ESTIMATION OF THE SCOPE OF CHANGE PROPAGATION IN OBJECT-ORIENTED PROGRAMS

ELMIRA RAJINIA

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN QUALITY SYSTEM ENGINEERING

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

DECEMBER 2010

Approved by:

Dr. Abdessamad Ben Hamza, Graduate Program Director,
Concordia Institute for Information Systems Engineering

Dr. Anjali Awasthi, Chairman,
Concordia Institute for Information Systems Engineering

Dr. Simon Li, Supervisor,
Concordia Institute for Information Systems Engineering

Dr. Benjamin Fung, Examiner,
Concordia Institute for Information Systems Engineering

Dr. Yuhong Yan, Examiner,
Department of Computer Science & Software Engineering

ABSTRACT

Estimation of the Scope of Change Propagation in Object-Oriented Programs

Elmira Rajinia

When minor modifications need to be made in an object-oriented computer program, they often incur further more changes due to presence of dependency in the codes and the program structure. Yet, to accommodate the required change, there can also be more than one option to carry out the initial modifications. To select the modification option in this context, this thesis proposes a systematic approach to estimate the scope of change propagation of an object-oriented program given some initial modifications.

The present Master's thesis seeks to develop an approach to predict the scope of propagated change through the entities of object-oriented software due to a modification in the software. Despite the previous works that just studied the change propagation in object oriented programs from the aspect of high level entities like classes or from the aspect of UML diagrams, we have studied the finer entities of the object oriented program and the relationships among them. In this regard, this thesis has focused on the calculation of probability of change propagation between each two specific types of entities through the analysis of dependency types among the fundamental entities of object-oriented program and categorization of existing dependencies between each couple

of entities. Then, we have defined the priority number concept as a representative scale for the scope of change propagation in software based on the probability rules.

The strategy is to first capture the dependency relationships of the entities, pertaining to an object-oriented program via the matrix representation. In this work, we have used Design Structure Matrix to capture and trace dependency among software's entities. Based on this matrix-based model, the priority number method is proposed and applied to estimate the scope of change propagation by assuming some initial modifications. The core of this method is to estimate the chance of affecting other program entities due to some modified entities and the matrix structure.

Finally, the obtained results from a case study have been tested to validate the effectiveness of the change propagation probability numbers and priority number concept.

Acknowledgments

I would like to express my gratitude to Dr. Simon Li, that this endeavour was not possible without his great support, research guidance, inspiration and patience over the past two years of my study.

It is also a pleasure to thank those who made this thesis possible such as my husband Behzad and my parents who gave me the moral support I required.

Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

Object-oriented programming is a software development paradigm that is used vastly from early 1990's. The main characteristic of object-oriented programming is the use of objects to construct a program. Particularly, an object in a program is defined by a class, which can be considered as a blueprint describing the behaviours of that object. Classes can have relationships like interface or inheritance with each other. Therefore, one can say that an Object-Oriented Program (OOP) is a collection of interactions among different objects and classes related to them. The popularity of OOPs can be attributed to some of its characteristics such as data encapsulation, inheritance, modularity, modifiability, extensibility, maintainability and re-usability (Sierra and Bates 2005).

To debug and improve an existing OOP, modifications are very common to apply during a software development process. Yet, how to apply such modifications may not be an easy task due to the presence of relationships among those classes and objects. Particularly, even though only minor modifications are made initially, such modifications can affect some related parts of the program, leading to an intractable propagation of changes. Moreover, there can be more than one option to modify the program in order to achieve the same purpose. Due to the intractable propagation, it is difficult to predict which option will lead the minimum scope of change.

Thus, the purpose of this research is to propose a systematic approach to estimate the scope of change propagation in the context of object-oriented programming. The methodical strategy is to treat an OOP as a network of interconnected program entities (e.g., objects and classes). By explicitly identifying their interrelated dependency relationships, we can analyze how initial modifications can potentially affect some other parts of the program.

