Extract Class Refactoring by analyzing class variables

by

Jasmeet Singh

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Simanta Mitra, Co-major Professor

Carl K. Chang, Co-major Professor

Shashi K. Gadia

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

CMM	Cognate Members Metric
VSM	Variable based Similarity between Methods
SSM	Structural Similarity between Methods
CDM	Call based dependence between Methods
CSM	Conceptual Similarity between methods
ECR	Extract Class Refactoring
AST	Abstract Syntax Tree
OO	Object Oriented
IDE	Integrated Development Environment
LCOM	Lack of Cohesion of methods
LCOM HS	Lack of Cohesion of methods by Henderson-Sellers
GUI	Graphical User Interface

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to Dr. Simanta Mitra for his substantial guidance and support throughout the course of this research. I would also like to extend my appreciation to my co-major professor Dr. Carl K. Chang and my committee member Dr. Shashi K. Gadia for their help and support.

In addition, I would also like to thank my friends, colleagues, the department faculty, and department staff for making this journey not only possible, but a memorable one.

Last but not the least; I would like to extend my gratitude to my parents, Mr. Surjit Singh and Mrs. Harjeet Kaur for their unbounded support and love.

ABSTRACT

Software maintenance activities often cause design erosion and lead to increased software complexity and maintenance costs. Extract Class Refactoring attempts to address design erosion by identifying and pulling out extraneous functionalities from a class and distributing them to new classes. This thesis extends previous research in this area by improving a metric known as Structural Similarity between Methods (SSM) used during Extract Class Refactoring. The improved metric, called Variable based Similarity between methods (VSM), establishes similarities between methods based on the variables they share, and on how they use these variables. Strongly connected methods are then allocated into new classes. The thesis also introduces another metric, Cognate Members Metric (CMM), which identifies those members of a class that are only used in combination with each other, and hence, probably belong together in a separate class. Additionally, this work extends and modifies existing refactoring processes for extracting classes. A software prototype that performs Extract Class Refactoring has been developed to substantiate the research. A few Case studies are discussed and comparison and analysis of results of refactoring using the new and older approaches of the Extract Class Refactoring process are presented.

CHAPTER 1: INTRODUCTION

1.1. Blob Anti-patterns - the problem

Changes during software development are inevitable [12]. As these changes accumulate, the code becomes large and complex causing undesirable changes in the original system design. This process, called design erosion can make maintenance of the code difficult and time consuming. Empirical studies have shown maintenance tasks can be as expensive as 75% of total costs of the system [13, 14]. To control maintenance costs, significant effort must be spent on keeping the code clean and the design in good shape, i.e. design erosions must be dealt with immediately!

A common occurrence of design erosion is in the form of blob anti-patterns [12]. A blob, also called a "god" class, is a single class containing multiple operations that monopolizes a portion of the system and uses other classes only as data holders. A blob may contain responsibilities that overlap with other system modules, hence, giving itself characteristics of a procedural design; further addition of functionalities to the blob will lead to more procedural code. These characteristics make blobs difficult to maintain and more error prone [6]. Hence, there is a need to identify and report the formation of such blobs, and to suggest corrective measures.

1.2. Extract Class Refactoring – a solution approach

Extract Class Refactoring (ECR) strives to rectify blob anti-patterns by delegating some of the responsibilities of a blob to new classes. Due to the high complexity of blobs, it might be difficult to perform ECR manually [6]. Therefore, there is a need to build automated refactoring tools to deal with blob anti-patterns. One drawback of dividing a class into multiple classes is that the new classes will have to pass messages between each other, whereas in the original blob

class, these messages were originally being passed between methods of the same class. This added coupling induces dependency among classes, i.e. changing a class individually might bring about an undesirable change in another. A desirable solution to this problem will be the one that divides a god class into several cohesive classes with minimum assistance from the user that keeps the coupling between the refactored classes to minimum. Several approaches to automate this process have been briefly introduced in the following section.

1.2.1. **Current solutions**

Different ideas based upon graph theory [9, 6], clustering algorithm [5], and game theory [4] have been employed to deal with the problem of ECR. These approaches strive to provide an automated solution with minimum user interaction but have crucial limitations like only being able to split a class into two classes [9], or requiring the user to predefine the number of classes to extract [4, 5].

The research approach suggested in [6] addresses some of these limitations, but has drawbacks of its own - the resulting refactored classes have less than optimal design with significant high coupling between each other.

1.2.2. **Proposed solution**

The proposed research advocates techniques to improve the current solutions by suggesting adaptations to one of the metrics used in [6], introducing a new metric, and significantly altering the manner in which the class extraction process is carried out.

In order to determine the strength of relationship between methods, three metrics are used in [6]. Our research proposes a modification to one of those metrics, known as Structural Similarity between Methods (SSM) [10]. SSM tries to build links between methods based on the number of

variables they share. Our approach modifies this metric to additionally analyze "how" a method uses the variables (i.e. reads it / writes it). The manner in which a variable is used holds important information that cannot be neglected while determining the strength of similarity between methods. This new metric is called Variable based Similarity between Methods (VSM).

We also introduce a new metric called Cognate Members Metric (CMM). This metric finds a set of instance variables that are used together in a set of limited methods, such that these methods do not use any other variable from outside the set; these variables along with the functions that use them are later extracted into separate classes.

Finally, we propose modifications to the class extraction procedure used in [6]. These modifications alters the process of allocating original class' variables to the refactored classes, modifies the way a method may access a variable belonging to another class, and groups completely unrelated methods of the class into a utility class.

This proposed approach significantly improves the coupling and cohesion of the refactored classes. It also improves the overall design of the system and has some positive effects on the complexity of the final classes. A detailed discussion on this claim is provided in Chapter 5.

1.3. **Work done**

Following is a brief list of the work that was responsible for the inception, development and conclusion of this thesis.

Literature survey

Broadly, the following areas of research in the field of software refactoring are substantially studied, and will be discussed in Chapter 2.

- Software refactoring

- Measuring complexity, coupling and cohesion of classes
- Extract Class Refactoring

Development of a prototype Refactoring Tool

In order to substantiate the claims made in this research, a prototype that performs ECR using VSM, SSM and CMM schemes was developed and used to analyze and compare their results. A list of the system modules used in the prototype are presented below and elaborated in Chapter 4.

- Java code inspectors
- Metric (VSM/SSM/CMM) calculating modules
- GUI
- Creating new classes
- Code refactoring module

Case studies

The classes refactored using the SSM and VSM approaches were compared using complexity, cohesiveness, and coupling metrics (Appendix A2), as well as a design analysis. A similar analysis was performed to establish the importance of the CMM metric. The detailed findings are recorded in Chapter 5 using a few case studies.

The study of results on refactoring of classes in our case studies let to improvements in our approach to help produce better results.

1.4. Thesis organization

In chapter 2, we briefly review the current research in the field of ECR, their drawbacks and limitations. Furthermore, we discuss current software metrics to analyze the quality of a class. In

Chapter 3, we discuss our extensions to the current state-of-the-art of ECR. We introduce modifications to one of the metric used during the refactoring process (VSM), discuss another metric (CMM) that can be used during ECR and propose significant alterations to the manner in which the class extraction is carried out.

In Chapter 4, we present the design of our prototype, refactoring tool. We describe the subsystems making up the tool and discuss design challenges faced during the development of the prototype. The steps taken to ensure logical equivalency of the final refactored classes with the blob class are also described. In Chapter 5, we provide a detailed comparative and analytical study on the outputs from different schemes (SSM, VSM, and CMM) on three randomly selected case studies. Finally, in Chapter 6, we present our conclusions and future work.

CHAPTER 2: LITERATURE REVIEW

This chapter presents the related works in the field of refactoring in general and more specifically, ECR. We discuss the importance of refactoring, various types and methods of performing automated refactoring, class quality metrics, and finally, the research in the field of Extract Class Refactoring.

2.1. Software refactoring

As discussed before, changes during software development life cycles can lead to design erosion, which may consequently result in high maintenance costs; hence, there is a growing need to tackle software complexities to decrease the tedium and cost of software maintenance. Frequent software refactoring can help decrease code complexity and improve quality of code, which is why it is a crucial field of study in software development. Formally defining refactoring, *it is the process of improving the design of a program without influencing any logical or external behavior*. The idea is to redistribute functions, variables and classes in order to help in future adaptations [1].

Refactoring can be broadly divided into two categories [1], Pattern Matching Techniques (PMT) and Meta-modeling Techniques (MMT). PMT is the use of shell-based regular expression tools and semantic analyzers for the compiler to refactor the code. MMT usually deals with meta-model transformations. Meta-model transformations require working with compiler parsers to build a model, to check preconditions, and to validate behavioral properties. This thesis focuses on refactoring based on MMT.

Another classification of refactoring is Floss refactoring and Root canal refactoring [2]. Floss based refactoring is to refactor the code whenever adding some feature to the software and aids

in keeping the code simple and organized. Root canal refactoring is the process of improving a program that has deteriorated over a long period of time and is usually tedious, protracted, difficult to execute manually, and very error prone. This research project focuses on the idea of automated root canal refactoring.

Automated software refactoring compared to manual refactoring is less error prone and more reliable; therefore, there is an imperative need to study and understand automated refactoring techniques. Following are the three phases of refactoring that an automated refactoring tool must implement [15]

- Identifying when to refactor
- Analyzing which refactoring techniques are to be applied
- Applying the refactoring

Fully automated tools complete all the three phases of refactoring without any user interaction. The tool developed in this research is semi-automated and requires developer's perception in selecting the most optimal refactoring. One reason for our approach is that developers typically do not prefer using fully automated tools for refactoring [2] due to the uncertainty of the final design of the product.

Research has shown that refactoring tools are under-used [16]. One of the reasons for this is that the developers usually fail to realize that they are going to refactor before they manually start doing it while trying to fix some bugs. In [16], a tool is developed that detects and informs developers when they start to manually refactor and suggests automated refactoring tools for the same. It recognizes some common patterns for manual refactoring and triggers a message for the developer when the same patterns are being manually retraced. One future prospect of this thesis

could be to extend the usage of the proposed tool to also alert developers whenever it is a good time to use the tool due to detection of blobs.

In [18], researchers concentrate on the reasons for the under-usage of automated refactoring tools: *need, awareness, naming, trust, predictability, and configuration.* All of these issues have either been resolved in this research or their respective solutions have been suggested for future research - see Appendix A1.

There has been substantial progress in the field of automated refactoring in the past few years; nevertheless, the field of study offers numerous scope for improvements.

2.2. **Software Metrics**

Software metrics are a good measure for the quality of classes. They help in deciding if a class complies with Object Oriented (OO) standards or if it requires refactoring. A good system design as defined in [10] is one that distributes the functionalities appropriately to its sub-components. A fitting distribution of functions can be measured by two key concepts: coupling and cohesion. Coupling measures how classes are connected to one another. Too strong of a connection would mean too much dependency of one class on the other with changes to one leading to changes to the other. Cohesion is the measure of the connection between methods in a class. A loose connection would mean that the functions in the class are unrelated and the functionalities have been poorly distributed.

A good class also has less complexity and is easy to understand and use. Software metrics help us establish the quality of a class. Appendix A2 lists these metrics, some of which we use in our research.

The metrics used for ECR in [6] are different from the ones listed in Appendix A2. These metrics, as described below, find relations between methods (structural and semantic). Strongly related methods are then extracted to separate classes.

2.2.1. **Structural Similarity between Methods (SSM) [10]**

As the name suggests SSM measures the degree of structural similarity between classes. Methods are said to be similar if they share overlapping instance variables. [10] describes two variations of this metric; they use the metric for calculating direct (SimD) and transitive (SimT) similarity between the pair of methods. They define these metrics as follows:

SimD(i,j), the measure of direct structural similarity between the methods Mi and Mj can be calculated by the following formula:

$$SimD\ (i, j) = \frac{|V_i \cap V_j|}{|V_i \cup V_j|}$$

SimT(i, j, π), the measure of transitive structural similarity between two methods due to a specific path, π, can be calculated as follows:

$$SimT\ (i, j, \pi) = \prod_{e_{s,t} \in \pi} SimD\ (s, t) = \prod_{e_{s,t} \in \pi} \frac{|V_s \cap V_t|}{|V_s \cup V_t|}$$

Here, V_x is the set of variables used by method M_x.

2.2.2. **Call based dependence between Methods (CDM) [9]**

The calls made between methods of a class hold a piece of information that can be exploited to calculate the cohesion of the class. Let Calls (m_i, m_j) be the number of calls made by method$_i$

to method$_j$. And let Calls$_{in}$(m_j) be the number of calls made to the method m_j. Then, CDM (m_i, m_j) can be defined as:

$$CDM_{i\to j} = \begin{cases} \dfrac{calls(m_i, m_j)}{calls_{in}(m_j)} & \text{if } calls_{in}(m_j) \neq 0; \\ 0 & \text{otherwise.} \end{cases}$$

2.2.3. **Conceptual Similarity between methods (CSM)** *[11]*

CSM is based on Latent Semantic Indexing (LSI). LSI captures significant portions of the meanings for words, sentences and passages. It can compute the similarities between sentences or paragraphs or even documents. LSI is adapted in [11] to be used to predict the conceptual similarity between methods (CSM).

2.3. **Extract Class Refactoring**

Sometimes while writing code, programmers tend to put most of the functionalities for a part of the software in a single class. These classes monopolize on most of the functionalities and other classes are basically used for only encapsulating data [19]. Such a design anti-patterns is called a "blob". The Blob is usually a result of an iterative development where the developer gradually adds a few functionalities to the class feeling that a separate class for them is not required. As these new functionalities grow, the class's quality declines and the class also becomes more complex [12].

A pioneer in the field of software refactoring, Martin Fowler, describes how a class must only be responsible for a few clear responsibilities, and offers some similar guidelines [12] for good design. A blob, failing to fall into this category is to be divided into smaller, more cohesive classes. It is also advisable to keep the coupling between these newly created classes to the minimum.

[9] uses max-flow min-cut theorem [28] to divide the blob into two separate classes. It first builds a graph with all the methods of the class as its vertices and an edge between the methods representing the semantic and structural similarity between the two. Next, max-flow min-cut theorem divides the graph into two disconnected subgraphs and makes sure minimum possible information (min cut) flows from one subgraph to the other. This also ensures minimal coupling between the two new classes. The drawback to this approach is that only two refactored classes can be formed.

[5] uses the Hierarchical Clustering Algorithm to divide the blob into a predefined number of classes. The idea of hierarchical clustering is to group together all the nodes that are similar to each other in a cluster. Each cluster will finally correspond to a class. As the non-similar clusters have minimal similarity, the coupling between the final classes is also minimal. One problem with this approach is that it is difficult to predict the number of output classes that would be needed. Solution by trial and error is tedious and untenable.

[4] proposes performing class extraction based upon the concept of game theory. It proposes how Nash Equilibrium can be used to calculate the optimal compromise between coupling and cohesion of the final extracted classes. Nash equilibrium is a solution strategy for non-cooperative games between two or more players. Moreover, it is the strategic point(s) where no player has anything to gain by changing their strategy unilaterally [21]. What this means for the refactoring process is that the final classes will reach an equilibrium point such that no class can increase its cohesion or decreases its coupling with other classes by redistributing its methods. The number of refactored classes to divide the blob into has to be predefined, which, as mentioned earlier, is a significant limitation.

Similarity between methods of a class is a good measure for cohesiveness of the class. [6] uses SSM, CSM, and CDM to calculate similarities between methods. A graph is built with nodes representing the class methods and edges between two methods representing their quantitative similarity. Edges between weakly similar methods are filtered out with the help of a threshold (minCoupling). In this graph, method chains having total number of nodes less than a threshold, minLength, are called trivial chains. Chains with nodes more than minLength are called non-trivial chains. Similarity between every trivial and non-trivial chain is evaluated by averaging the similarity (based on SSM, CSM, and CDM) for every pair of methods from them and is used to merge trivial chains to the most similar non-trivial chain. Next, every chain is matched up to a separate class where methods in the chain end up as methods in the class. In case of a conflict between classes over a variable, it is allotted to the class that uses it in more methods. If two or more classes use the variable in equal number of methods, it is allotted to the smaller class (having fewer methods).

The approach introduced in [6] as discussed above, is a good model that resolves many of the problems encountered with the other approaches. However, the research in this field of ECR is fairly new and has plenty of scope for improvement. Tools like ARIES [6] and JDEODORANT [5] are available to perform ECR but there is still a scope of improvement. Our research aims at further optimizing the field of ECR. Chapter 3 describes our proposed approach.

