared and the process of presenting the input or target vectors to compute novel weights and biases for the network is repeated; this measure is known as *epoch*. This phase is partially demonstrated as "`net.divideParam.valRatio = 15/100;`". The data fraction that is situated in the validation set is 0.15 by default. This may be manipulated with to identify which ratio produces a minimal classification error.

### 4.7 Evaluation

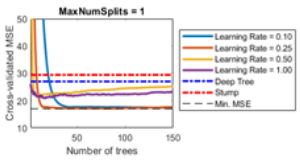
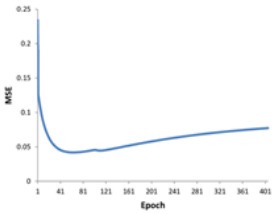
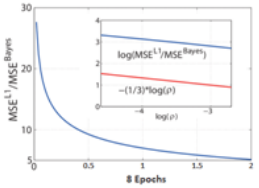
| Model | MSE | Citation |
|---|---|---|
| Decision Tree |  | (Mostafazadeh-Fard, 2012) |
| SVM |  | (Andre Carnieletto Dotto, 2018) |
| BBN |  | (Junzi Sun, 2017) |

*Table 5 MSE plots for various ML models*

MSE is known as a risk function that refers to the desired value of an error loss. As depicted in *Figure 4*, the MSE value decreased towards the end of the training phase. This signifies that the network had been trained accordingly. The value of MSE is required to be proximate to zero as the desired and actual outputs are obliged to be adjacent to each other.

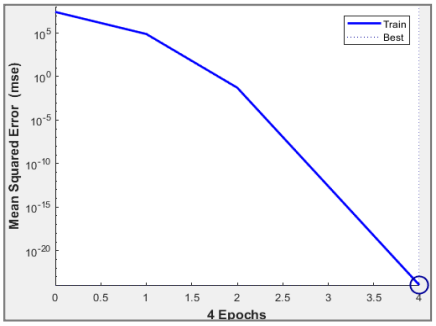



*Figure 4 Performance plot of the model*

*Table 5* illustrates the MSE plots for decision trees, SVM, and BBN. In contrast to ANN, BBN, and SVM; decision trees are not relative to the number of epochs *(an epoch is the digit of repetition where all training vectors are utilized to update the weights)* (Kahlon, 2015). Comparatively they correspond to the number of trees that constructs the model. Furthermore, it undergoes cross validation which requires excessive processing power and training time. With regard to SVM, the best performance of the models is attained at epoch 39. Whereas the best performance was 1.1132e-24 at epoch 4 in the proposed ANN, this value may be improved by ensuring that the input parameters are less than or equal to the output parameters as this will utilize less memory and would accelerate the model's training as illustrated in BBN's best performance that is attained at epoch 2 (Kahlon, 2015).

## 5. Results and Analysis

The results from the testing phase has been summarized in *Table 5*. It highlights the code smells that are present in the application; therefore, resulting in a decline in software quality. 63% of the total JAVA projects comprised of code smells. The difference in quality amongst the projects is synthesized in this section.

| Project | Efficiency | Maintainability | Reusability | Identified Code Smell |
|---|---|---|---|---|
| APM | 24.5 | 85.7587 | 11.9460 | Data class, Feature envy, Long Parameter List |
| WCS | 23.5 | 73.7596 | 5.6540 | Data class, Data clumps, Long Parameter List |
| SMS | 19.5 | 81.1699 | 35.09 | Data clumps, Parallel Inheritance Hierarchies |
| BMS | 17 | 95.2099 | 58.2340 | - |
| CMS | 19.5 | 84.1801 | 23.5180 | Long Parameter List |
| DCS | 39 | 58.7858 | 5.6540 | Data class, Data clumps, Feature Envy, God class, Large class |
| DMS | 4 | 84.1843 | 23.5180 | Long Parameter List |
| LMS | 18 | 93.1847 | 43.0900 | - |
| IMS | 17.5 | 89.1867 | 46.6620 | - |
| WMS | 17 | 82.1610 | 51.9420 | - |
| EMS | 11.5 | 87.1954 | 23.5180 | Lazy class, Long Parameter Lists |