1.1. Background and Motivation

After finding software's bugs and defects through techniques like program slicing (Gallagher and Lyle 2002) in testing phase of software life cycle, the software will be returned to implementation phase for change application. Even after releasing the software, it will continue to have changes in software maintenance phase.

For successful software systems, more than half of the software lifecycle costs are dedicated to software maintenance and evolution. Software maintenance is one of the phases in the software development process. One of the most important characteristics of software maintenance is making changes to software in order to eliminate the defects and deficiencies that have been found during the usage of software and also improving the software features through techniques like code refactoring (Fowler 1999; Mens and Tourwe 2004).

Due to dependency that exists among different parts of the system, when changes happen to one part of the system, it will migrate to other living parts. Thus, in order that the system can continue to work correctly, we should manipulate the affected parts of the system appropriately. Therefore, change propagation can be defined as migration of

change among different parts of the system due to change happening in a particular part. Most of the time applying change to a system is more complex that it may appear in first glance. Since usually most parts of the system are connected to each other, changes are also connected.

A simple mechanical example is that if we have a container with fixed volume, the volume factor is dependent to surface dimensions and height factors. Now, if we decrease the dimensions of the container's surface, we need to increase its height in order that the volume remains unchanged. Therefore, we can say that change has propagated from surface dimensions to height.

As an example of change propagation in software systems, we can refer to dependencies that exist among classes in OOPs. In software that is designed based on OOPs, classes may be interconnected due to interface and inheritance relationships. Any changes that happen to a class due to an internal change, like changes in methods, fields, implementation, etc., may propagate to other classes due to mentioned relationships.

The level of change propagation in systems depends on the complexity of systems. The more connections exist among parts of the system, the more complex the system will be. As an example of a complex and highly interconnected system in mechanical engineering, we can refer to helicopters. For example, any change that happens to the rotor of helicopter will cause enormous changes on the other parts of the helicopter (Clarkson, Simons and Eckert 2001).

Most of the time applying changes to the products or software will be so costly for those projects. The bigger change scope in the system will cause the higher costs for the project

up to the point that may lead to negative return of investment. Therefore, change management in projects is of great importance. For example, in the helicopter example, any changes that happen to rotor will be so costly. Therefore, changes in any parts that may affect the rotor should be avoided as much as possible.

In software projects, costs of the project are not just based on expenses. Since the project may lose its justifiability due to delay in product delivery, time should also be considered as an important factor. Applying changes for each class will be costly from time and money aspects. Therefore, for those classes of OOPs that application of change on them may have a great impact on a large number of other dependent classes, applying changes to those kinds of classes should be avoided as much as possible.

In this research, we have developed an approach through that we can minimize change propagation in software systems and also minimize the project costs. In this work, we have introduced a method to find and choose the change option that imposes the smallest scope of change to the OOP among existing change options. This task will be done through predicting the scope of change for each option.

1.2. Literature Review

In literature, some research efforts have been reported in the topic of change propagation in software. Some efforts have been devoted to the analysis of UML models (a common modeling language for object-oriented systems) to assess the change-proneness of a software system (Sharafat and Tahvildari 2007; Sharafat and Tahvildari 2008; Han et al. 2008). Sharafat and Tahvildari (2007, 2008) used dependencies obtained from the UML diagrams and some other code metrics. Then, they combined it with the change log of the

software system and the expected time of next release to assess the probability that each class will change in a future generation.

Ah-Rim Han and his team (2008) developed Behavioural Dependency Measure (BDM). They used the obtained structural information of classes and the relationships between those classes from a class diagram, the behavioural information of UML 2.0 design models and a Sequence Diagram and an Interaction Overview Diagram for capturing the behavioural aspects of the software to develop this measure. This measure is used to predict the change proneness of classes in the software.