CHAPTER 3: SOLUTION APPROACH

In this chapter, we present details of our proposal for improvements to the existing Extract Class Refactoring process: a) addition of our VSM metric (a modification of the existing SSM metric), b) addition of our CMM metric, and c) modifications to the current class extraction procedure. We also list some of the limitations of our approach.

3.1. Variable based Similarity between Methods (VSM)

In this section we describe the VSM metric and differentiate it with the SSM metric (discussed in Section 2.3.1). We also explain in detail how to calculate this metric.

3.1.1. Description of VSM

SSM, a popular metric in the field of ECR, measures structural similarity between methods based on the number of *instance variables* shared among them. Closely related methods are then grouped into a class of their own. VSM modifies SSM to additionally analyze how the variables are *used* in methods as an additional factor in deciding how to group methods into classes.

A method can use a variable in the following two ways:

- **Write/ Update:** A method is said to write or update a variable if it either directly changes its value or one of the methods in its call tree does so.
- **Read Only:** If a method accesses a variable, but does not update its value, it is said to read it.

While deciding the final class for a variable during the refactoring process, it is preferable to allocate the variable to the class that has functions updating it, rather than a class having

functions that read it. Otherwise, every time that the updating class needs to write to the variable, it will need to use a getter and setter method for the variable. VSM adjusts the similarity between methods so that methods that use a variable in a similar fashion (both reading or both writing) are considered more similar than methods using the variable in a dissimilar manner. Let us look at an example to understand the difference in the functioning of the SSM and VSM based approaches.

Example of SSM and VSM based ECR

Suppose, we have a class with three methods: M1, M2, and M3. Let's assume M1 and M2 updates and M3 reads two variables x1 and x2. Furthermore, suppose M1 updates another variable, x3, which M3 only reads. As per the formula explained in the subsequent section, M1 and M3 will have the strongest SSM based similarity metric, whereas M1 and M2 will have the strongest similarity using VSM metric.

Before selecting the final refactored class design, users can adjust a threshold which will filter out weak similarities. If we filter out all the similarities except the strongest ones, while distributing these methods between classes, the SSM based approach will allocate M1 and M3 in a class whereas our VSM based approach will allocate M1 and M2 together in a class. Furthermore, the SSM based approach will allocate all three variables to the class with M1 and M3 functions. In this case, M2 will have to make setter and getter method calls to access x1 and x2 variables as they were not allocated to its class. On the other hand, our VSM based approach will allocate all the variables to the class having methods M1 and M2. Since M3 is only reading the variables, all three variables can be passed as arguments to it; this will result in reduced coupling, code complexity, and lines of code in comparison to the SSM based approach.

3.1.2. **How to calculate the VSM**

The following formula is used while calculating the similarity between two methods using VSM. After calculating the similarity for every pair of methods, the values are normalized to have a value between 0 (lowest) and 1 (highest) by calculating it as a percentile.

$$VSMij = \frac{\sum_{v \in Vi \cup Vj} vShare(v)}{total\ number\ of distinct\ variables\ used\ by\ Mi\ and\ Mj}$$

Where,

M_i and M_j are the two methods being analyzed,

V_i and V_j are the variable sets of methods M_i and M_j respectively,

$$\text{vShare(v)} = \begin{cases} 0.5, & if\ Mi\ updates\ v\ while\ Mj\ reads\ it\ or\ vice\ versa \\ 1.0, & otherwise \end{cases}$$

This definition is a modification from *SSM where vShare(v) is always equal to 1* for shared variables regardless of whether they are being read from or written to. VSM ensures that two methods reading a variable or two methods updating a variable have a higher similarity metric in comparison to similarity metric between methods where one method is reading a variable while the other method is updating it. A trivial explanation for this would be that two methods performing similar operations on a variable are more similar than two methods performing different operations. But a more appropriate reason for such a propensity is optimization. This increases the probability of methods that operate on a variable in a similar fashion, ending up in the same class. A class having methods that only read a variable can be deprived of it without increasing any coupling, whereas a class performing writes on the variable should have a higher chance of owning it.

Once the VSM is calculated between all methods, a graph for the god class is created. Every node in the graph represents a method from the class, and the weight of the edge between two methods represents the VSM between them. The user can adjust a threshold which defines a minimum possible VSM between two methods. If the VSM between two methods falls below the threshold, they are considered to have no similarity and can belong to different classes. In that case, the VSM can be considered to be equal to zero and the edge between the methods in the Graph can be removed.

Method chains having total number of nodes less than a threshold, minLength, are called *trivial chains*. Chains with nodes more than minLength are called *non-trivial chains*. Similarity between every trivial chain and non-trivial chain is evaluated by averaging the VSM for every pair of methods from the two chains and is used to merge trivial chains to the most similar non-trivial chain. This procedure was introduced in [20] and is also used in the SSM methodology. The default value for minLength is kept at three as advised in [20].

At this point, every chain of methods can be considered to form a separate class. When the user is satisfied with the proposed design as per the Graph, they can give a command to extract classes from the God Class and generate their code. The next step in the process of refactoring is to allocate the instance variables to the classes. This is discussed in Section 3.3.1.

VSM provides an opportunity to refactor the original class into classes that are less coupled and follow a better system design. Less coupling entail that the classes can be changed independently without affecting other classes. A better system design is always favorable and helps during the maintenance phase of the software.

3.2. **Cognate Members Metric (CMM)**

In this section we describe the CMM metric and explain its purpose and how to calculate this metric.

3.2.1. **Description of CMM**

We use the CMM to identify a set of class members (variables or methods) that are only used in combination with one other and seem to have no relation at all with other members of the class. In terms of VSM, VSM between a method from this set and a method from outside of the set will be 0, i.e. they will not share any variable. In other words, no variable belonging to this set will be accessed by a method from outside the set. As this set is completely exclusive from the whole class structure, it can be effectively and seamlessly segregated out to form a class of its own.

Merely using the VSM metric will not allow us to identify this set. This is because, when you change the VSM threshold, the CMM identified set will lose some of its edges and some of its members will not be a part of it anymore. Therefore, we propose to first identify all of these sets and separate them from the god class before continuing with further refactoring by using VSM.

Example of ECR using the CMM approach

Consider a class having *name, age, sex, ethnicity, countryCode, areaCode, and phoneNum* as some of its variables and *getName(), getEthnicity(), getCountryNameFromCountryCode(), getStateNameFromAreaCode(), getCompletePhoneNumber(), and isPhoneNumberValid()* as some of its methods. Quite intuitively it is easy to predict that the variables *countryCode, areaCode, and phoneNum* and the methods *getCountryNameFromCountryCode(), getStateNameFromAreaCode(), getCompletePhoneNumber(),and isPhoneNumberValid()* should be together in a separate class. This intuitive idea is captured by the CMM metric.

3.2.2. **How to calculate the CMM**

Consider a graph with the nodes in the graph representing instance variables of a class. If a method uses two class variables, an un-weighted bidirectional edge will be established between the two corresponding nodes of the graph. In such a graph, all disconnected sub-graphs will correspond to our desired sets. The variables from the set along with the methods using them together form a refactored class. The only restriction on the set is that it cannot have only one method and one variable in it. In case only one valid set is formed, there is no point refactoring the set as it clearly embodies most or all the class variables. In all other cases at least two sets will be formed, and the original class will hence be refactored into two separate classes.

CMM can be used as a filter to other methodologies. CMM is useful to help find the set of methods and variables that is isolated from the rest of the class. It can filter out these isolated parts of a class and the rest of the system can be used as input to other refactoring approaches.

3.3. **Modifications to class extraction procedure**

In addition to the VSM and CMM metrics, we introduce several modifications to the procedure used for extracting classes. These modifications are described in this section.

3.3.1. **Assigning variables to classes**

Sometimes methods that share a variable end up in different classes in the refactored design. In this unfavorable situation, we need to decide which class to assign the variable. Other researchers [6] have assigned such variables to the class that uses them in more methods. If two classes use the variable in equal number of methods, it is assigned to the smaller class.

We have modified this approach and apply the following formula to determine the affinity of a variable to be allocated to a class.

$$affinity(c, v) = \sum_{m \in Mc} affinity(m, v)$$

Where,

affinity(c,v) is the affinity of a class c for a variable v.

affinity(m,v) is the affinity of a method m for a variable v which is defined as follows:

$$affinity(m, v)$$

$$= \begin{cases} Wu * \left(1 + \frac{\left((v.numDirectUpdates) + (v.numReads)\right)}{14.0}\right), if\ m\ directly\ updates\ v \\ WT * \left(1 + \frac{v.numReads}{14.0}\right), if\ m\ transitively\ updates\ v \\ WR * \left(1 + \frac{v.numReads}{14.0}\right), if\ m\ reads\ v \\ 0, otherwise \end{cases}$$

Where,

W_u = 0.64,

W_T = 0.60,

W_R = 0.20,

v.numDirectUpdates = number of times m directly updates or writes to v

v.numReads = number of times m reads v

If affinity(m,v) exceeds 1 at any point it is made equal to 1.

A direct update is when a method changes the value of a variable in its body, whereas a transitive update is when one of the methods in its call tree changes the value of the variable.

The class with maximum affinity for a variable will finally "win" the variable and have it as one of its instance variables. Setters and getters for the variable, if not already present, will be created and used by other classes to access the variable.

Here is the intuition behind the formulae. A method that updates or writes to a variable v tends to have more affinity towards it than a method that performs transitive updates or reads it. This is because a method that needs to update a variable that is not in its class needs to call both getters and setters for the variable, whereas a method that performs transitive updates on the variable only needs to call the getter function. While accessing a variable that is read in the method, the variable can simply be passed as an argument to the function. Hence, to decrease the coupling between extracted classes, a variable is deemed to have more affinity towards the method that is updating it. Using the above logic, and empirically testing the system against different classes, the values for the weights ($W_{u,}$ $W_{T,}$ W_R) were determined.

The weights are then multiplied by a quantity determined by the frequency at which the variable is used in the method. For example a variable used 14 times in a function will make the affinity double the value of the weight associated with it.

Let us discuss two *implementations* of the formula – Suppose a method transitively updates a variable twice while another method writes it once. The affinity in both cases will be approximately equal (0.68). This derives from the fact that both methods will have to make equal number of method calls (getter/setter) to access the variable. However, a method that reads a variable even up to 33 times will still have affinity less than the method that writes the variable. This is because the variable can still be passed as an argument to the method reading it, thereby not introducing any coupling. The coefficient values and other numeric configurations in the formula were empirically established but can still be further adjusted in future research.

In case the affinity for a variable comes out to be equal for two or more classes, it is allotted to the class with lesser number of functions. Once the variables are allotted to respective classes,

the code is changed to maintain logical and syntactical accuracy. The details of the process followed during the implementation process are discussed in Chapter 4.

3.3.2. **Omitting variable setter and getter from the refactoring process**

Setters and getters of a variable are functions that are responsible for updating a value for the variable or retrieving the value of the variable. We have not used these methods during any diagnostic or analytical process during the course of this research. Furthermore, we do not use them while building method similarities during the VSM calculation phase and they are not factor in our decision to assign a variable to a class. These methods end up in the same class as the variable they operate upon. The rationale behind our approach is that these methods are simply an access route for the variable and do not contribute to the overall design of the class. Since the metric values are computed as percentiles, one metric value can subject a change on another. By using setters and getters in the refactoring process, we risk changing the dynamics of the whole process. Moreover, it was observed that many times a class was formed with only the variable and its setter and getter. Additionally, it is also possible, that the final variable is not allotted to the same class as the setters and getters. These are obviously extreme but undesirable cases.

After experimentally analyzing the cons of using setters and getters as normal functions, it was concluded not to use them as such. Some of the comparative studies between previous approaches (SSM) and the proposed approach (VSM) showed great improvements and some of this improvement can be attributed to the notion of ignoring setters and getters.

3.3.3. **Passing variables as an argument to a function**

Instead of using getter method calls, we chose to pass variables as an argument to the function trying to access it. This is only possible in the case the variable is read by the function. This approach decreases the number of getter calls and thereby decreases the coupling and complexity of the system.

3.3.4. **Utility Class**

Set of methods that are completely unrelated to all the other methods of the class can be extracted together into a Utility class. Because these methods don't share any variable with any other methods, they can be labeled as independent methods and can be segregated out. This utility class is very non cohesive. However, by creating such an Utility class, we make the overall design of the system better. As utility methods don't have any similarity with any other methods, if they were to be put in any other class, they would erode the design of that class.

3.4. **Limitations**

Here are some of the limitations and drawbacks of our proposed approach to ECR.

1. User perception is required to decide the *minCoupling* and *minLength* thresholds. Although we provide a GUI tool that makes it easier to make such a decision, it may still confuse a user, especially if they are not the designer of the system.

2. The design of the original system is modified in many ways. Private methods that are accessed outside the refactored classes need to be modified to have default data access. Also, final instance variables of a class that are defined during variable declarations, and use a non-final variable in their definition, need to be made non-final. This is because non-final data

members are declared in the constructor of the refactored classes, and hence, to maintain consistency, we must also define the final instance variables inside the constructor, but final instance variables cannot be defined inside a function or a constructor. Therefore, we need to make these members non-final.

3. Even though the CMM methodology proposes a good theoretical solution, it was tested against many classes but did not yield good results with most of them. This can be a deterrent to its usage. A tool that run in the background of an IDE and automatically finds and suggests clearly isolated sets in the classes while the developer is coding on the system will be helpful.

To study our proposal for improvements to the existing ECR process, a prototype-refactoring tool was built to automatically perform ECR and was tested against numerous blob anti-patterns. Chapter 4 describes the implementation challenges and design strategies that were followed to refactor the code while ensuring logical equivalency with the original class.

CHAPTER 4: DESIGN AND IMPLEMENTATION

To evaluate our approach (as discussed in chapter 3), we built a prototype tool that performs ECR using the three approaches VSM, SSM, and CMM. A discussion of the system architecture as well as implementation challenges faced during the prototype development, have been discussed in this chapter.

Refactoring results in changes to the existing code and we need to ensure that the changed code behaves in the same manner as the original code. Herein, we also attempt to explain how our transformations to the code preserve the behavior of the system. Finally, we present some of the limitations of the prototype.

4.1. System Architecture

In this section we discuss the major components that were developed during the implementation of the prototype. Figure 1 presents a system diagram depicting a block representation of these modules. Subsequent to which, is a discussion on their roles and development strategies.

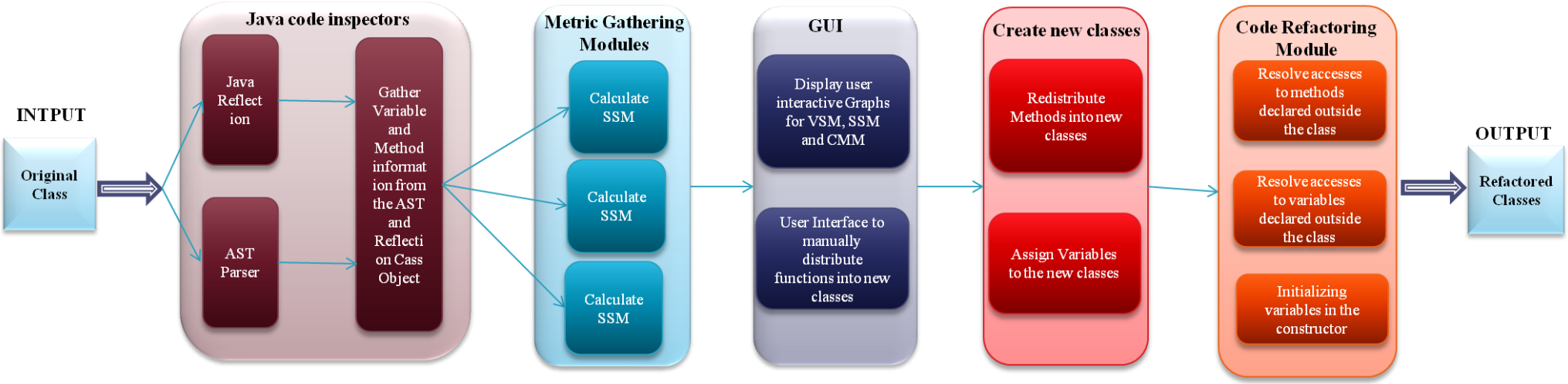


Figure 1: System diagram of the proposed refactoring tool

Java Code Inspectors

The foundation of this research is based upon analyzing variables; where and how they are used and with what frequency. This information cannot be retrieved using search-based query engines. Reflection and Abstract Syntax Trees (AST) parsing mechanisms are used in this research to serve this purpose.