*Table 6 Quantified quality of the JAVA projects*

### 5.1 Efficiency

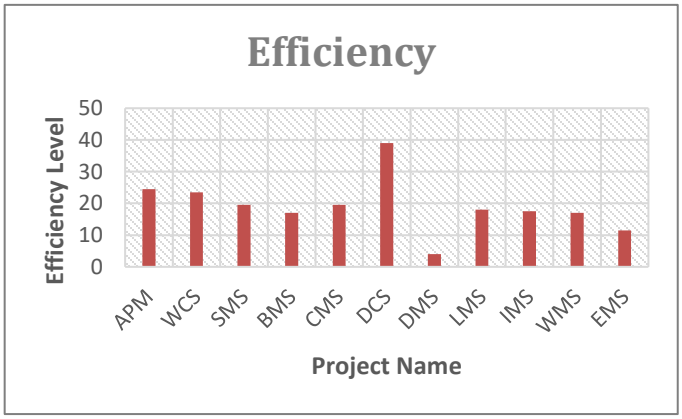



*Figure 5 Efficiency levels of the tested JAVA projects*

*Figure 5* provides a visual representation of the efficiency level relative to the JAVA projects tested. The lower the efficiency level, the higher the efficiency for the reason that the efficiency level represents the response time and average CPU utilization ratio. This suggests the speed of the application as the lesser the response time of the application, the increase in user productivity and throughput.

Furthermore, the application's efficiency is affected when data clumps, feature envy, God class, and/or large class are present in the program. These code smells exist in the following projects: APM, WCS, and DCS. Among these projects, DCS exhibits the most decline in efficiency which results in user dissatisfaction due to a prolonged response time. Additionally, the existence of this code smell violates the Interface Segregation Principle as the interface of a class is not separated into different methods (Giovanni Grano, 2019). However, this may be subdued by implementing an Adapter design pattern to decouple the system and organize the delegation of methods in order to improve the system's efficiency.

### 5.2 Maintainability

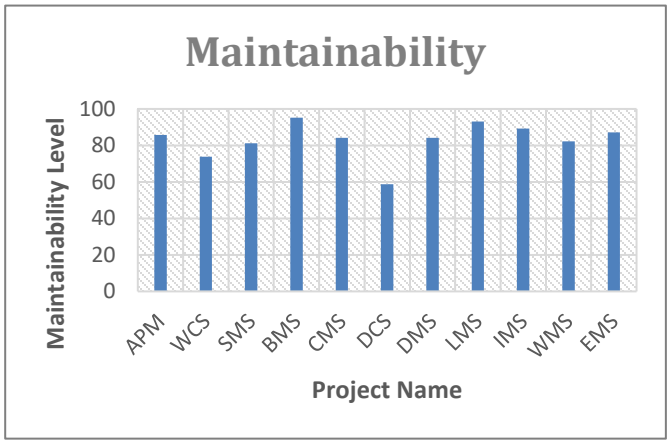



*Figure 6 Reusability levels of the tested JAVA projects*

Most applications that comprise of code smells often endure maintainability issues (Dag I.K. Sjøberg, 2019). This is due to the nature of code smells that forces the application to adopt a bad design. In contrast to the efficiency levels, the higher the maintainability level, the more sustainable the system. As illustrated in *Figure 6*, projects that have the highest maintainability levels do not contain code smells.

Data clumps, God class, large class, and long parameter list are the code smells that affect maintainability in addition to efficiency. These code smells violate the Open-Close Principle whereby a system should be open for extension but closed for modification (Dag I.K. Sjøberg, 2019). This is due to the need to change existing methods/classes that have been tested in the earlier phases of software development. Additionally, there would be an increase in cost and effort to implement a single new feature for the reason that the maintenance phase of software development consumes 67% of the total effort and cost (Dag I.K. Sjøberg, 2019). However, the maintainability of the system may be improved by implementing the Strategy or Template pattern to avoid this design violation.

### 5.3 Reusability

Reusability levels are calculated differently than maintainability levels; however, they are interpreted the same *(the higher the reusability level, the better the component's reusability)*. Maintainability and reusability are interrelated as components that are tightly coupled and complex will not be fit to support a different system with a similar nature (Gupta, 2019). In addition to Lazy class, the projects comprise of the exact code smells that affect its maintainability. The decreased reusability of APM, WCS, and EMS differs the projects from the maintainability computation; this is visualized in *Figure 7*.

In conjunction with the violation of the Open-Close Principle, the projects breach the Dependency Inversion Principle. This does not ease the process of using components of a class for a different project as several errors will arise (Almas Hamid, 2018). Code reuse is a fundamental goal that assists in reducing time-effort, and cost of software development. Therefore, the structure of the system should be carefully planned out to ensure it adopts a well-designed, OO application (Giovanni Grano, 2019).