Tsantalis et al. (2005) focused on the axes of change among classes in an object-oriented program and applied a probabilistic approach to estimate the change-proneness of classes. When change happens to a class, it can propagate to other classes through axes of change. In his work, first he tried to identify axes of change. Then, through using an improved correlation coefficient that was obtained from the calculated probabilities and actual changes for all classes and for all generations of two open source projects, he developed a measure to assess the probability that each class will change in a future generation.

Several other methods have also been proposed to predict changes in software systems, such as graph-based modeling (Rajlich 1997), a heuristic approach (Hassan and Holt 2004), an approach combining impact analysis and mining software repositories (Kagdi and Maletic 2006), and Bayesian belief networks (Mirarab et al. 2007).

Rajlich (1997) discussed about change propagation in software based on the inconsistencies that will happen to software after applying change to each entity of the software. Then, he introduced two formal models of change propagation and change-and-

fix based on graph rewriting. In these models, the software is represented as a graph of dependencies among the entities of the software. When change happens to the entities of software, the entity may no longer fit with the other entities of the software and it will lead to dependencies that are inconsistent. Each change removes some inconsistencies and also it may create new ones. This tool will help the programmer to trace the inconsistencies and changes during the process of software maintenance.

Hassan and Holt (2004) used historical co-change data to develop heuristics that predict change propagation to address the question "*How does a change in one source code entity propagate to other entities?"* and to assist developers during the change propagation process. They studied changes that happened to five large open source software systems and using the obtained dataset, first they studied several general heuristics that predict change propagation and then, they built their enhanced heuristics and measured their effectiveness in predicting change propagation.

Kagdi and Maletic (2006) focused on Mining Software Repositories (MSR), which is an approach to support software-change prediction from a historical perspective. Also, they focused on Impact Analysis (IA), which is an approach to predict the change proneness of the software through analysis of the current version of a system. Then, they compared the expressiveness and effectiveness of these two approaches to find exclusive and synergistic benefits of the two paradigms to improve software-change prediction. Through combining the two mentioned approach, they developed and evaluated a hybrid approach to produce more accurate results.

Mirarab and his team (2007) used Bayesian Belief Networks as a probabilistic tool to predict the possibly affected parts of the software due to a change that has happened in the system. Their approach is based on two main steps:

- Extracting Information step, in which they extract the existent dependencies among system elements and the change history of the software.
- Predicting Changes steps, in which they develop the Bayesian Belief Network based on the extracted information from the first step.

Then, through dividing the Bayesian Belief Network in to three different branches, which are different from the aspect of sources of used information, they made predictions using probabilistic inference.

The major difference between this work and the mentioned researches is that most of the jobs that have been done up to now is just a high level prediction of the scope of change through the architecture of the software and they rarely have gone as deep as the code level. In contrast to the above efforts, this work is intended to analyze more detailed entities (rather than just classes or UML diagrams) in the OOP. In previous researches, the scope of change can be predicted even without having the actual code and only by knowing the UML diagram of the software or being aware of the architecture of classes. But, in this research we have analysed the software up to the code level. In this method, our prediction about the scope of change cannot be completed, unless we can have access to the code details and the exact specifications of the change that is going to be applied on the code. Also, in this research we have focused on change options which are the different solutions that can address a change request. In this regard, we have predicted

and compared the scope of change that each change option imposes to the system. It should be mentioned that this method is original in this research.

The result of previous researches that have been done in this regard can be used in the design phase of the software development lifecycle and through having access to the related UML diagram or the software architecture. Despite the other works, in this method we have focused more on the details of the code. We believe that the result of this research can help the developers to have more precise estimation about the scope of change during the implementation and maintenance phases of the software.

The method of approach includes the use of matrix to capture the dependency information and the probabilistic approach to assess the scope of change propagation. This combination of the approaches is also original in this research.