Java Reflection

Reflection is the ability of a program to examine the structure and behavior of class objects at runtime. Java reflection API was used to inspect classes and gather the following information:

- Methods' name, arguments, return-type, modifiers and exceptions are captured; a class called "*SimpleMethods"* is designed to store this information
- Instance variables' name and type are captured; a class called "Simple*Variables"* is designed to store this information

AST Parser

ASTs are created during the code compilation process and provide a tree representation of the abstract syntactical structure of the program. We used one such open source tool called JavaParser [27] to generate an AST. JavaParser provides a batch of API calls to help the user analyze the AST, which has all sorts of information about the syntax and the structure of the program. The following information was gathered through the use of JavaParser:

- Which method is using a variable, how many times and is it reading it or writing it
- Method Call hierarchy
- Body of a method

Metric Gathering Modules

Using the formulas and methodology described in Section 3.1.2 and 3.2.2 VSM/SSM between every pair of methods and CMM between the instance variables were calculated.

Graphical User Interface (GUI)

Interactive graphs for VSM, SSM and CMM

A graphical user interface (GUI) was built to facilitate the handling of the tool. The GUI provided an illustration of the graph representing the original class. Every method in the graph represents a node, whereas a weighted edge between two nodes signifies the strength of the relation, in terms of VSM, between the respective methods. Methods with no edges are unrelated to each other. Users can adjust a minimum threshold, called *minCoupling*, for the edges such that an edge with a weight below this threshold is dissolved.

A tabular representation of the graph is also provided. Every cell in the table corresponds to the similarity between the methods represented by the corresponding row and column headers of the table. Cells with no value represent 0 similarities. The cell values are also dissolved upon changing the value of minCoupling.

Additionally, a panel is provided, which displays a list of the original class variables, and the methods using them. The exact same GUI is built for the SSM metric scheme, but the GUI for the CMM metric scheme has a different graph (discussed in Section 3.2.2).

User Interface to manually distribute functions into new classes

A GUI is also provided for the user to manually distribute functions in new classes. The tool will then automatically refactor the code to the specified design. This is a favorable case when the developer of the system is refactoring the code.

Creating New Classes

A class called "*RefactoredClass*" was created during the implementation of this tool. Once the user selects the final design of the system, an object of RefactoredClass is created for every new class; it holds the SimpleMethods objects and SimpleVariables objects. The distribution of these objects is discussed below.

Distributing methods to new classes

After filtering the SSM/ VSM based graph with the help of *minCoupling* threshold, *trivial chains* are grouped with the most similar *non-trivial chain* (discussed in Section 3.1.2). Methods from every chain are distributed to new classes. For CMM, the variables belonging to new classes are decided first and the methods that use those variables are then distributed to the respective class.

An object of SimpleMethods class is created for each method and distributed to the respective object of RefactoredClass.

Distributing variables to new classes

Variables in the case of VSM/SSM techniques are distributed to the classes using the formulas suggested in Section 3.3.1. In case of CMM, the variables are decided using the methodology suggested in Section 3.2.2. A SimpleVariables class object is created and added to the respective RefactoredClass object.

Code Refactoring Module

The code refactoring module is discussed in Section 4.4.

4.2. Using the tool

Let us discuss step-by-step, how a user can perform ECR on a class using our tool.

- Step I: Update the variables - packageName, className and path in the file Config.java. By keeping ClassName = "*", the ECR tool will be run on the whole package.
- Step II: Run the project. A separate GUI will be presented for every approach (SSM/ VSM/ CMM). Figure 2 displays a sample GUI for a class with 5 functions. The nodes in the graph in the figure are the methods of the class and the edges between two nodes represent the magnitude of similarity between them.

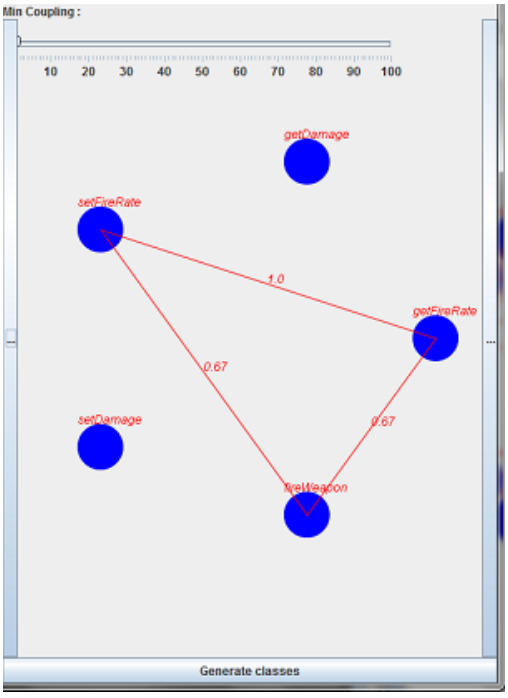


Figure 2: Sample GUI for ECR using the proposed tool with unfiltered edges.

The graph can become difficult to understand once the number of methods in the class increases. This is why a table displaying similarities between every pair of method is also displayed. This can be viewed by clicking to the button on the right side of the above GUI in Figure 2. Figure 3 displays a sample table.

Fn()	getFireRate	fireWeapon	setDamage	setFireRate	getDamage
getFireRate		0.67		1.0	
fireWeapon	0.67			0.67	
setDamage					
setFireRate	1.0	0.67			
getDamage					

Figure 3: Sample table depicting similarities between methods.

- Step III: Adjust minCoupling to filter out unwanted edges. Figure 4 exhibits the filtered sample graph from Figure 2, with minCoupling = 0.68.

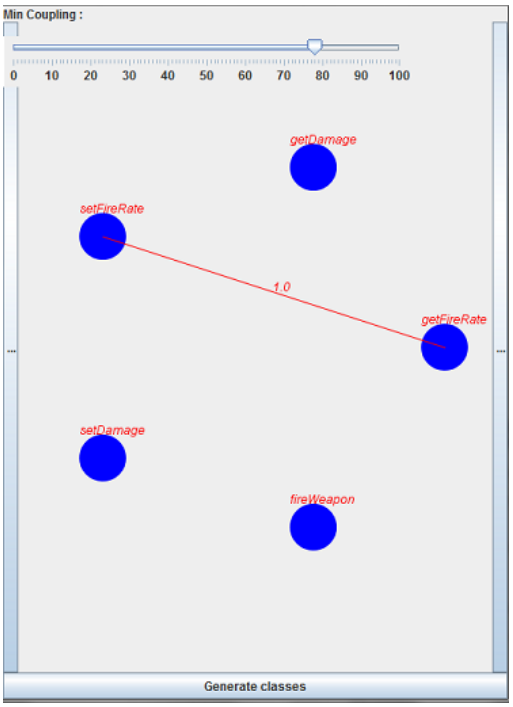


Figure 4: Sample GUI for ECR using the proposed tool with filtered edges.

- Step IV: The user can click on “generate classes” button and the tool will distribute the functions and instance variables to new classes, refactor them and finally print them in .java files.

  Additionally, the users can also manually select method distribution in new classes and the tool will refactor the system to the proposed design. This can be carried out by clicking on the left most button in the above generated UI (Figure 2), and then adding classes and distributing methods to them. Figure 5 shows how this can be done.

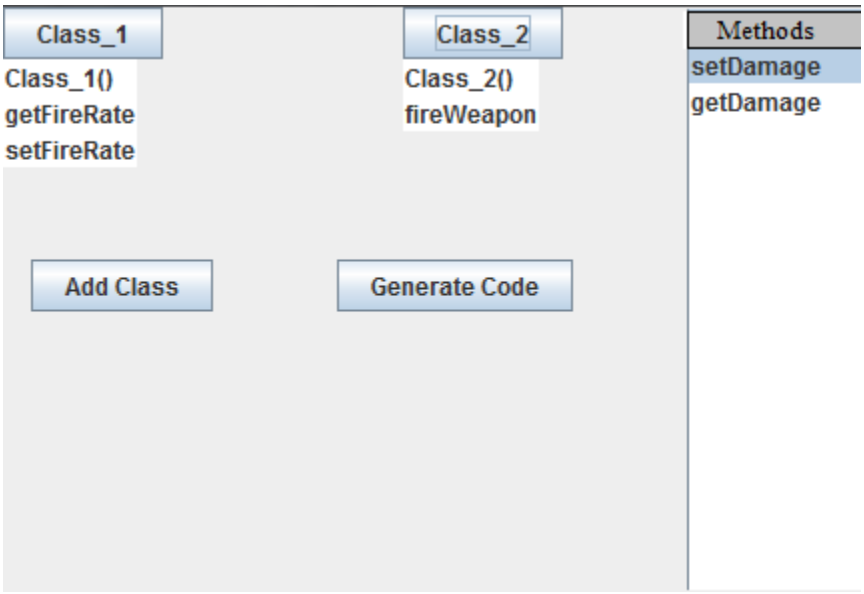


Figure 5: Manually distributing methods into new classes for a sample class.

To add a new class, the button “Add Class” can be clicked upon. The selected method on the right panel in Figure 5 will be distributed to the class whose button (Class_1 or Class_2) is pressed.

- Step V: After clicking the “Generate Classes” button, use the refactored files by replacing the original file with them in the project.

4.3. **Implementation Challenges**

We will now discuss the problems that were faced during the implementation phase of the ECR tool and how they were resolved.

Empirically testing the formulas used while refactoring

An essential motive for building this tool was to empirically verify the formulas against different test cases and adjust them using the feedback from the tests. The output classes for the test cases were evaluated via software metrics and class design analysis. Unsatisfactory results lead to adjustments in the VSM and variable distribution formulas (Section 3.1.2 and 3.3.1). This process was repeated several times before settling upon values used in the formulas.

Refactoring the new classes

Several iterations of code testing and code redesigning had to be undertaken during the refactoring procedure (Section 4.4) to ensure that the refactored code complies with every Java compilation rule and maintain logical correctness. Sufficient testing has indicated that the tool will work with a subset of Java classes. Limitations of the tool are discussed in Section 4.5.

LCOM HS metric evaluation

Because none of the software metric tools used in this research could provide an apt LCOM HS (Appendix A2) software metric implementation, we developed it ourselves. In our implementation, we do not use getters and setters of a variable.

4.4. Refactoring the code while preserving the system behavior

After distributing the methods and variables in new classes, we need to adjust the code to maintain syntactical and logical correctness. Following code changes were implemented during the refactoring process:

4.4.1. Code changes related to class methods

Method Signature transformations

A method signature transformation is the modification of the signature of a method to add one or more parameters to it. Needless to say, all the calls to this method must be modified to accommodate the new parameters. A caller method may, however, not have accesses to the parameters that are needed. Thus, the signature of the caller method may also need to be transformed to accommodate the new parameters, and this must propagate until a method is reached that has access to the parameter. Consequently, many methods may need signature modifications.

Invoking a method belonging to a different class

To invoke a method that belongs to a different class, the class's object reference is needed. If the reference is not accessible to the caller method, a method signature transformation (discussed above) needs to be performed to have the reference passed as a parameter to it. Once the reference is accessible, the method can be called using the dot operator, for example – classObject.method().

4.4.2. **Code changes related to instance variables**

Using a variable belonging to a different class

A variable can be made accessible inside a method by passing it as an argument to the method (method signature transformation), but this should only be done if the method neither writes the variable it in its body, nor calls a method in its method call tree that writes to it.

If a method writes a variable in its body, setter and getter method calls are made in lieu of every write statement, whereas a getter method call is made in lieu of a read statement.

In case a variable is only read inside a method, but is written in its method call tree, the variable must be accessed using its getter method. We cannot use the alternative approach of storing the variable's value in a temporary variable and using the temporary variable wherever the variable is being used, as it can lead to inconsistency since the variable's value might change inside one of the method calls before the temporary copy is used.

Initializing variables in the constructor

Instance variables in the original class are initialized either in the constructor or during declaration. When the class gets split into multiple classes, the variable initializations in the split classes will have to be performed in the same sequential order as it was in the original class.

We tackle this problem by initializing all the variables in the constructor of one of the final classes (called Main-class). All other classes initialize their instance variables to a dummy value while declaring it (to avoid "variable not declared" compilation error). The constructor for Main-class initializes the variables in the same order as in the original class. It uses setters and getters to access variables from other classes (unless they are public).

4.4.3. **Code changes related to static data members**

Static data members of a class should be refactored in a different manner as compared to the non-static data members. This is because of the following reasons:

- Static data members should not be initialized in the constructor of a class. Furthermore, non-static variable cannot be used in the definition of static variables.
- Static methods and variables should be accessed only via the class name (not via an object).
- Static methods cannot access non-static data members.

To address these issues, the system was designed with abstract super-classes having a set of subclasses responsible for refactoring non-static data members, and another set for refactoring static data members.

The tool has been tested against several god classes and has given positive results. No logical or syntactical bugs were found in these test cases. It is important to note however that there are a few limitations where the tool may fail to function as listed in the following section.

4.5. **Limitations**

- The tool fails to maintain logical accuracy with threaded systems.
- The tool currently does not support classes that extend other classes or implement interfaces.
- The tool does not resolve the method calls and class variable accesses from outside classes in the project. At the moment these need to be resolved manually.

The refactoring tool provides a foundation of an analytic study done on the outputs using the three schemes (SSM, VSM, and CMM). This analytic study is carried out with the help of

software quality metrics discussed in Appendix A-2 and a class level design analysis. The following chapter gives insights into the findings and outcomes of the study.

CHAPTER 5: RESULTS AND OBSERVATIONS

In this chapter, we present the results of a comparative study between the SSM and VSM based approaches for Extract Class Refactoring on two case studies. We also present a third case study of Extract Class Refactoring using our CMM metric. The refactored classes are analyzed via several tools including “JArchitecture" a Java code analyzing software [23], STAN4J, structural analysis for Java [24], and Source code Metrics 1.3.6 Plugin for Eclipse [25]. Additionally, a design analyses on the refactor classes is presented.

5.1. Comparative Study

In this section, we provide a comparison of refactoring based on the SSM and VSM metrics by applying ECR on two case studies. For each case study, we first describe the case study and the original blob class. Next, we evaluate the values of the two metrics for the different methods of the class. Then, we use the metric values to refactor the blob class. Finally, we compare the quality of the refactored code using the two metrics. We looked at many different metrics; however, we moved the description and explanation of each metric to Appendix A1.

5.1.1. Case Study I (BackPropagation)

Description

This case study focuses on performing ECR on a class titled BackPropagation. This class creates and trains a backpropagation neural network. Some of its functionalities are defining and normalizing network input parameters, training and building a neural network, and performing computation operations while training the network. Manual inspection of this code reveals that it

is a blob class and is a blend of functionalities that can be isolated from each other. Although the methods share many instance variables with one another, the overall design of the code is poor and can be improved upon by refactoring. Let us now discuss the refactoring performed using the two approaches.

Evaluating the Structural Similarity between methods

The first step in refactoring is to evaluate the structural similarity between methods in the class. SSM and VSM metrics can both be used to calculate the structural similarity between methods. Manual experimentation with values of minCoupling and minLength can help to filter out the weaker relations between methods (and help in the refactoring process). After careful analysis of the refactored designs with numerous values for minCoupling, it was set to 0.72 in case of SSM and 0.69 in case of VSM. These values were empirically verified to give best system designs for respective approaches. Meanwhile, minLength is kept at 2 for both the approaches. This is because the original class is small in size (number of methods) and does not form large chains of methods. By keeping minLength = 3, most chains were merging with the bigger chains yielding ineffective results. For sake of completeness, we present the unfiltered data in Appendix B1 and B2. Appendix B1 depicts the unfiltered structural similarities between every pair of methods using the two approaches. Weak relations between methods are filtered out using minCoupling and the filtered structural similarities between the methods are given in Appendix B2.

Figure 6 and 7 displays the SSM/VSM based similarities (after filtering) between strongly connected function sets of the class. Every node represents a function set, with the methods written as text inside the circular nodes. The weight of the edge between two sets represents the

average value of SSM/VSM between the methods of each set. The chosen value for minCoupling filters out all the edges in these figures, forming disconnected functional sets and provides an idea on how to refactor the blob class. Similarities between the functions inside the sets are higher than minCoupling, and hence don't get filtered out. Trivial sets (defined in Section 3.1.2) are represented by smaller circles and are merged with non-trivial sets (bigger circles) of the same color. These merged sets form a single class.