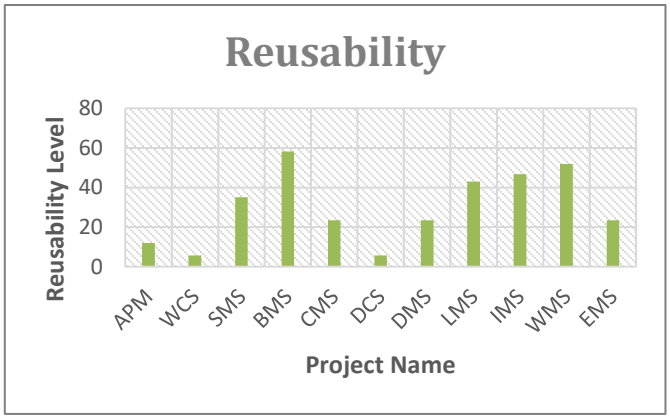



*Figure 7 Maintainability levels of the tested JAVA projects*

## 6. Conclusion and Future Research

This dissertation has successfully outlined issues in the area of study that is being addressed in this research. A software's complexity decreases its efficiency, maintainability, and reusability; this may result in a software that does not accommodate changes. By recognizing issues in the system, ANN is the proposed solution due to its potential to support refactoring options that would optimize the code and augment several quality attributes. Additionally, a set of research questions were developed to assist the researcher with regard to investigating the proposed model to acknowledge the challenges faced in software development. The scope of the research is clearly defined to ensure that the research conducted is within the disclosed boundaries in order to prevent delays and direct the focus on the proposed solution.

The execution of this research is procedurally documented in Chapter 3: Research Methodology. This enabled the researcher to successfully implement an ANN model that is specifically designed to determine software quality. The results of the chapter are documented in Chapter 4: Results and Analysis. The two sets of results included the quantified quality and performance measures of the proposed model. Based on the measured quality, it was discovered that applications that hosted code smells had a lesser quality than applications that were unrestricted from code smells. Additionally, the performance measures gathered were differentiated from other ML models. The proposed model did not have the best performance; however, the differentiating environment consisted of varying independent variables that are not able to solidify this conclusion.

It is promising that the ANN model is able to predict the quality attributes accurately based on metric parameters generated by SciTools. This measure is able to support software refactoring by displaying the quantified quality attributes to the developer. Further research should be conducted to investigate the possibility of embedded refactoring tools in an IDE determining which parts of the code should be refactored in order to enhance the system's quality.

## 7. References

Almas Hamid, M. I. M. H., 2018. A Comparative Study on Code Smell Detection Tools. *International Journal of Advanced Science and Technology* , Volume 60, pp. 25-32.

Alotaibi, M., 2018. Advances and Challenges in Software Refactoring: A Tertiary Systematic Literature Review. *ROCHESTER INSTITUTE OF TECHNOLOGY*, Volume 1, pp. 5-9.

Andre Carnieletto Dotto, S. G., 2018. A systematic study on the application of scatter-corrective and spectral-derivative preprocessing. *Journal of Artificial Intelligence,* Volume 31, pp. 23-41.

B.Delahunt, C., 2019. Putting a bug in ML: The moth olfactory network learns to read MNIST. *Neural Networks*, Volume 118, pp. 54-64.

Carvalho, M. L. L., 2018. On the implementation of dynamic software product lines: An exploratory study. *The Journal of Systems and Software*, 13(6), pp. 74-100.

Chen, R., 2018. A Robust Learning Approach for Regression Models Based on Distributionally Robust Optimization. *Journal of Machine Learning Research*, 19(1), pp. 1-26.

Dag I.K. Sjøberg, A. Y. B. A. A. M. T. D., 2019. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 14(12), pp. 1-14.

David G. Brown, F. W. S., 2019. Pitfalls and Opportunities in the Development and Evaluation of Artificial Intelligence Systems. *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, 1(2), pp. 139-159.

David Shriver, D. X. S. E. M. B. D., 2018. *Refactoring Neural Networks for Verification.* Stockholm, Sweden, Proceedings of the 35th International Conference on Machine Learning.

Davide Arcelli, V. C., 2018. Performance-driven software model refactoring. *Information and Software Technology*, 9(5), pp. 366-397.

El-Sharkawy, S., 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, 10(6), pp. 1-30.

Giovanni Grano, F. P. H. C. A. B., 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 18(1), pp. 1-15.

Gupta, P., 2019. Multi-objective optimization framework for software maintenance, component evaluation and selection involving outsourcing, redundancy and customer to customer relationship. *Information Sciences*, 48(3), pp. 21-52.