1.3. Thesis Objectives and Organization

In this research, we will start through analysis of dependency modeling for OOPs in Section 2. In this section, we will classify the fundamental entities of OOP and dependencies that exist among these entities. Also, we will provide the dependency model that characterizes the relationships in OOPs. Afterwards, matrix-based modeling will be applied to represent and analyze the dependency information.

Based on the matrix-based modeling effort, Section 3 will show the quantitative procedure to estimate the scope of change propagation for OOPs. In this section, different types of change that may happen to an OOP entity will be classified. Then, the mechanism of change propagation among these entities will be discussed along with an

introduction to the meaning of change options and formulation of priority number, which is a representative scale to the scope of change propagation for each option.

Section 4 will provide a case study to evaluate and verify the proposed method in previous section and finally Section 5 will conclude this work with closing remarks.

Chapter 2

2. Dependency Modeling of Object-Oriented Programs

To analyze the change propagation process, we develop a dependency model for Object-Oriented Programs (OOP). This dependency model consists of different sets of entities that are interconnected. Accordingly, this section first discusses the OOP entities and their relationships involved in the dependency model. Then, matrix-based modeling is applied to capture the dependency information among the OOP entities.

2.1. Classification of Entities

Most of object oriented programming languages are almost the same from the aspect of involved entities and their relationships. But, in this research, we need to focus on a specific language. Therefore, we have chosen Java programming language, exclusively as the object oriented programming language to work with.

To know the connections and dependencies that exist in Java OOP, first we should know the types of entities for Java OOP. Table 1 shows the list of involved types of entities in Java and a brief description for each one. (Korn, Yih-Farn and Koutsofios, 1999)

Table 1: Java Entity Types and Their Descriptions

Class	Classes are fundamental building blocks of an OOP and they contain declarations and definitions of a collection of methods and fields.
Method	Method is an operation of a particular object or it is a function that is part of a class.
Field	Field is a variable or constant that is part of a class.
Interface	An interface is a named collection of method definitions (without implementations). An interface can also declare constants. Classes implement the declarations of zero or more interfaces.
Package	A set of classes and interfaces.

Each software program may contain zero, one or more from each specified type. In this research, we will dedicate a set to each type. This set will contain the entities related to that type. For example, if the program contains three classes that are called C1, C2 and C3, the set related to Class type will be Class={C1, C2, C3}. To make this concept clearer, we will introduce a simple example at this stage and then we will continue through that example.

The example program is a simple object oriented program that is obtained from Sierra and Bates (2005), and it is a guess-the-numbers game. The game starts from generating three random integers between 1 and 9. Then, the player is asked to guess the values of these three numbers. At the end, the program will return the number of guesses that the player has made.

The code for this short program is shown in Appendix A for reference. Through analysis of this program, 25 OOP entities are found. The list of these entities can also be found in Appendix A. The following sets show the classification of entities for this sample OOP program.

Class= { SimpleDotCom, GameHelper}

Method= {main, setlocationCells, checkYourself, getUserInput}

Field= {numOfGuesses, helper, theDotCom, randomNum, locations, isAlive, guess, result, locationCells, numOfHits, locs, stringGuess, guess, result, cell, prompt, inputLine, is}

Interface= {}

Package= {java.io}

2.2. Classification of Dependencies

In OOP, entities are related to each other through different types of relationships. Although the practice of object-oriented programming has the intention to minimize the dependency among program entities, (as compared to procedural programming), the dependency cannot be entirely avoided in order to support the functions of the program.