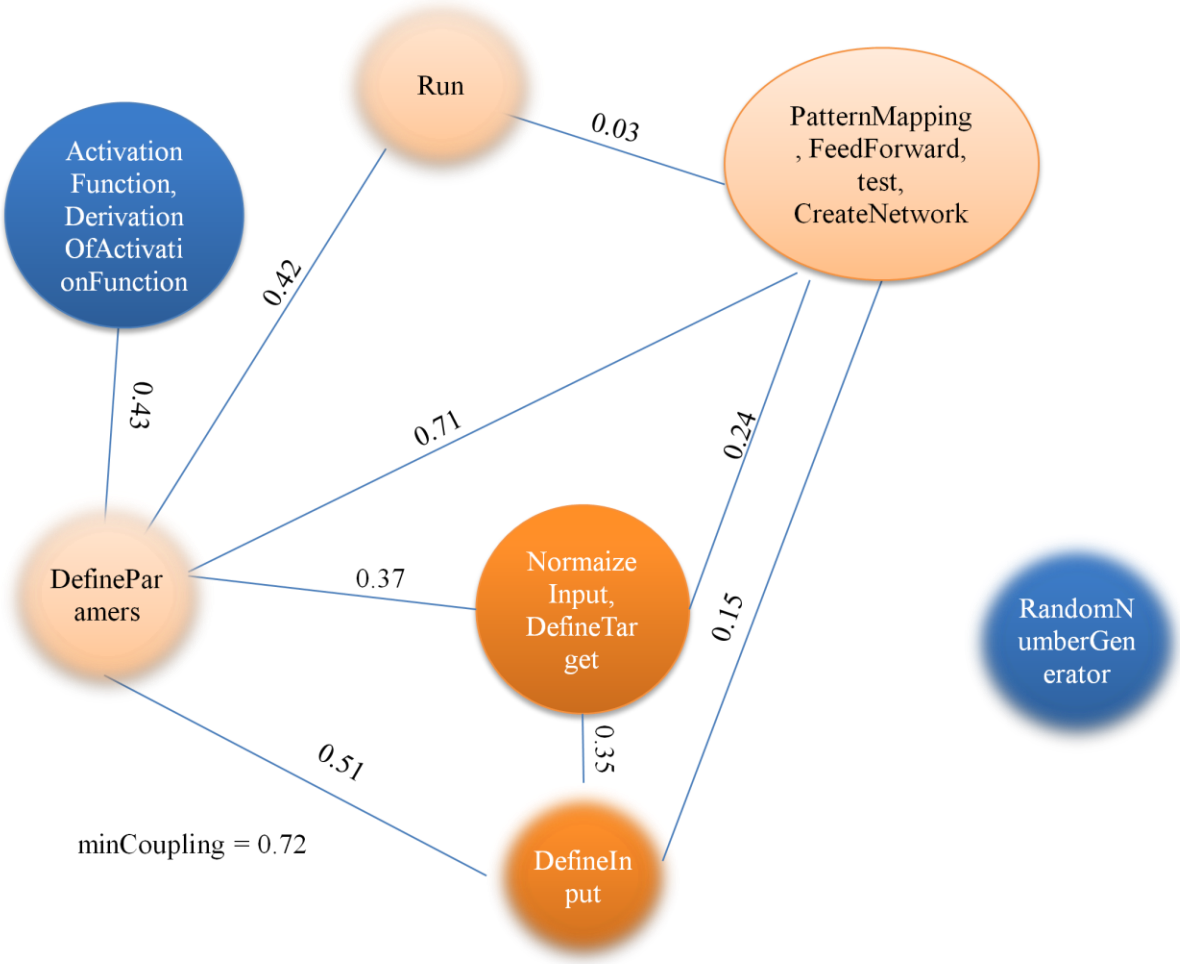


Figure 6: Strongly related (SSM based) functional sets and similarities between them, Case Study I. Nodes represent set of closely related methods; weight of the edge between two nodes represents the average value of SSM among the methods of the corresponding sets. Non trivial

sets are represented by smaller circles and are merged with trivial sets (bigger circles) of the same color.

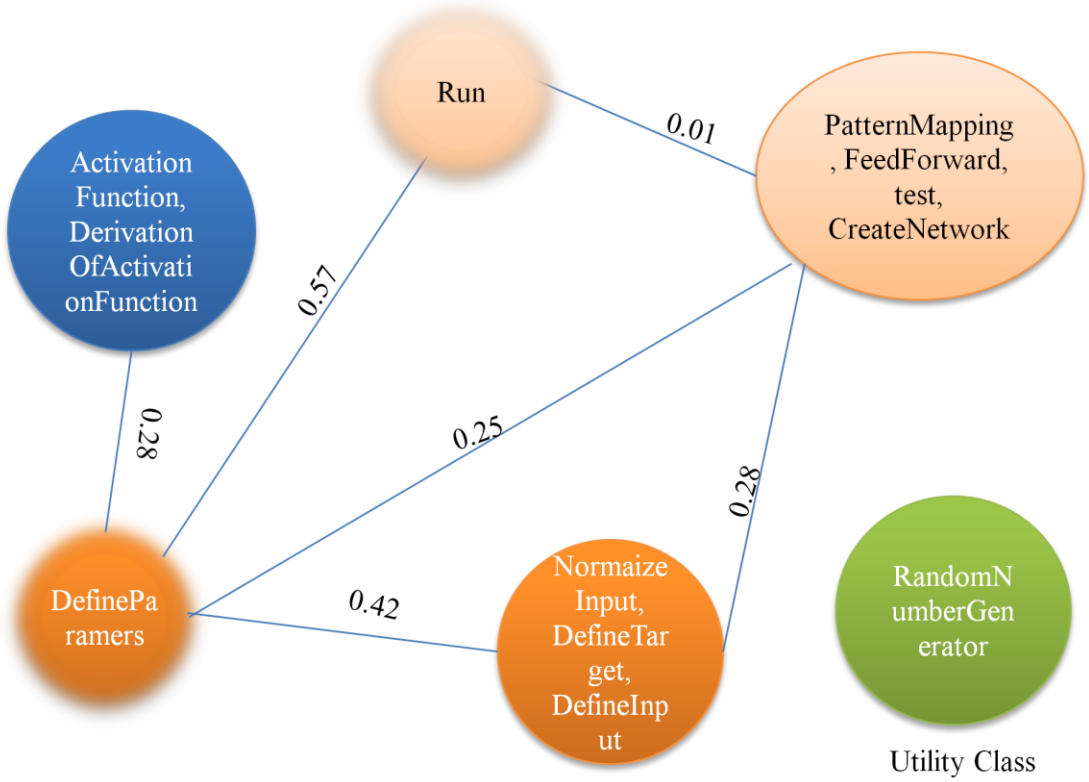


Figure 7 : Strongly related (VSM based) functional sets and similarities between them, Case Study I. Format of the figure is similar to Figure 6.

Refactoring the blob Class

Following is a list of functions in their respective refactored classes for the SSM based approach.

Table 1: Functional distribution in refactored classes, case study I (SSM)

Class Name	Function List
Class1	test, FeedForward, PatternMapping, CreateNetworks, Run, DefineParameters, main
Class2	DefineTarget, NormalizeInput, DefineInput
Class3	DerivationOfActivationFunction, ActivationFunction, RandomNumberGenerator

Following is a list of functions in their respective refactored classes for the VSM based approach.

Table 2 : Functional distribution in refactored classes, case study I (VSM)

Class Name	Function List
Class1	test, FeedForward, PatternMapping, CreateNetworks, Run, main
Class2	DefineTarget, NormalizeInput, DefineInput, DefineParameters
Class3	DerivationOfActivationFunction, ActivationFunction
Utility Class	RandomNumberGenerator

Comparison of refactored classes

A metric and design analysis is provided for the original class and the classes refactored using both the approaches. For comparison purposes, the data is presented side by side.

Table 3: Metrics analysis of original and refactored, Case Study I

Type of Metric	Metric Name	OriginalClass	Class1		Class2		Class3		Utility
			SSM	VSM	SSM	VSM	SSM	VSM	VSM
Countable Metrics	LOC	725	291	282	778	331	46	39	8
	M	13	7	6	3	4	3	2	1
	F	31	25	16	2	12	1	0	0
Cohesion	LCOM	0.72	0.38	0.21	0.75	0.86	0.5	0.0	0.0
Coupling	CBO	0	2	3	1	1	1	1	1
Complexity	WMC	101	69	68	24	23	10	9	2
	FAT	97	90	59	5	29	4	1	0

Very high metric value Moderate to high metric value

Low metric value Metric value leads to no conclusion about code quality.

The original class has high values (marked in red color) of LOC, M and F, which contribute to its large complexities as indicated by FAT and WMC. Design analysis of the class shows that the class has three elementary sets of functionalities, i.e. defining and normalizing network input parameters, training and building a neural network, and performing computation operations while training the network. Based on the metric and design analysis, the refactored classes are definitely better than the original class in terms of cohesion as observed from the LCOM values and complexity as observed from FAT and WMC.

Compared to the SSM based approach, the complexity metric FAT for the VSM based approach is significantly lowered for Class1 implying that the class is less complex. Even though FAT for Class2 has increased for the VSM approach, the class complexity is still very low, and the class is well within the limits of good design. For both cases, class1 is coupled to all the other

classes in the system, which makes it dependent on them. The metric Message Passing Coupling (MPC) (defined in Appendix A2), which is not computed in this project due to lack of available tools, will be much higher for Class2 formed using the SSM based approach in comparison to its counterpart formed using the VSM approach. For example, a particular variable was modified 224 times in a function of Class2, but the SSM methodology assigns the variable to Class1 because there it is used in two functions. This increases the method calls to the setter and getter functions for the variable, consequently increasing it's MPC. This also increases the LOC in Class2 formed using SSM based approach. This problem is rectified in the VSM based approach.

The LCOM values for the refactored classes based on both approaches are similar. Low LCOM values for Class1, Class3, and the Utility Class indicate good cohesiveness in the classes. This entails a good class design. It is important to note that a high LCOM value for a class (Class2) is not conclusive of low cohesiveness in the class [26]. Design analysis of the classes provides a better estimate. Table 4 lists down each of the refactored class and its high level functionality. A direct comparison between the design of the two systems can be made using the table.

Table 4: Design analysis of refactored classes, case study I

Class Name	SSM based approach	VSM based approach
Class1	Contains all the neural network training and testing functionalities, also includes a function that defines network parameters	Contains neural network training and testing functionalities

Table 4 (continued)

Class Name	SSM based approach	VSM based approach
Class2	Contains most functions that define and modify network parameters. Most of these parameters are declared in Class1; therefore, the class is poorly designed.	Contains all the functions that define and modify network parameters; all of which are declared in the same class.
Class3	Provides computational functionality related to "Activation Function". Also provides a helper function.	Provides computational functionality related to Activation Function
Utility Class	N/A	Provides a single helper function.

The assignment of variables in the case of Class2 formed using the SSM based approach is not done satisfactorily. Most of the variables being processed in the class are declared outside of the class, thus, increasing the complexity and coupling of the code. However, the assignment of variables in the case of Class2 formed using the VSM based approach rectifies this issue.

Conclusion

Table 4 shows that the classes formed using the VSM based approach have more closely related functionalities. The overall analysis of the two approaches show an improvement in cohesiveness and complexity of the overall design of the refactored classes using the VSM based approach in comparison to the SSM one.

5.1.2. **Case Study II (MechaBattle)**

Description

This case study focuses on refactoring one of the classes in a group project called MechaBattle that builds a multiplayer strategic card game that follows a client server model. The class MechaBattle.mechaGui.BattleScreen basically provides a design for the main screen of the game. The class has many sets of isolated functionalities like producing graphical view, managing game players and cards etc. These functionalities can be divided into different classes.

Evaluating the Structural Similarity between methods

As in case study-1, the first step in refactoring is to evaluate the structural similarity between methods in the class using SSM and VSM metrics. Again, as before, manual experimentation with values of minCoupling and minLength can help to filter out the weaker relations between methods (and help in the refactoring process). After careful analysis of the refactored designs with numerous values for minCoupling, it was set to 0.84 in case of SSM and 0.69 in case of VSM. Meanwhile, minLength is kept at 3 for the SSM based approach and 2 for the VSM based approach. This is because with minLength = 2, some of the classes had only setter and getter methods. As discussed in Section 3.3.2 this is an unfavorable situation; thus, we make minLength = 3 to get more appropriate results. The problem of setter and getter methods is resolved in the refactoring approach proposed in this research (Section 3.3.2); therefore, minLength =2 is used for the VSM based approach.

For sake of completeness, we present the unfiltered data in Appendix C1 and C2. Appendix C1 depicts the unfiltered structural similarities between every pair of methods using the two

approaches. Weak relations between methods are filtered out using minCoupling and the filtered structural similarities between the methods are given in Appendix C2.

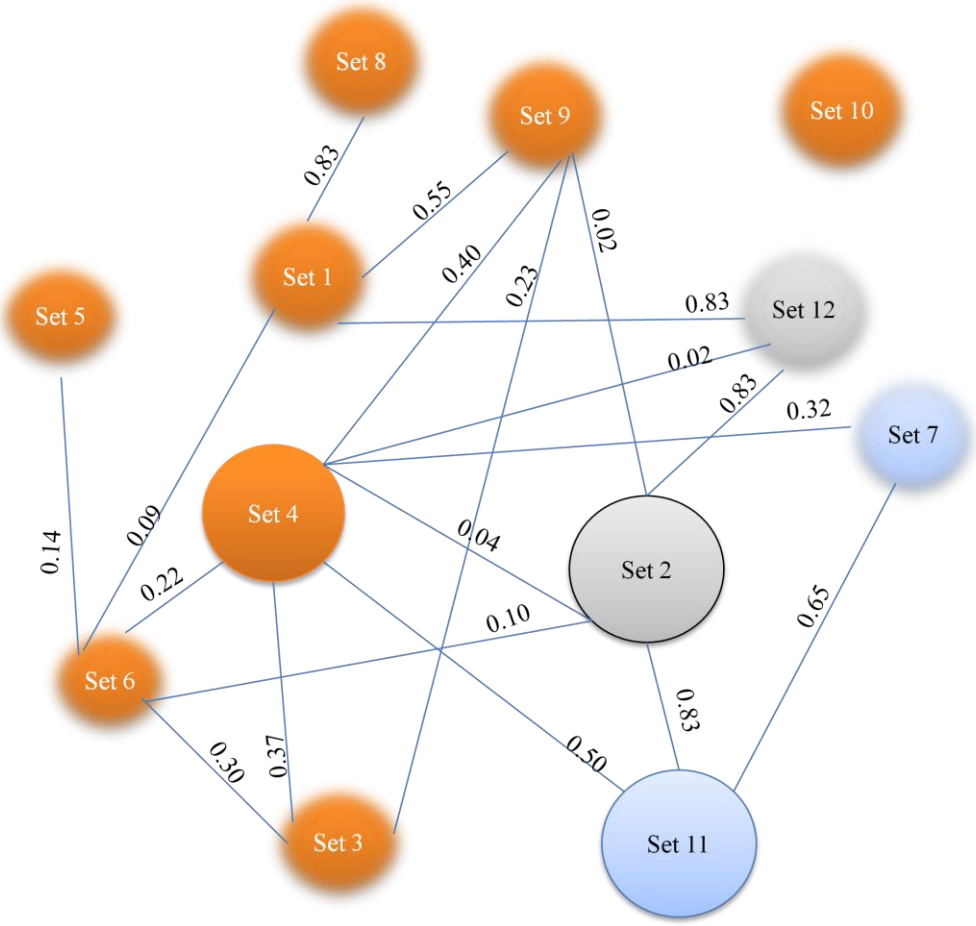


Figure 8 : Strongly related (SSM based) functional sets and similarities between them, Case Study II. Format of the figure is similar to Figure 6. Functions corresponding to each set are described in Table 5.

With the exception of the function sets defined exclusively from the figures, the format of Figures 8 and 9 are similar to Figure 6 and 7 which are described in case study-1.