HaitaoZhao, 2019. Global-and-local-structure-based neural network for fault detection. *Neural Networks*, Volume 118, pp. 43-53.

Ibrahim, R., 2018. Reducing redundancy of test cases generation using code smell detection and refactoring. *Journal of King Saud University – Computer and Information Sciences*, 5(10), pp. 1-8.

Javed, M. A., 2007. *A Rationale Focused Software Architecture Documentation method (RFSAD).* Göteborg, Sweden : IT UNIVERSITY OF GÖTEBORG.

José Amancio M. Santos, J. B. R.-J. L. C. L. P., 2018. A systematic review on the code smell effect. *The Journal of Systems & Software,* Volume 144, pp. 450-477.

Junzi Sun, J. H. J. E., 2017. *Bayesian Inference of code smells.* Seattle, AIP Publishing.

Kahlon, K. S., 2015. Object oriented software metrics threshold values at quantitative acceptable risk level. *Springer Link,* 2(3), pp. 191-205.

Kaur, S., 2015. A Comprehensive Review of Refactoring Techniques. *IJLTEMAS,* 9(10), pp. 1-6.

Kim, D. K., 2017. Finding Bad Code Smells with Neural Network Models. *International Journal of Electrical and Computer Engineering (IJECE) ,* 7(6), pp. 3613-3621.

M. Mohan, D. G., 2019. Using a many-objective approach to investigate automated refactoring. *Information and Software Technology,* 11(2), pp. 83-101.

Maen Hammad, A. L., 2018. Automatic Detection of Bad Smells from Code Changes. *International Review on Computers and Software,* 12(1), pp. 2-8.

Mesfin Abebe, C.-J. Y., 2014. Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review. *International Journal of Software Engineering and Its Applications,* 8(6), pp. 299-318.

Morales, R., 2017. On the use of developers' context for automatic refactoring of software anti-patterns. *The Journal of Systems and Software,* 12(8), pp. 236-251.

Moskalenko, V., 2019. The Model and Training Algorithm of Compact Drone Autonomous Visual Navigation System. *Data Stream Mining and Processing,* 4(1), pp. 19-34.

Mostafazadeh-Fard, B., 2012. Linear and Nonlinear Modeling for Predicting Code Smells. *Journal of Artificial Intelligence,* 29(8), pp. 765-775.

Mujtaba Alshakhouri, J. B. S. G. M., 2018. Synchronised visualisation of software process and product artefacts: Concept, design and prototype implementation. *Information and Software Technology,* 9(8), pp. 131-145.

Muslim, M. T., 2017. Manifold absolute pressure estimation using neural network with hybrid training algorithm. *PLoS ONE ,* 12(11), pp. 51-56.

O.Deryugina, 2019. *Analysis of the AnyWalker Software Architecture Using the UML Refactoring Tool.* s.l., Elsevier.

Oliveira, J., 2019. Revisiting the refactoring mechanics. *Information and Software Technology,* 11(1), pp. 136-138.

Ouni, A., 2017. MORE: A Multi-objective Refactoring Recommendation Approach to Introducing Design Patterns and Fixing Code Smells. *JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS,* 9(1), pp. 1-67.

Peters, R., 2017. Evaluating the Lifespan of Code Smells using Software Repository Mining. *IEEE Transactions on Software Engineering,* 13(12), pp. 1-6.

Samir, A., 2016. Reusability Quality Attributes and Metrics of SaaS from Perspective of Business and Provider. *International Journal of Computer Science and Information Security (IJCSIS),* 14(3), pp. 295-309.

Satwinder Singh, S. K., 2018. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal,* Volume 9, pp. 2129-2135.

Sievi-Korte, O., 2019. Challenges and recommended practices for software architecting in global software development. *Information and Software Technology,* 10(6), pp. 234-253.

Ting, D. S. W., 2019. Artificial intelligence and deep learning in ophthalmology. *BMJ,* 10(3), pp. 167-175.

Tummala Pradeep, P., 2011. Comparison of variable learning rate and Levenberg-Marquardt back-propagation training algorithms for detecting attacks in Intrusion Detection Systems. *International Journal on Computer Science and Engineering (IJCSE),* 3(11), pp. 3572-3580.

Wu, J., 2016. Open software architecture for east articulated maintenance arm. *Fusion Engineering and Design,* 10(9), pp. 474-479.

YILMAZ, M., 2013. *A Software Process Engineering Approach to Understanding Software Productivity and Team Personality Characteristics: An Empirical Investigation.* Dublin: Dublin City University.