If we demonstrate the relationships among the entity types, which are described in Table 1 in a high level demonstration, we will have the hierarchy model in Figure 1. As it is shown in this model, we have *field* and *method* in the lowest level of hierarchical structure. At the upper level of the hierarchical structure, both *class* and *interface* consist of *field* and *method.* At the end, *package* contains *classes* and *interfaces.*

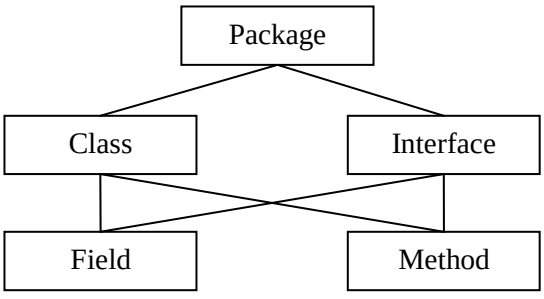

Figure 1: Hierarchy of OOP Entities

To understand the connections that exist between each two OOP entities, we should know the dependency relationships that are among these entities. As it is discussed before, in this research, five types of dependency relationships are characterized and they are listed in Table 2. In this table, signs (→ and ↔) show the direction of dependency between two entities. In the change propagation process, any modifications of one entity in OOP, may lead to change of other OOP entities through one of the mentioned dependency relationships.

Table 2: OOP Dependency Relationships

OOP Relationships	Descriptions
Call	➢ A method use a predefined method in its body *(Method → Method)* ➢ A method use a predefined field in its body *(Method → Field)* ➢ A field is equal to return value of a method *(Field → Method)* ➢ A field is equal to combination of other fields *(Field → Field)* ➢ A class use the contents of another package in its body *(Class → Package)* ➢ A method use the contents of another package in its body *(Method → Package)*
Inheritance	➢ Class A has an inheritance relationship with class B when one of them extends another one. *(Class → Class)*
Interface	➢ A Class has interface relationship with an interface when that class implements that interface. *(Class ↔ Interface)* ➢ When a class implements an interface the peer methods will be related to each other through interface relationship *(Method ↔ Method)*
Composition	➢ A field is defined as an object of a class in the body of a method or in the body of another class *(Field → Class)* ➢ A method use another method of a class through an object of that class in its body *(Method → Method)* ➢ A method use a field of a class through an object of that class in its body *(Method → Field)* ➢ A field is equal to return value of a method from another class through an object of that class *(Field → Method)*
Definition	➢ A method is defined in the body of a class *(Class → Method)* ➢ A field is defined in the body of a class *(Class → Field)* ➢ A field is defined in the body of a method *(Method → Field)* ➢ An interface is defined in the body of a package *(Package → Interface)* ➢ A class is defined in the body of a package *(Package → Class)*

Through application of mentioned OOP relationships in Table 2 to mentioned OOP entities in Table 1, we can define different types of dependencies among those entities. Let's consider A and B as two different entity types and we suppose that relationship D is the existed relationship among A and B. Therefore, we can define *D(A,B)* as the dependency of entity type A to entity type B through relationship D and it will be formulated as $D(A,B) \subset A \times B$, that means any entity of type A can be dependent to any entity of type B through relationship D.

Table 3 has listed most of common and possible dependencies between each two types of entities through the mentioned relationships along with a brief description for each one. In this table, *C* stands for Class, *M* stands for Method, *F* stands for Field, *I* stands for Interface and *P* stands for Package.

Back to the described example in Section 2.1 and by referring to the list of the sample OOP entities, which is cited in Appendix A, Table 4 has listed dependency relationships among the entities of sample program and a brief description for each of them.