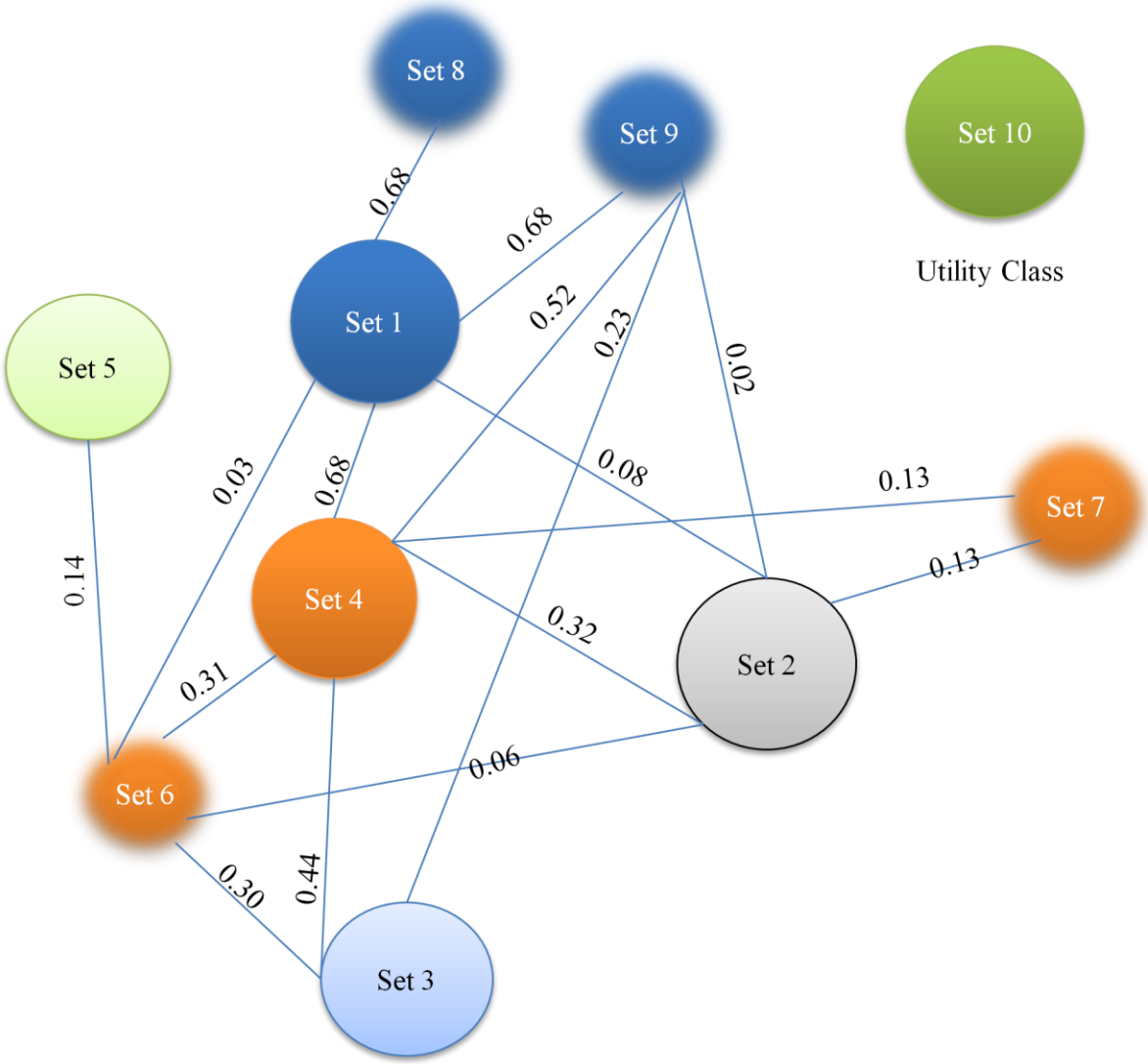


Figure 9 : Strongly related (VSM based) functional sets and similarities between them, Case Study II. Format of the figure is similar to Figure 6. Functions corresponding to each set are described in Table 5.

Following is a list of the function sets used in figure 8 and 9

Table 5: List of functions corresponding to the sets used in Figure 8 and 9

Set number	Functions inside the set (SSM)	Functions inside the set (VSM)
1	getCardAtHandLocation, setCardAtHandLocation	
2	getNumberOfCardsOnBattleField, setCardAtMainLocation, getCardAtMainLocation	getNumberOfCardsOnBattleField, generateTileNullsGrid, setCardAtMainLocation, battleFieldIsEmpty, getCardAtMainLocation, setBattleField, generateTileFieldsGrid, getBattleLabels
3	setPlayer2Turn, setPlayer1Turn	
4	labelsFromInts setBackgroundLabelBlueOrGreen, setBackgroundLabelRedOrYellow, findPlayer1Score, transcribeLabels, findPlayer2Score, capturedCards	
5	start15SecCountDown, start5SecCountDown	
6	createPanel	
7	initialLabelsFiller	
8	playerHandIsEmpty	
9	setRightCardTiles	
10	whoWon	
11	setBattleField, generateTileFieldsGrid, battleFieldIsEmpty	N/A
12	generateTileNullsGrid	N/A

Refactoring the blob Class

Following is a list of functions in their respective refactored classes for the SSM based approach.

5.1.1.1.a.1 *SSM Based Approach*

Table 6: Functional distribution in refactored classes, case study II (SSM)

Class Name	Function List
Class1	setBattleField, battleFieldIsEmpty, generateTileFieldsGrid, setMainLabels, getBattleLabels, initialLabelsFiller
Class2	labelsFromInts, findPlayer2Score, transcribeLabels, findPlayer1Score, setBackgroundLabelBlueOrGreen, capturedCards, setBackgroundLabelRedOrYellow, getScore1, setScore1, getScore2, setScore2, setPlayer1Name, getPlayer1Name, start15SecCountDown, start5SecCountDown, getPlayer2Name, setPlayer2Name, setPlayer1Turn, setPlayer2Turn, getCardAtHandLocation, setCardAtHandLocation, createPanel, playerHandIsEmpty, whoWon, setRightCardTiles, setplay, setOutcomeDisplay
Class3	getNumberOfCardsOnBattleField, setCardAtMainLocation, getMainTiles, getCardAtMainLocation, setMainTiles, generateTileNullsGrid

Following is a list of functions in their respective refactored classes for the SSM based approach.

Table 7: Functional distribution in refactored classes, case study II (VSM)

Class Name	Function List
Class1	getCardAtHandLocation, setCardAtHandLocation, playerHandIsEmpty, setRightCardTiles
Class2	labelsFromInts, findPlayer2Score, capturedCards, transcribeLabels, setBackgroundLabelBlueOrGreen, setBackgroundLabelRedOrYellow, findPlayer1Score, createPanel, initialLabelsFiller, setOutcomeDisplay, setPlay, getScore1, setScore1, settScore2, getScore2
Class3	getNumberOfCardsOnBattleField, getCardAtMainLocation, generateTileNullsGrid, battleFieldIsEmpty, setCardAtMainLocation, generateTileFieldsGrid, setBattleField, getBattleLabels, setMainTiles, getMainTiles
Class4	setPlayer2Turn, setPlayer1Turn, getPlayer2Name, setPlayer2Name, getPlayer1Name, setPlayer1Name
Class5	start15SecCountDown, start5SecCountDown
Utility Class	whoWon

Comparison of refactored classes

A metric and design analysis is provided for the original class and the classes refactored using both the approaches. For comparison purposes, the analysis is made side by side.

Table 8: Metrics analysis of original and refactored classes, case study II

Type of Metric	Metric Name	Original Class	Class1		Class2		Class3		Class4	Class5	Utility
			SSM	VSM	SSM	VSM	SSM	VSM	VSM		
Countable Metrics	LOC	508	181	32	373	267	43	68	34	57	10
	M	38	7	4	27	15	7	10	6	2	1
	F	24	1	1	22	16	1	1	3	3	0
Cohesion	LCOM	0.91	1.0	0.0	0.90	0.77	0.0	0.0	0.0	0.66	0.0
Coupling	CBO	0	2	2	2	3	2	1	0	0	1
Complexity	WMC	120	33	10	63	44	6	1	3	11	1
	FAT	119	2	5	85	62	6	4	9	8	4

Very high metric value Moderate to high metric value

Low metric value Metric value leads to no conclusion about code quality.

The high values of number of methods and fields in the original class attribute to its high complexity. Design analysis of the class shows that the class has numerous sets of functionalities, for example: building a graphical view, managing game players, managing different types of cards, managing locations where the cards are placed (player's table / common table). Each different set of functionalities can be extracted into separate classes.

The refactored classes as observed from the metric evaluation are more cohesive (based on LCOM and design analysis) and less complex (based on FAT and WMC) as compared to the original class. Coupling is introduced in the refactored classes but the overall quality of the code is improved.

Complexity as observed using the WMC and FAT metrics, for Class2 (SSM) is very high. This is attributed to the fact that this class retains most of the original class' functionalities (methods and variables). On the contrary, Class2 for VSM based approach is significantly lower in complexity and has clearer design roles. None of the other classes exhibit complex characteristics.

The coupling between classes formed using the VSM based approach is within the satisfactory range; Class3 is the only class having CBO = 3 entailing class dependency with three classes (out of a total of six classes). On the contrary, the SSM based approach produced three classes each being coupled to the other, implying that changing any class in the system can potentially induce a change in another.

The LCOM values for refactored classes based on the VSM approach is significantly lower than its SSM based counterpart. The class design analysis in Table 9 provides some more intuition into the cohesiveness of the two systems. It lists down the output classes and their high-level functionalities. A direct comparison between the refactored classes can be drawn using the table.

Table 9: Design analysis of refactored classes, case study II

Class Name	SSM based approach	VSM based approach
Class1	Initializes the graphical display parameters of "battle field", the common area where cards are placed in a card game.	Sets, resets and queries cards at the player's table in a card game.

Table 9 (continued)

Class Name	SSM based approach	VSM based approach
Class2	Contains almost all the functionalities of the original class except the ones that deal with “battle field”.	Contains functions that handle the graphical display of the system as well as functions that calculate values to be set at the display.
Class3	Sets cards on the “battle field”.	Works with functions related to the “battle field”, i.e. initializing graphical display parameters and setting cards on it
Class4	N/A	Deals with the players of the game. Sets the names of the players and decides which player’s turn it is to play.
Class5	N/A	Sets, resets and counts down game timers that run after the game has finished.
Utility Class	N/A	Provides a single helper function.

From the table, it appears that the functionalities in Class 1 and 3 (in the SSM approach) are related and should be grouped together into a single class, whereas Class 3 (in the VSM approach) handles this by providing the grouped functionalities.

Also, Class2 in the SSM based approach is very similar to the original blob class. It exhibits both complexity and lack of cohesiveness. This has been addressed in the VSM approach by

further dividing the class into more classes. However, Class5 in the VSM based approach provides two interconnected helper functions that should really be a part of the Utility class.

Conclusion

It is quite evident from the quality analysis of the two approaches that the classes formed using the VSM based approach have better defined functionalities. Moreover, these classes are less complex and are cohesive and have lower coupling. On the other hand, the SSM-based approach generated classes that have more complex and have higher coupling.

5.2. Analytical study of CMM

CMM provides a methodology to identify isolated sets of data members in a class and extract them into smaller classes. In this section, we study the impact of using CMM by applying it to a case study. We first describe the case study and the original blob class. Next, we evaluate the values of the metric for the different methods of the class. Then, we use the metric values to refactor the blob class. Finally, we discuss why it is necessary sometimes to use CMM.

5.2.1. Case Study III (SpaceBattle)

Description

SpaceBattle is a single player combative game where the user has a drone that fires bullets at computerized drones and conquers arenas. In this case study, one of the classes in this project, named Player, is analyzed. The class Player is selected for study as manual inspection reveals that it has many isolated data members that can be segregated out into different classes having clear independent set of functionalities. Some of the functionalities are: performing game related

operations (firing weapons), managing health points of each drone, changing the game display, and updating player specific configurations.

Evaluating the CMM

In Section 3.2.2, we describe the technique used for calculating the CMM metric. We build a graph (Figure 10) to focus on the relationships between every pair of variable in the original class. Every variable is a node in the figure and two nodes having the same color indicate that they are part of an isolated set. Each isolated set of variables, along with the associated methods, will be extracted to a separate class. From the figure, it is clear that there are four sets of functionalities obtained by calculating the CMMs and thus there will be at least four classes that will be extracted from the original blob class. In addition, there is the possibility of extracting a Utility class too.

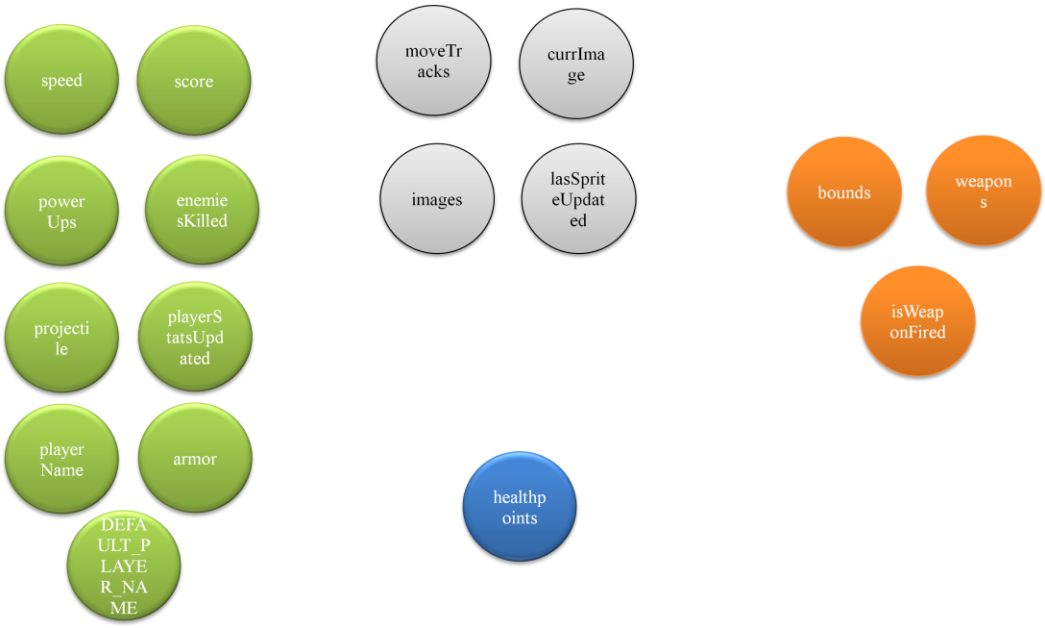


Figure 10: CMM graph case study III. Every node represents an instance variable; nodes with the same color belong to the same isolated sets.

Refactoring the blob class

Following is a list of functions and variables in their respective classes after refactoring.

Table 10: Functional and Variable distribution of refactored classes using VSM Based Approach, case study III:

Class Name	Function List	Variable List
Class1	addHealth, removeHealth, getHealth, setHealth	healthPoints
Class2	updateBounds, getBounds, getX, getY, setX, setY, setPoints, fireWeapon, setWeaponRate, setWeapon, getWeapon	bounds, weapon, isWeaponFired
Class3	updateImage, getImage, setMoveTrack	lastSpriteUpdateTime, moveTracks, images, curImage
Class4	initializeVariables, setPlayerName, getPlayerName, getArmor, setArmor, addPowerUp, getPowerUps, getSpeed, setSpeed, getSpriteToggleDelay, addProjectile, getProjectiles, updateScore, addEnemiesKilled, getScore, getEnemiesKilled	DEFAULT_PLAYER_NAME, enemiesKilled, powerups, playerStatsUpdated, projectiles, armor, playerName, speed, score
Utility Class	draw, collideWith	N/A

Generating refactored classes

A metric and design analysis is provided for the original class and the refactored classes using the CMM approach. For comparison purposes, the data is presented side by side.

Table 11: Metrics analysis of original and refactored classes (CMM), Case Study III

Type of Metric	Metric Name	OriginalClass	Class1	Class2	Class3	Class4	Utility Class
Countable Metrics	LOC	165	35	35	61	16	31
	M	37	4	11	3	16	2
	F	17	1	3	4	9	0
Cohesion	LCOM HS	0.98	0.0	0.70	0.75	0.95	0.0
Coupling	CBO	0	0	1	2	1	2
Complexity	WMC	41	4	3	5	5	4
	FAT	45	5	12	11	6	0

Very high metric value Moderate to high metric value

Low metric value Metric value leads to no conclusion about code quality.

The original blob class is moderately complex and has multiple functionalities cluttered in a single class. This class is responsible for performing game related operations (firing weapons), managing health points of each drone, changing the game display, and updating player specific configurations.

The refactored classes are much less complex as indicated by their WMC and FAT values. The classes are also more cohesive which can be observed from the detailed description of roles for every class below:

- Class1 – This class contains data variables and functions that modifying, removing, setting and getting player health points.

- Class 2 – This class is responsible for defining API calls that can help achieve certain game specific operations like a) firing of weapons from the drone b) setting the boundary of the current game arena. Clearly the class offers two sets of functionalities and can be further broken down into separate classes. This will be discussed in Section 5.2.2 in detail.
- Class 3 – This class updates the game screen depending upon the current position of the drone. It also has only one overall operation to perform.
- Class 4 – This class is responsible for handling player specific configurations like name, power, speed, score etc.
- Utility Class – This class has two helper functions and is non cohesive.

Most of the classes extracted using the CMM technology have clear and concise functionalities. Class 2 however can be further refactored using the VSM technology. This is discussed in the following section.

Additional refactoring using the VSM metric

Upon further refactoring of Class_2 using the VSM approach the following closely related function sets were formed

- Set 1 - updateBounds, getBounds, getX, getY, setX, setY, setPoints
- Set 2 - fireWeapon, setWeaponRate, setWeapon, getWeapon

This refactoring was performed at minCoupling = 0.52 and minLength = 2. These two sets are very cohesive and can be extracted out as separate classes. The six classes produced using the combination of CMM and VSM based approaches have very clear set of responsibilities.

Note that with the VSM based approach alone (i.e. not using CMM), no value of minCoupling was able to produce the six classes. With minCoupling = 0 the same classes as described in table 10 are formed. As Class 2 is not cohesive, we must increase minCoupling to divide its functionalities. Upon increasing minCoupling, Class 2 gets divided into the desired subclasses, but unfortunately, Class 1 and 3 are dissolved and joins with Class 4. Thus a better approach is to first refactor to the five classes using CMM and to then refactor Class 2 into the sub classes using VSM to yield the best results.

5.3. **Summary Observations**

Observations from case studies I and II

Here is a summary of the results from the first two case studies.

- In comparison to the SSM approach, the VSM approach showed improvement in the coupling based metrics, i.e. CBO and MPC metrics. This is because our approach favors assigning variables to classes having methods that write on the variable. This means that with good probability the classes which end up not having the variable will only need to read it. This variable is passed as an argument to the function that attempts to read it, thereby decreasing the need for setter and getter method calls. Our approach also favors assigning variables to classes which most frequently use it. This decreases the amount of setter and getter calls for the variable which in turn reduces messages between the extracted classes in comparison to the approach that uses SSM.
- If we consider the manner in which variables are assigned to classes, the LCOM metric should give lower values in case of the approach using SSM approach as it assignes variable to classes which have more methods using it and LCOM is lower for classes

having variables that are used in more methods. As the VSM based approach focuses on assigning a variable to the class that writes to it, or uses it more frequently, the LCOM value for these classes should be higher. However, as the VSM based approach also creates an Utility class with all unrelated methods in it, the LCOM value for other classes decreases. Thus the LCOM metric comparison is specific to the class being refactored.

  - Note that LCOM is not advocated as an actual measure of cohesiveness. A low value of LCOM may signify a good design but a high value does not indicate a bad one [26]. Moreover, it is also important to look at the frequency and the manner in which a variable is used in a function. This is ignored in LCOM, which is why we also provide a design analysis of the final classes for cohesiveness.

- Complexities in the case of our approach were also lower than its counterpart. This is because of a better functional distribution yielding to a more efficient and thereby less complex classes.
- As observed in the case studies, the overall design for VSM is better. The grouping of methods in classes is more meaningful.
- The Utility class has no cohesiveness. This is because the methods of a Utility class are completely unrelated to any other method in the original class. Obviously, a collection of these isolated helper methods will not form a cohesive class, but nonetheless, it remains a good idea to have a Utility or a Helper class to handle such disparate functionalities.

Observations from Case Study III

- CMM is good for identifying isolated data members of a class and for segregating them into separate classes. The result is usually a set of cohesive classes.

- CMM methodology should not be used in isolation as the only ECR technology; these classes may need further refactoring via other approaches like VSM.
- CMM methodology does not yield results with all god classes. It only produces good results with classes having clear isolatable functionalities.

CHAPTER 6: CONCLUSION AND FUTURE WORK

In this chapter we summarize the work done in this research and propose some future directions in the same field.

6.1. Conclusion

Through this research, many adaptations and additions to the current practices in the field of ECR are introduced. The necessities and applications of ECR are discussed. Furthermore, an implementation of the proposed research and the previous research is used as the basis for a comparative study which verifies the applicability of the research. The implementation tool is further enhanced to provide the developers with the opportunity of selecting the final design and then automatically refactoring the code to that design. Lastly, software metrics are applied against the final refactored classes to substantiate the effectiveness of the research. Our results indicate that our approach leads to improvements in the overall design of the refactored classes and that in general there is scope for further improvements in the ECR process.

6.2. Future Work

Here are a few ways in which this work can be extended:

- Development of an automated tool/plugin that detects design erosion and informs the developer when a class has lost its cohesiveness or increased its coupling with other classes. This can increase the usage of ECR tools.
- Development of a tool/plugin that alerts a developer whenever they are about to perform ECR manually. This decreases the dangers of manual refactoring by suggesting automated refactoring tools for the same.

- Development of a free, easily available plugin for famous IDEs like netbeans and eclipse for automated ECR.
- Removing user perception from the process of generating the final refactored design. This can be achieved by evaluating all the possible refactored designs using software metric to determine the best possible one (having maximum cohesion and minimum coupling).
- The tool can be further enhanced to make intelligent decisions about refactoring by including CDM and CSM while calculating the similarity between methods.
- The tool will currently not work for classes extending other classess or implementing interfaces or classes with inner classes. The tool does not resolve method calls and class variable accesses from outside classes.
- Automatic naming of extracted classes based on method and variable names.
- A project wide design restructuring based on VSM.
- Further research can be done into the formulas used to caculate VSM (Section 3.1.2) and assign instance variables to the classes (Section 3.3.1).
- A tool that run in the background of an IDE and automatically finds and suggests clearly isolated sets in the classes while the developer is coding on the system will be very useful.

REFERENCES

[1] Bart Du Bois, Pieter Van Gorp, Alon Amsel, Niels Van Eetvelde, Hans Stenten, Serge Demeyer and Tom Mens. “A Discussion of Refactoring in Research and Practice”, http://win.ua.ac.be/~lore/refactoringProject/publications/ADiscussionOfRefactoringInResearchAndPractice.pdf

[2] Emerson Murphy-Hill, Chris Parnin and Andrew P. Black. “How We Refactor, and How We Know It”. International Conference on Software Engineering 2009.

[3] Martin Fowler. “Alpha List of Refactoring”, http://www.refactoring.com/catalog/

[4] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol and Yann-Ga¨el Gu´eh´eneuc. “Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory”, International Conference on Software Maintenance, 2010.

[5] Marios Fokaefs, Nikolaos Tsantalis and Eleni Stroulia “JDeodorant: Identification and Application of Extract Class Refactorings”, International Conference on Software Engineering, 2011

[6] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Fabio Palomba and Andrian Marcus, “Supporting Extract Class Refactoring in Eclipse: The ARIES project”, International Conference on Software Engineering 2012

[7] Yi Wang. “What Motivate Software Engineers to Refactor Source Code? Evidences fromProfessional Developers”. International Conference on Software Engineering 2009

[8] Iman Hemati Moghadam, Mel ´O Cinn´eide. “Automated Refactoring using Design Differencing”, European Conference on Software Maintenance and Reengineering 2012

[9] G. Bavota, A. De Lucia, and R. Oliveto. “Identifying Extract Class Refactoring opportunities using structural and semantic cohesion measures” The journal of system and software, 84:397–414, 2011.

[10] G. Gui and P. D. Scott. “Coupling and cohesion measures for evaluation of component reusability”. Mining Software Repositories, 18–21, 2006.

[11] A. Marcus, D. Poshyvanyk, and R. Ferenc. “Using the conceptual cohesion of classes for fault prediction in object oriented systems” IEEE Transactions on Software Engineering, 34(2):287–300, 2008.

[12] M. Fowler. “Refactoring: improving the design of existing code”. Addison-Wesley, 1999.

[13] Mealy, E.; Carrington, D.; Strooper, P.; Wyeth, P., "Improving Usability of Software Refactoring Tools", Software Engineering Conference, 2007. ASWEC 2007. 18th Australian , vol., no., pp.307,318, 10-13 April 2007

[14] Mealy, E.; Strooper, P., "Evaluating software refactoring tool support," Software Engineering Conference, 2006. Australian , vol., no., pp.10 pp.,, 18-21 April 2006, doi: 10.1109/ASWEC.2006.26

[15] T. Tourw´e and T. Mens. Identifying refactoring opportunities using logic meta programming. In Proceedings of 7th European Conference on Software Maintenance and Reengineering, pages 91–100. IEEE Computer Society, 2003.

[16] Xi Ge; DuBose, Q.L.; Murphy-Hill, E., "Reconciling manual and automatic refactoring," *Software Engineering (ICSE), 2012 34th International Conference on* , vol., no., pp.211,221, 2-9 June 2012

[17] Jasmeet Singh, "Survey on Why are developers cautionary of automatic refactoring tools", http://jasmeetsingh.net/refactoring/?p=31

[18] Vakilian, M.; Chen, N.; Negara, S.; Rajkumar, B.A.; Bailey, B.P.; Johnson, R.E., "Use, disuse, and misuse of automated refactorings," *Software Engineering (ICSE), 2012 34th International Conference on* , vol., no., pp.233,243, 2-9 June 2012

[19] The Blob, *Source Making*, Retrieved May 28th, 2013 from http://sourcemaking.com/antipatterns/the-blob

[20] Gabriele Bavota , Andrea De Lucia , Andrian Marcus , Rocco Oliveto, A two-step technique for Extract Class Refactoring, Proceedings of the IEEE/ACM international conference on Automated software engineering, September 20-24, 2010, Antwerp, Belgium [doi>10.1145/1858996.1859024]

[21] Osborne, Martin J., and Ariel Rubinstein. "A Course in Game Theory". Cambridge, MA: MIT, 1994. Print.

[22] SONAR, *Documentation*, Retrieved June 6th, 2013 from http://docs.codehaus.org/display/SONAR/Documentation

[23] JArchitecture, *Metrics*, Retrieved June 6th, 2013 from http://www.jarchitect.com/Metrics.aspx

[24] STAN4J, *Structural Analysis for Java*, Retrieved June 6th, 2013 from http://stan4j.com/

[25] Source code Metrics, *Metrics 1.3.6*, Retrieved June 6th, 2013 from http://metrics.sourceforge.net/

[26] Sami Mäkelä and Ville Leppänen.”Observation on Lack of Cohesion Metrics”. Proceedings of the International Conference on Computer Systems and Technologies - CompSysTech’06.

[27] Julio Gesser, *JavaParser*, Retrieved June 22nd, 2013 https://code.google.com/p/javaparser/

[28] Wikipedia, *Max-flow min-cut theorem*, Retrieved June 22nd, 2013 http://en.wikipedia.org/wiki/Max-flow_min-cut_theorem

APPENDIX A1

REASONS FOR UNDER-USAGE OF AUTOMATED REFACTORING TOOLS

Table 12: Reasons for under-usage of refactoring tools and proposed resolutions

Issue	Description of the issue	Proposed Resolution
Need	Blob antipatterns are formed after a series of development changes resulting in huge classes. The developer might not realize the need for ECR until the formation of blobs	This problem can be resolved by building more awareness
Awareness	Most automatic refactoring tools are not widely known or used.	A future prospect of this project is to build a plug-in for famous IDEs like Eclipse and Netbeans.
Naming	Some refactoring tools are under-used because they have a complex name and hence difficult to search for	The tool developed in this thesis is called “Extract Class Refactoring tool”
Trust	Many developers are wary of the tools changing behavior and design of the system.	Substantial testing and increased awareness can build trust among users
Predictability	The final refactored design can be unlike the user’s expectation and may cause further inconvenience to them.	The developer is involved in the selection of the design for the final classes, thereby, reducing the surprise factor
Configuration	Too much configuration requirement from the user could be a deterrent factor	We provides an interactive GUI with only one configurable field

APPENDIX A2

SOFTWARE QUALITY METRICS

Table 13: Coupling, Cohesion and Complexity metrics

Metric	Description of the metric
Number of Methods (M)[10]	Total number of methods in the class.
Number of Fields (F) [10]	Total number of class variables
Lines of Code (LOC) [10]	Total lines of code in the class
Coupling Between Objects (CBO) [10]	Two classes are considered coupled if methods or variables of one class are used in another. CBO for a class is the measure of number of classes coupled to it
Response for a Class(RFC) [10]	RFC is the sum of number of methods in a class and the number of distinct methods calls made by methods in the class

Table 13 (continued)

Metric	Description of the metric
Lack of cohesion of metrics (LCOM) [23]	Henderson-Sellers LCOM HS is computed in [23] as follows $$LCOM\ HS = \frac{(M - \frac{\sum_M F}{F})}{(M-1)}$$ [23] Also computes LCOM as follow $$LCOM = (1 - \frac{\sum_M F}{M * F})$$ Where, M = number of methods in class (static, instance methods, constructors, properties getters/setters, events add/remove methods), F = number of instance fields, MF = number of methods accessing a particular instance field, $\sum_M F$= summation of MF for all instance fields of the class. A special note here is that LCOM has never been advocated to measure the accurate cohesiveness of a class. In fact a low value of LCOM can be indicative of a cohesive class, but a high value of LCOM does not necessarily point to a bad design [26]. The best way to measure cohesiveness is to manually assess the class design.
Ratio Lack of cohesion of metrics (RLCOM) [10]	Ratio of unrelated method pairs to total number of method pairs
Tight class cohesion (TCC) [10]	Ratio of related method pairs to total number of method pairs.

Table 13 (continued)

Metric	Description of the metric
Loose class cohesion (LCC) [22]	Ratio of related method pairs (both directly and indirectly related) to total number of pair
Afferent Coupling (Ca) [23]	Number of classes outside the assembly that are dependent upon a class. High Ca indicates high responsibility for the class
Efferent Coupling (Ce) [23]	Number of classes outside the assembly that a class uses. High Ce indicates that the class is dependant
Association Between Classes (ABC) [23]	Number of members of other classes that are directly used by a class
McCabe Cyclomatic Complexity (CC) [23]	Total number of decisions that can be taken in a module. CC is incremented by 1 for every branch in the code.
Weighted Methods for Class (WMC) [25]	Sum of McCabe CC for all methods in a class
FAT [24]	FAT measures the dependency of methods on a variable. FAT is incremented by 1 for every method that depends on a variable. FAT is important because it measures the complexity of the class
Message Passing Coupling (MPC)	Total number of messages in the form of method calls communicated between two classes

APPENDIX B1

UNFILTERED SIMILARITIES BETWEEN METHODS. CASE STUDY I

Similarity between methods based on SSM and VSM

Figure 6 depicts the relationships between every pair of methods in the original class. Every method is a node in the graph and the weight of the edge between two methods represents the similarity between them based on SSM or VSM. Methods with no edge between them are not similar at all.

Because the graph might be difficult to read and analyze, a tabular representation is also provided. Every cell in the table corresponds to the similarity between the methods represented by the corresponding row and column headers of the table. Cells with no value represent 0 similarities.

Figures 11 and 12 portray a graphical representation of the input class in terms of methods and their similarity (SSM and VSM respectively). Following to that Figures 13 and 14 represent the table view of the method similarity (SSM and VSM respectively).