Table 3: Dependencies among OOP Entities

Dependency Types	Sub-Types	Description
Definition (De)	De(C, M) $\subset C \times M$	The methods that have been defined in a class
	De(C, F) $\subset C \times F$	The fields that have been defined in a class
	De(M, F) $\subset M \times F$	The variables or constants that have been defined in a method
	De(P, I) $\subset P \times I$	The interfaces that have been defined in a package
	De(P, C) $\subset P \times C$	The classes that have been defined in a package
Call (Ca)	Ca(M, M) $\subset M \times M$	A method that uses other methods from the same class in its body
	Ca(M, F) $\subset M \times F$	A method that uses other variables from the same class in its body
	Ca(F, M) $\subset F \times M$	A field that is equal to return value of a method
	Ca(F, F) $\subset F \times F$	A field that is equal to the combination of other fields
	Ca(C, P) $\subset C \times P$	Classes that use classes or interfaces from other packages in their body
	Ca(M, P) $\subset M \times P$	Methods that use classes or interfaces from other packages in their body
Composition(Co)	Co(F,C) $\subset F \times C$	A variable that is defined in form of an object from a specific class
	Co(M,M) $\subset M \times M$	A method that uses other methods from different classes in its body through the objects of those classes
	Co(M,F) $\subset M \times F$	A method that uses other variables from different classes in its body through the objects of those classes.
	Co(C,F) $\subset C \times F$	The fields that have been defined in a class in form of objects
Interface(Int)	Int(C,I) $\subset C \times I$ Int(I,C) $\subset I \times C$	A class that implements an interface
	Int(M_1,M_2) $\subset M_1 \times M_2$ Int(M_2,M_1) $\subset M_2 \times M_1$	The methods that have been defined in an interface and the methods that have been defined in the correspondent class that implement the interface
Inheritance(Inh)	Inh(C,C) $\subset C \times C$	A class that extends another class
	Inh(I,I) $\subset I \times I$	An interface that extends another interface

Table 4: Description of Dependencies among Example Program's Entities

$De(M_1, F_1)$	Variable (numOfGuesses) is defined in (main) method.
$De(M_1, F_2)$	Object (helper) is defined in (main) method.
$De(M_1, F_3)$	Object (theDotCom) is defined in (main) method.
$De(M_1, F_4)$	Variable (randomNum) is defined in (main) method.
$De(M_1, F_5)$	Variable (locations) is defined in (main) method.
$De(M_1, F_6)$	Variable (isAlive) is defined in (main) method.
$De(M_1, F_7)$	Variable (guess) is defined in (main) method.
$De(M_1, F_8)$	Variable (result) is defined in (main) method.
$Co(M_1,M_2)$	Method (main) uses (setlocationCells) method from (SimpleDotCom) class in its body through an object of that class
$Co(F_2,C_2)$	Variable (helper) is defined in form of an object from (GameHelper) class
$Co(F_3, C_1)$	Variable (theDotCom) is defined in form of an object from (SimpleDotCom) class
$Ca(F_5, F_4)$	Field (locations) is equal to the combination of other fields (randomNum)
$Ca(F_7, M_4)$	Field (guess) is equal to return value of method (getUserInput)
$Ca(F_8,M_3)$	Field (result) is equal to return value of method (checkYourself)
$De(C_1, F_9)$	Field (locationCells) is defined in class (SimpleDotCom),
$De(C_1,F_{10})$	Field (numOfHits) is defined in class (SimpleDotCom)
$De(C_1, M_2)$	Method (setlocationCells) is defined in class (SimpleDotCom)
$De(C_1, M_3)$	Method (checkYourself) is defined in class (SimpleDotCom)
$Ca(M_2, F_9)$	Method (setlocationCells) uses variable (locationCells) from the same class in its body
$De(M_2, F_{11})$	Variable (locs) is defined in method (setlocationCells)
$Ca(M_3,F_9)$	Method (checkYourself) uses variable (locationCells) from the same class in its body
$Ca(M_3, F_{10})$	Method (checkYourself) uses variable (numOfHits) from the same class in its body
$De(M_3,F_{12})$	Variable (stringGuess) is defined in method (checkYourself)
$De(M_3, F_{13})$	Variable (guess) is defined in method (checkYourself)
$De(M_3, F_{14})$	Variable (result) is defined in method (checkYourself)
$De(M_3, F_{15})$	Variable (cell) is defined in method (checkYourself)
$Ca(F_{13}, F_{12})$	Field (guess) is equal to the combination of other field (stringGuess)
$Ca(F_{15}, F_9)$	Field (cell) is equal to the combination of other field (locationCells)
$Ca(C_2,P_1)$	Class (GameHelper) uses classes from package (java.io) in its body
$De(C_2,M_4)$	Method (getUserInput) is defined in class (GameHelper)
$De(M_4,F_{16})$	Variable (prompt) is defined in method (getUserInput)
$De(M_4, F_{17})$	Variable (inputLine) is defined in method (getUserInput)
$De(M_4, F_{18})$	Object (is) is defined in method (getUserInput)