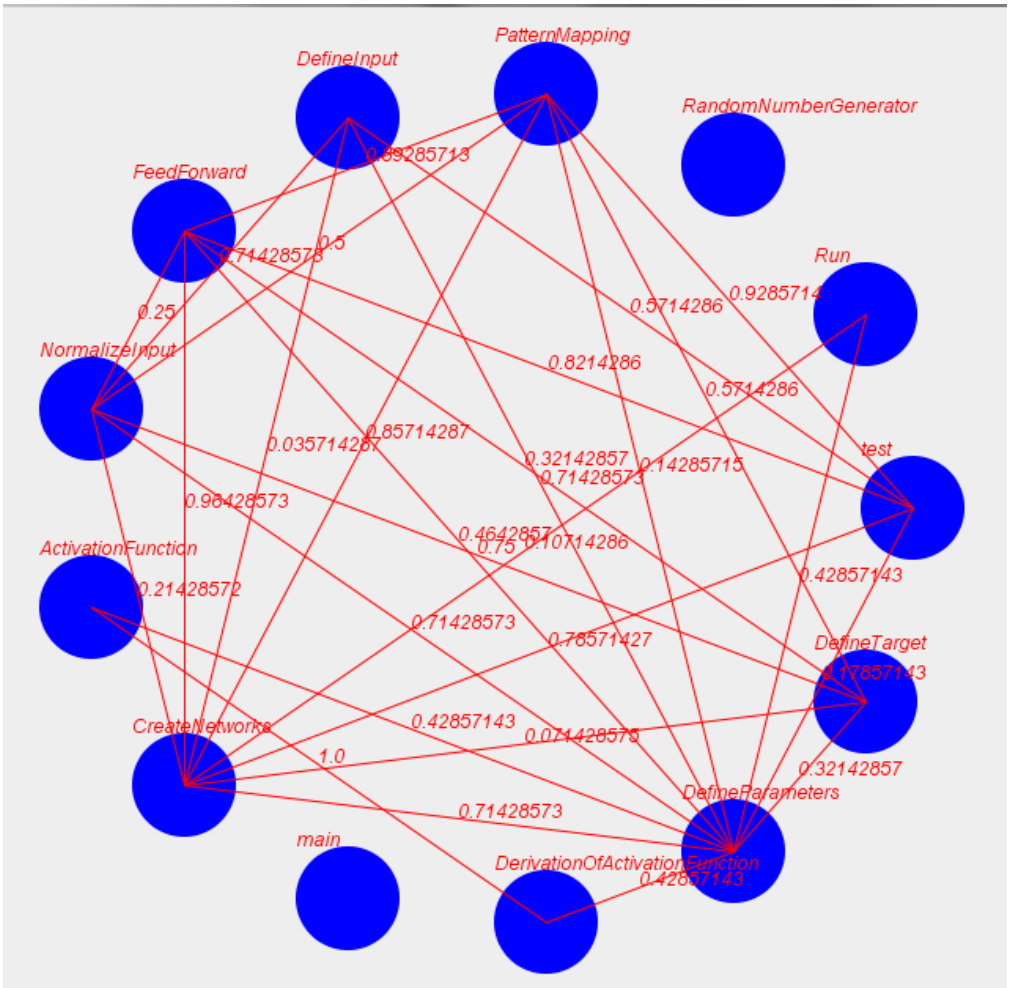


Figure 11: SSM based graph, case study I, minCoupling = 0 – This is a graphical representation of the SSM based structural similarity between every pair of methods for the input class

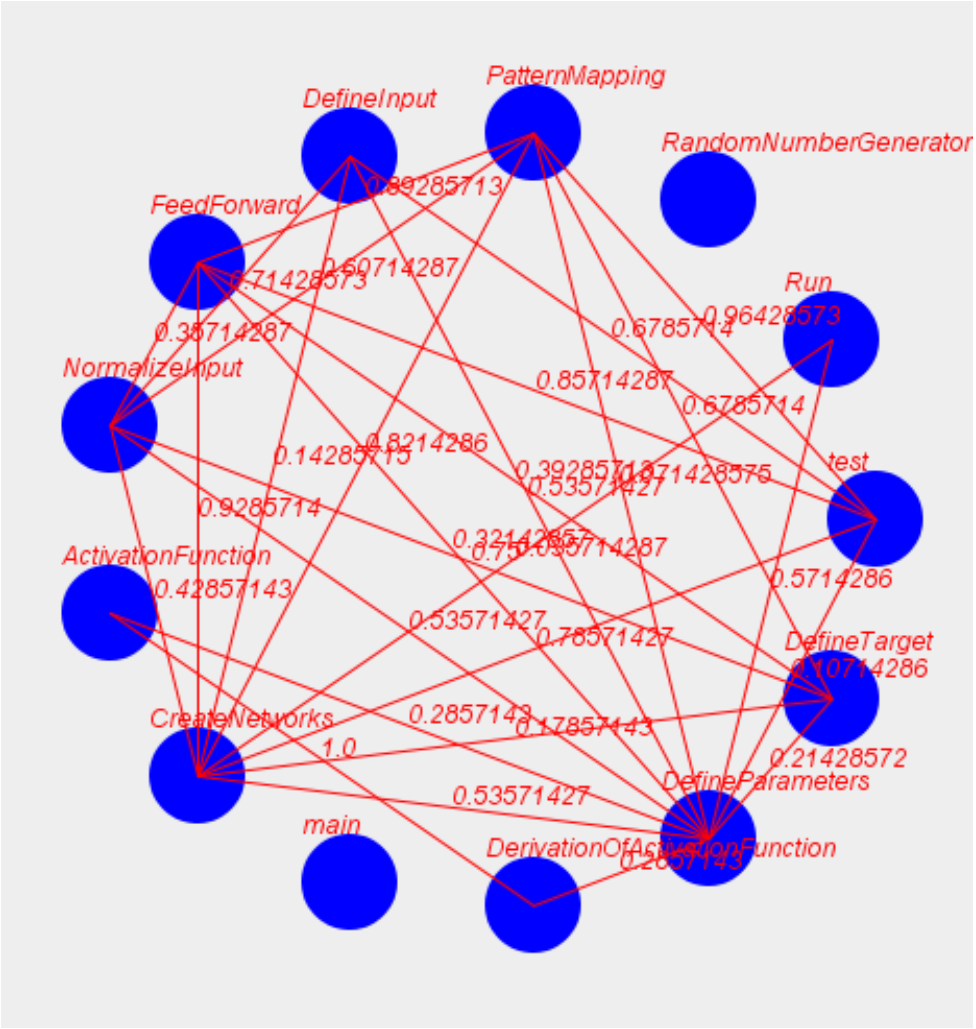


Figure 12: VSM based graph, case study I, minCoupling = 0 – This is a graphical representation of the VSM based structural similarity between every pair of methods for the input class

Fn()	test	DefineTarget	DefineParameters	DerivationOfActivationFu...	main	CreateNetw...	ActivationFun...	NormalizeIn...	FeedForward	DefineInput	PatternMap...	RandomN...	Run
test			0.17857143			0.78571427			0.8214286	0.5714286	0.9285714		
DefineTarget			0.32142857			0.071428575		0.75	0.32142857		0.5714286		
DefineParameters	0.17857143	0.32142857		0.42857143		0.71428573	0.42857143	0.71428573	0.4642857	0.71428573	0.14285715		0.42857143
DerivationOfActivati...			0.42857143				1.0						
main													
CreateNetworks	0.78571427	0.071428575	0.71428573					0.21428572	0.96428573	0.0357142...	0.85714287		0.10714286
ActivationFunction			0.42857143	1.0									
NormalizeInput		0.75	0.71428573			0.21428572			0.25	0.71428573	0.5		
FeedForward	0.8214286	0.32142857	0.4642857			0.96428573		0.25			0.89285713		
DefineInput	0.5714286		0.71428573			0.035714287		0.71428573					
PatternMapping	0.9285714	0.5714286	0.14285715			0.85714287		0.5	0.89285713				
RandomNumberGe...													
Run			0.42857143			0.10714286							

Figure 13: SSM matrix representation, case study I, minCoupling = 0 – This is the matrix representation of the SSM based structural similarity between every pair of methods for the input class

Fn()	test	DefineTarget	DefineParam...	DerivationOfA...	main	CreateNetwor...	ActivationFun...	NormalizeInput	FeedForward	DefineInput	PatternMappi...	RandomNum...	Run
test			0.17857143			0.78571427			0.8214286	0.5714286	0.9285714		
DefineTarget			0.32142857			0.071428575		0.75	0.32142857		0.5714286		
DefineParam...	0.17857143	0.32142857		0.42857143		0.71428573	0.42857143	0.71428573	0.4642857	0.71428573	0.14285715		0.42857143
DerivationOfA...			0.42857143				1.0						
main													
CreateNetwor...	0.78571427	0.071428575	0.71428573					0.21428572	0.96428573	0.035714287	0.85714287		0.10714286
ActivationFun...			0.42857143	1.0									
NormalizeInput		0.75	0.71428573			0.21428572			0.25	0.71428573	0.5		
FeedForward	0.8214286	0.32142857	0.4642857			0.96428573		0.25			0.89285713		
DefineInput	0.5714286		0.71428573			0.035714287		0.71428573					
PatternMappi...	0.9285714	0.5714286	0.14285715			0.85714287		0.5	0.89285713				
RandomNum...													
Run			0.42857143			0.10714286							

Figure 14: VSM matrix representation, case study I, minCoupling = 0 – This is the matrix representation of the VSM based structural similarity between every pair of methods for the input class

APPENDIX B2

FILTERED SIMILARITIES BETWEEN METHODS. CASE STUDY I

Filtered Similarity between methods based on SSM and VSM

We use a threshold minCoupling to filter out less similar methods. Any pair of method with the percentile of similarity less than minCoupling is considered dissimilar, and their similarity is equated to 0. The drawback of having a large minCoupling is that many valuable chains might dissolve or shorten down after losing some of its valuable members. Another drawback is that a high minCoupling which leads to more number of class divisions will lead to a higher coupling.

Figures 15 and 16 represent the filtered graphical representation of the class (SSM and VSM respectively). Followed by that, Figures 17 and 18 represent the table view of the filtered similarities based on SSM and VSM respectively. miCoupling for SSM approach was 0.72 and for VSM approach was 0.69.

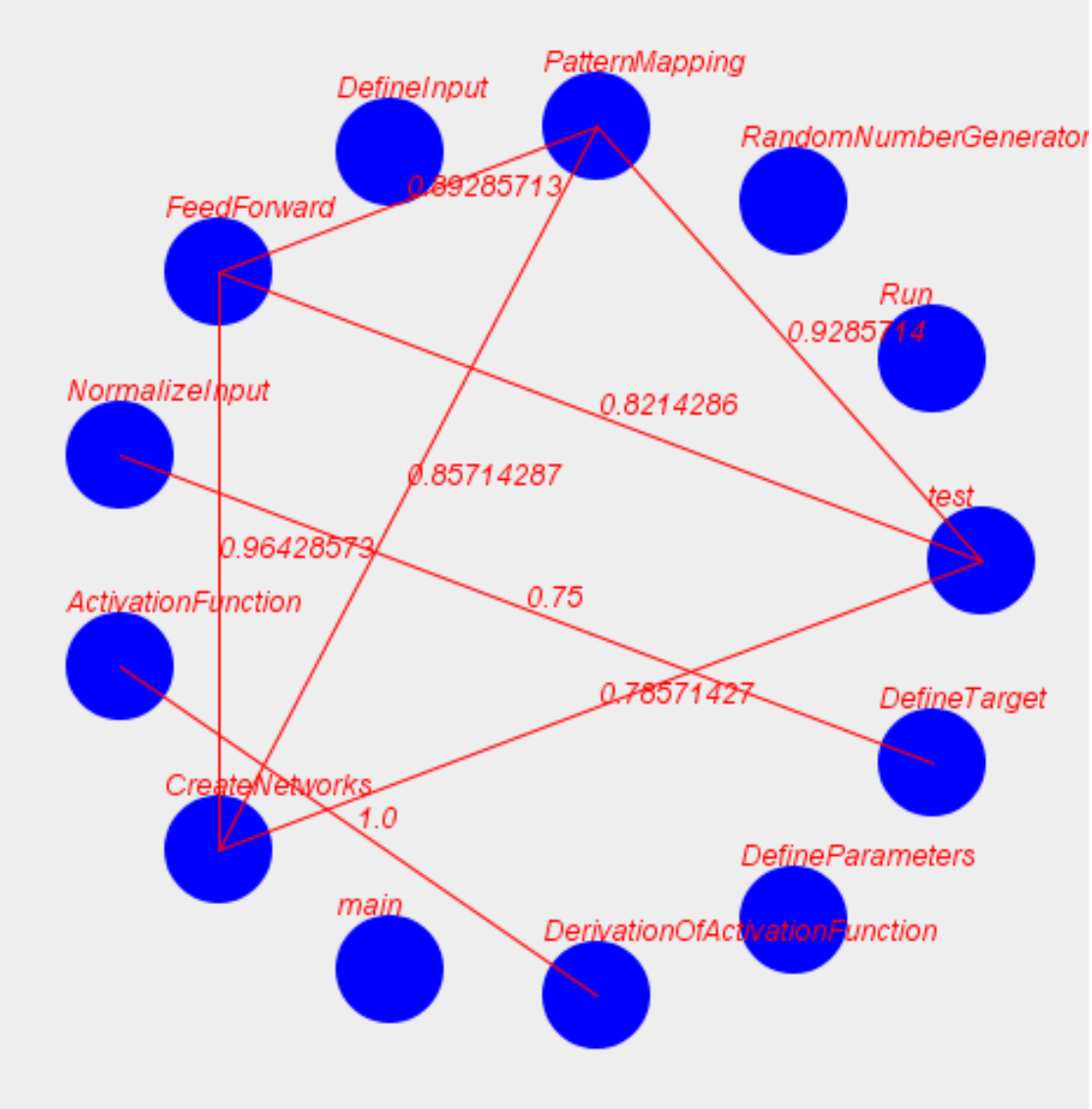


Figure 15: SSM based graph, case study I, minCoupling = 0.72 – This is a graphical representation of the SSM based structural similarity between every pair of methods for the input class

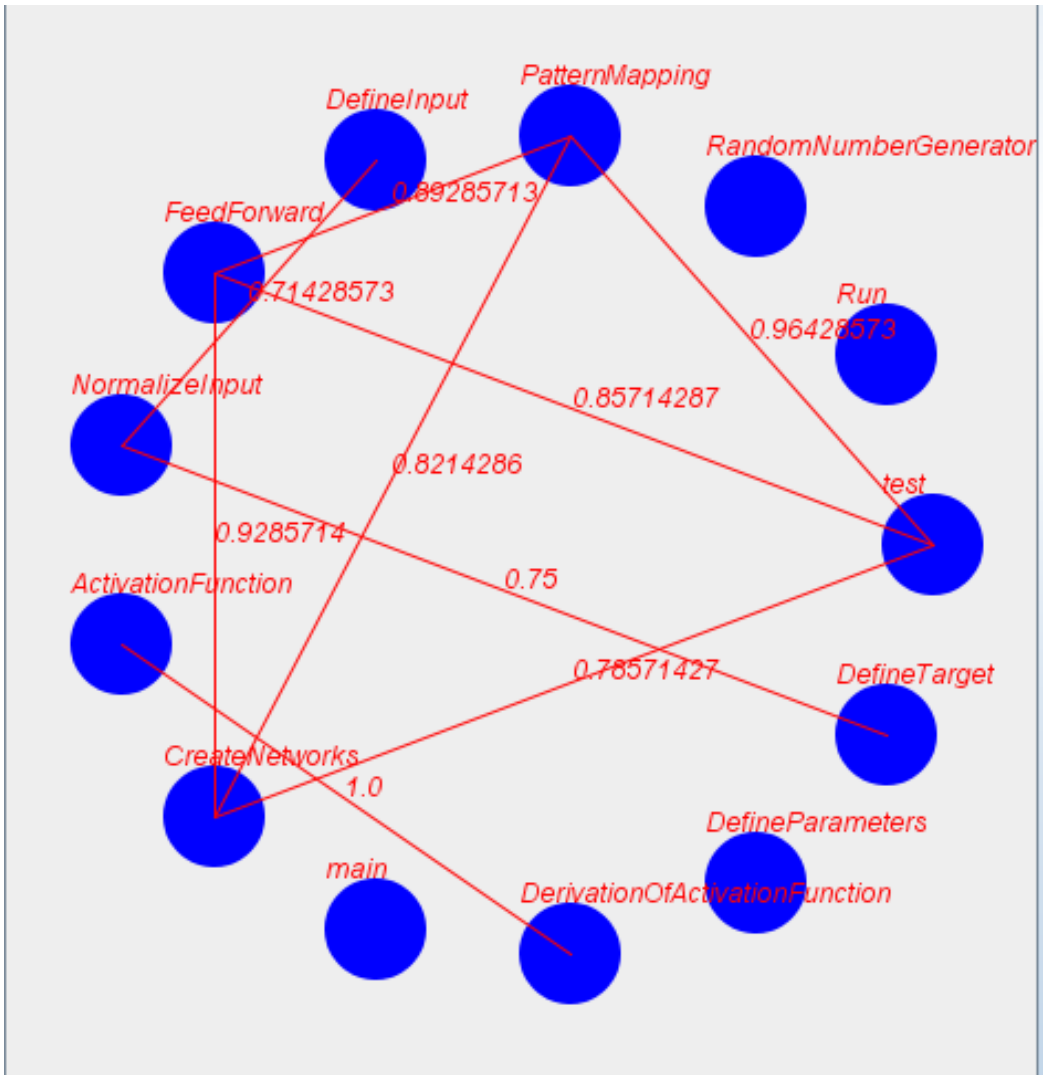


Figure 16: VSM based graph, case study I, minCoupling = 0.69 – This is a graphical representation of the VSM based structural similarity between every pair of methods for the input class

Fn()	test	DefineTarget	DefineParam...	DerivationOfA...	main	CreateNetwor...	ActivationFun...	NormalizeInput	FeedForward	DefineInput	PatternMappi...	RandomNum...	Run
test						0.78571427			0.8214286		0.9285714		
DefineTarget								0.75					
DefineParam...													
DerivationOfA...							1.0						
main													
CreateNetwor...	0.78571427								0.96428573		0.85714287		
ActivationFun...				1.0									
NormalizeInput		0.75											
FeedForward	0.8214286					0.96428573					0.89285713		
DefineInput													
PatternMappi...	0.9285714					0.85714287			0.89285713				
RandomNum...													
Run													

Figure 17: SSM matrix representation, case study I, minCoupling = 0.72 – This is the matrix representation of the SSM based structural similarity between every pair of methods for the input class

Fn()	test	DefineTarget	DefineParam...	DerivationOfA...	main	CreateNetwor...	ActivationFun...	NormalizeInput	FeedForward	DefineInput	PatternMappi...	RandomNum...	Run
test						0.78571427			0.85714287		0.96428573		
DefineTarget								0.75					
DefineParam...													
DerivationOfA...							1.0						
main													
CreateNetwor...	0.78571427								0.9285714		0.8214286		
ActivationFun...				1.0									
NormalizeInput		0.75								0.71428573			
FeedForward	0.85714287					0.9285714					0.89285713		
DefineInput								0.71428573					
PatternMappi...	0.96428573					0.8214286			0.89285713				
RandomNum...													
Run													

Figure 18: VSM matrix representation, case study I, minCoupling = 0.69 – This is the matrix representation of the VSM based structural similarity between every pair of methods for the input class

APPENDIX C1

UNFILTERED SIMILARITIES BETWEEN METHODS

CASE STUDY II

Similarity between methods based on SSM and VSM in the original class

Figures 19 and 20 represent the table view of the similarities based on SSM and VSM respectively.

Fn()	createP...	getPlayer2...	start15...	setPlayer2...	start5...	who...	setOut...	getScor...	getS...	initialL...	setMain...	setPlayer1...	label...
createPanel		0.0945273...	0.1393...	0.094527...	0.139...		0.0945...	0.0945...	0.09...	0.169...	0.0945...	0.094527...	0.31...
getPlayer2Name	0.0945...			1.0									
start15SecCountDown	0.1393...				1.0								
setPlayer2Name	0.0945...	1.0											
start5SecCountDown	0.1393...		1.0										
whoWon													
setOutcomeDisplay	0.0945...												
getScore2	0.0945...												
getScore1	0.0945...												
initialLabelsFiller	0.1691...										0.6567...		0.20...
setMainLabels	0.0945...									0.656...			0.42...
setPlayer1Name	0.0945...												
labelsFromInts	0.3134...									0.203...	0.4228...		

Fn()	trans...	getNum...	setBack...	setBattl...	getCard...	findPla...	setSco...	battleFiel...	setScore1	playerHa...	setMainTi...	findPla...
transcribeLabels			0.87064...	0.2935...		0.8706...		0.29353...				0.870...
getNumberOfCards...				0.8308...				0.8308458			1.0	
setBackgroundLabel...	0.870...			0.5522...		0.4228...		0.5522388				1.0
setBattleField	0.293...	0.83084...	0.55223...			0.5522...		1.0			0.8308458	0.552...
getCardAtHandLocat...										0.83084...		
findPlayer1Score	0.870...		0.42288...	0.5522...				0.5522388				0.422...
setScore2												
battleFieldIsEmpty	0.293...	0.83084...	0.55223...	1.0		0.5522...					0.8308458	0.552...
setScore1												
playerHandIsEmpty					0.8308...							
setMainTiles		1.0		0.8308...				0.8308458				
findPlayer2Score	0.870...		1.0	0.5522...		0.4228...		0.5522388				

Fn()	setBac...	captu...	setplay	getB...	setPlay...	setPl...	getPlay...	getCa...	setCard...	setCar...	getMain...	generat...	generat...	setRight...
setBackgroundLabel...		0.83...		0.65...	0.2039...	0.65...						0.5522...		0.29353...
capturedCards	0.8308...			0.29...	0.4378...	0.43...		0.293...		0.293...	0.29353...	0.6567...	0.20398...	0.55223...
setplay														
getBattleLabels	0.6567...	0.29...										0.8308...		
setPlayer1Turn	0.2039...	0.43...				0.87...	0.4228...							0.15422...
setPlayer2Turn	0.6567...	0.43...			0.8756...		0.4228...							0.15422...
getPlayer1Name					0.4228...	0.42...								
getCardAtMainLocati...		0.29...								1.0	1.0	0.8308...	0.83084...	
setCardAtHandLocat...													0.83084...	0.55223...
setCardAtMainLocati...		0.29...						1.0			1.0	0.8308...	0.83084...	
getMainTiles		0.29...						1.0		1.0		0.8308...	0.83084...	
generateTileFieldsG...	0.5522...	0.65...		0.83...				0.830...		0.830...	0.83084...		0.65671...	
generateTileNullsGrid		0.20...						0.830...	0.8308...	0.830...	0.83084...	0.6567...		0.42288...
setRightCardTiles	0.2935...	0.55...			0.1542...	0.15...			0.5522...				0.42288...	

Figure 19: SSM matrix representation, case study II, minCoupling = 0 – This is the matrix representation of the SSM based structural similarity between every pair of methods for the input class

Fn()	create...	start1...	start5...	whoW...	initial...	labels...	transc...	setBac...	getNu...	getCar...	setBa...	findPla...	battle...	findPla...	playe...	setBac...
createPanel		0.142...	0.142...		0.244...	0.306...	0.306...	0.190...	0.054...	0.034...	0.054...	0.074...	0.088...	0.1904...	0.08...	0.074...
start15SecC...	0.142...		1.0													
start5SecCo...	0.142...	1.0														
whoWon																
initialLabels...	0.244...					0.102...	0.102...	0.176...			0.231...	0.176...	0.231...	0.1768...	0.23...	0.176...
labelsFroml...	0.306...				0.102...		1.0	0.918...			0.408...	0.918...	0.408...	0.9183...		0.918...
transcribeLa...	0.306...				0.102...	1.0		0.918...			0.408...	0.918...	0.408...	0.9183...		0.918...
setBackgrou...	0.190...				0.176...	0.918...	0.918...				0.680...	0.455...	0.680...	1.0		0.455...
getNumber...	0.054...										0.680...		0.863...			
getCardAtHa...	0.034...														0.68...	
setBattleField	0.054...				0.231...	0.408...	0.408...	0.680...	0.680...			0.680...	0.938...	0.6802...		0.680...
findPlayer1S...	0.074...				0.176...	0.918...	0.918...	0.455...			0.680...		0.680...	0.4557...		1.0
battleFieldIs...	0.088...				0.231...	0.408...	0.408...	0.680...	0.863...		0.938...	0.680...		0.6802...		0.680...
findPlayer2S...	0.190...				0.176...	0.918...	0.918...	1.0			0.680...	0.455...	0.680...			0.455...
playerHandI...	0.088...				0.231...					0.680...						
setBackgrou...	0.074...				0.176...	0.918...	0.918...	0.455...			0.680...	1.0	0.680...	0.4557...		

Fn()	capt...	getBatt...	setP...	setPla...	get...	setC...	set...	gene...	setR...	gen...
capturedCar...		0.4081...	0.46...	0.469...	0.12...		0.12...	0.68...	0.68...	0.27...
getBattleLab...	0.40...							0.86...		
setPlayer2T...	0.46...			0.925...					0.23...	
setPlayer1T...	0.46...		0.92...						0.23...	
getCardAtMa...	0.12...						1.0	0.86...		0.68...
setCardAtHa...									0.68...	0.68...
setCardAtMa...	0.12...				1.0			0.86...		0.68...
generateTile...	0.68...	0.8639...			0.86...		0.86...			0.40...
setRightCar...	0.68...		0.23...	0.231...		0.68...				0.17...
generateTile...	0.27...				0.68...	0.68...	0.68...	0.40...	0.17...	

Figure 20: VSM matrix representation, case study II, minCoupling = 0 – This is the matrix representation of the VSM based structural similarity between every pair of methods for the input class

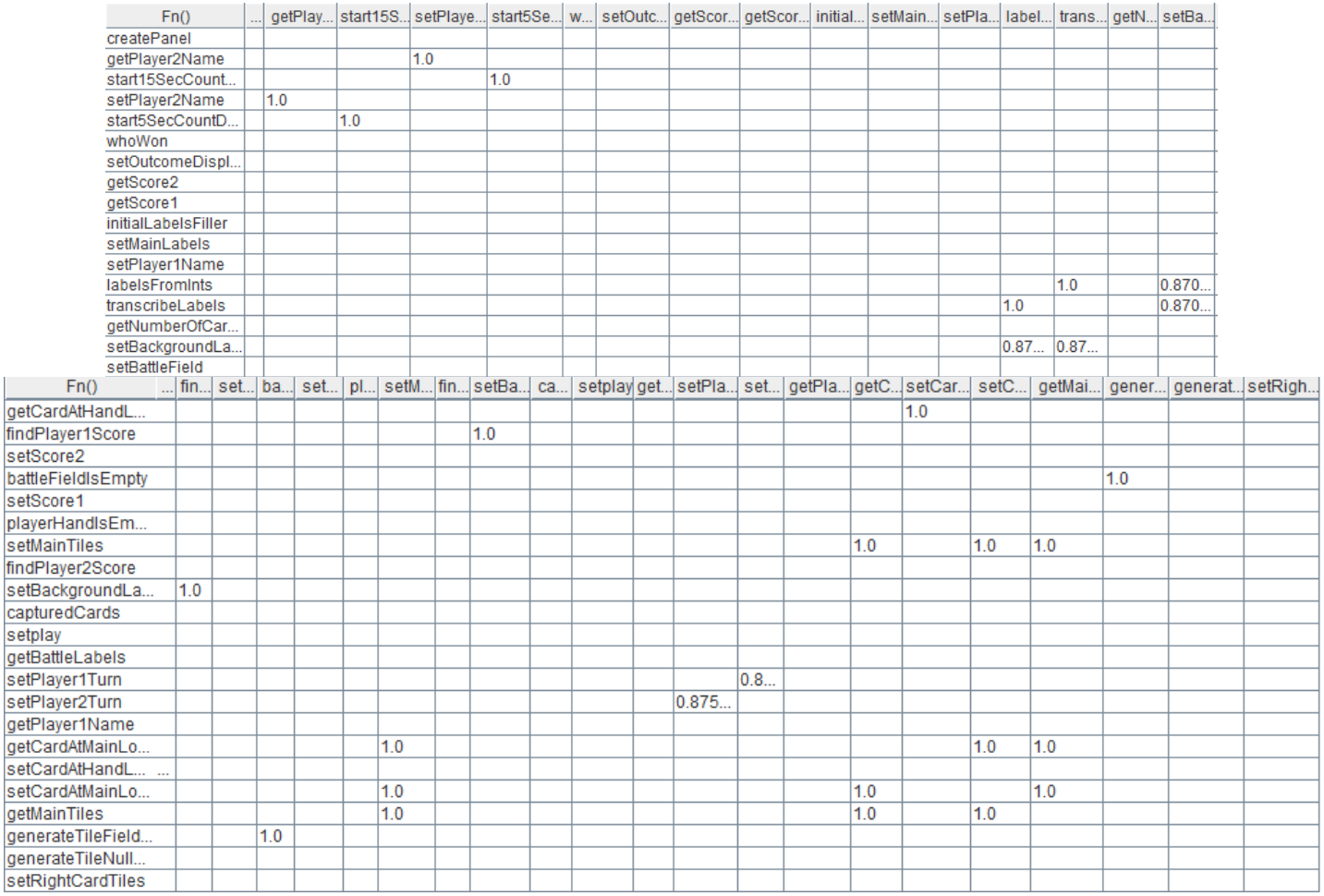
APPENDIX C2

UNFILTERED SIMILARITIES BETWEEN METHODS

CASE STUDY II

Filtered Similarity between methods based on SSM and VSM

Figures 21 and 22 represent the table view of the filtered similarities based on SSM (minCoupling = 0.82) and VSM (minCoupling = 0.78) respectively.



Fn()	...	getPlay...	start15S...	setPlaye...	start5Se...	w...	setOutc...	getScor...	getScor...	initial...	setMain...	setPla...	label...	trans...	getN...	setBa...
createPanel																
getPlayer2Name				1.0												
start15SecCount...					1.0											
setPlayer2Name		1.0														
start5SecCountD...			1.0													
whoWon																
setOutcomeDispl...																
getScore2																
getScore1																
initialLabelsFiller																
setMainLabels																
setPlayer1Name																
labelsFromInts														1.0		0.870...
transcribeLabels													1.0			0.870...
getNumberOfCar...																
setBackgroundLa...													0.87...	0.87...		
setBattleField																

Fn()	...	fin...	set...	ba...	set...	pl...	setM...	fin...	setBa...	ca...	setplay	get...	setPla...	set...	getPla...	getC...	setCar...	setC...	getMai...	gener...	generat...	setRigh...
getCardAtHandL...																	1.0					
findPlayer1Score									1.0													
setScore2																						
battleFieldIsEmpty																				1.0		
setScore1																						
playerHandIsEm...																						
setMainTiles																1.0		1.0	1.0			
findPlayer2Score																						
setBackgroundLa...	1.0																					
capturedCards																						
setplay																						
getBattleLabels																						
setPlayer1Turn														0.8...								
setPlayer2Turn													0.875...									
getPlayer1Name																						
getCardAtMainLo...							1.0											1.0	1.0			
setCardAtHandL...	...																					
setCardAtMainLo...							1.0									1.0			1.0			
getMainTiles							1.0									1.0		1.0				
generateTileField...				1.0																		
generateTileNull...																						
setRightCardTiles																						

Figure 21: SSM matrix representation, case study II, minCoupling = 0.82 – This is the matrix representation of the SSM based structural similarity between every pair of methods for the input class

Fn()	create...	start1...	start5...	whoW...	initial...	labels...	transc...	setBa...	getNu...	getCa...	setBat...	findPl...	battle...	findPl...	player...	setBa...	captu...	getBat...	setPl...	setP...	get...	setC...	set...	generat...	setR...	gen...
createPanel																										
start15SecCou...			1.0																							
start5SecCount...		1.0																								
whoWon																										
initialLabelsFiller																										
labelsFromInts							1.0	0.918...				0.918...		0.918...		0.918...	0.952...									
transcribeLabels						1.0		0.918...				0.918...		0.918...		0.918...	0.952...									
setBackground...						0.918...	0.918...							1.0			0.863...									
getNumberOfC...													0.863...								0.8...		0....			0.86...
getCardAtHand...																						1.0				
setBattleField													0.938...					0.863...			0.8...		0....	1.0		
findPlayer1Score						0.918...	0.918...									1.0	0.863...									
battleFieldIsE...									0.863...		0.938...							0.863...						0.93877...		
findPlayer2Score						0.918...	0.918...	1.0									0.863...									
playerHandIsE...																										
setBackground...						0.918...	0.918...					1.0					0.863...									
capturedCards						0.952...	0.952...	0.863...				0.863...		0.863...		0.863...										
getBattleLabels											0.863...		0.863...											0.86394...		
setPlayer2Turn																				0.92...						
setPlayer1Turn																			0.92...							
getCardAtMain...									0.863...		0.863...												1.0	0.86394...		
setCardAtHand...										1.0																
setCardAtMain...									0.863...		0.863...										1.0			0.86394...		
generateTileFi...											1.0		0.938...					0.863...			0.8...		0....			
setRightCardTi...																										
generateTileNu...									0.863...																	

Figure 22: VSM matrix representation, case study II, minCoupling = 0.78 – This is the matrix representation of the VSM based structural similarity between every pair of methods for the input class