2.3. Matrix Representation

Design Structure Matrix (DSM) has been a common tool to represent the dependency information in systems modeling and engineering design. Examples include design structure matrix (Browning 2001) and domain mapping matrix (Daniovic and Bronwing 2007). Furthermore, DSM has been used to facilitate the control of change propagation in complex designs (Chen et al. 2007; Li and Chen 2010). In this research, we intended to continue these efforts by using DSM as a tool to represent the interactions and interdependencies among object oriented program's elements and extending the matrix-based techniques to control the change propagation in OOP.

DSM is a tool that provides a compact and clear view of dependencies in systems and simplifies analysing and management of complex systems. DSM is basically a square matrix that relates entities of one kind to each other. DSM can be binary or numerical and at the same time, it can be directed or non-directed. Binary DSM only represents the existence of relationship between every two entities. Numerical DSM also represents the strength or weight of each relationship. In non-directed DSM, since dependencies do not have specific directions, the DSM is a symmetric matrix or it contains two identical upper triangle and lower triangle parts. Unlike non-directed DSM, in directed DSM, each relationship has its specific direction. Therefore, in directed DSM, existence of dependency between two entities in a specific direction will not guarantee the existence of dependency in the opposite direction between those two entities necessarily. In DSM, the dependency of an entity to itself is not allowed. The DSM that we use in our work is a directed binary DSM.

Towards the matrix representation effort, an object-oriented program is first analyzed to identify the OOP entities that are involved. (The OOP entities are described in Table 1.) All these entities are then labelled and represented in a DSM's rows and columns. If a program has *n* OOP entities, the resulting matrix's dimension is *n*-by-*n*. Each DSM entry represents the dependency relationship of the corresponding two entities. For instance, if *entity i* depends on *entity j*, the corresponding matrix entry denoted as m_{ij} is marked to indicate the presence of such dependency. It should be noted that the OOP relationships are directional. Thus, the resulting matrix is not necessarily symmetric.

To illustrate, suppose that we have six different entities {*a, b, c, d, e, f*}, where *a* is a *field* entity; *b, c* are *method* entities; *d, e* are *class* entities, and *f* is an *interface* entity. The relationships of these entities are represented in a matrix in Figure 2. For instance, the matrix entry m_{bc} shows that *method entity b* calls *method entity c* (i.e., call relationship). In turn, by checking the column of *c*, the presence of m_{bc} implies that any change of *c* may lead to the change of *b*. Similarly, the matrix entry m_{de} represents the inheritance relationship that *class entity d* inherits *class entity e* (*e* is parent of *d*). Both matrix entries m_{ef} and m_{fe} represent the *reciprocal* interface relationship between *e* and *f*. The matrix entry m_{ad} shows that *field entity a,* that is a reference to an object, composes *class entity d* (i.e., composition relationship). Lastly, the matrix entry m_{ea} shows that *class entity e* defines *field entity a* (i.e., definition relationship).

	a	*b*	*c*	*d*	*e*	*f*
a				m_{ad}		
b			m_{bc}			
c						
d					m_{de}	
e	m_{ea}					m_{ef}
f					m_{fe}	

Figure 2: Sample Matrix to Show Different OOP Relationships

Therefore, if we know all the dependency relationships that exist among the entities of an OOP, we can create the DSM related to that program. Then, if we accept Table 4 as the existing dependency relationships among the entities of our sample program, Figure 3 can be considered as its correspondent DSM.

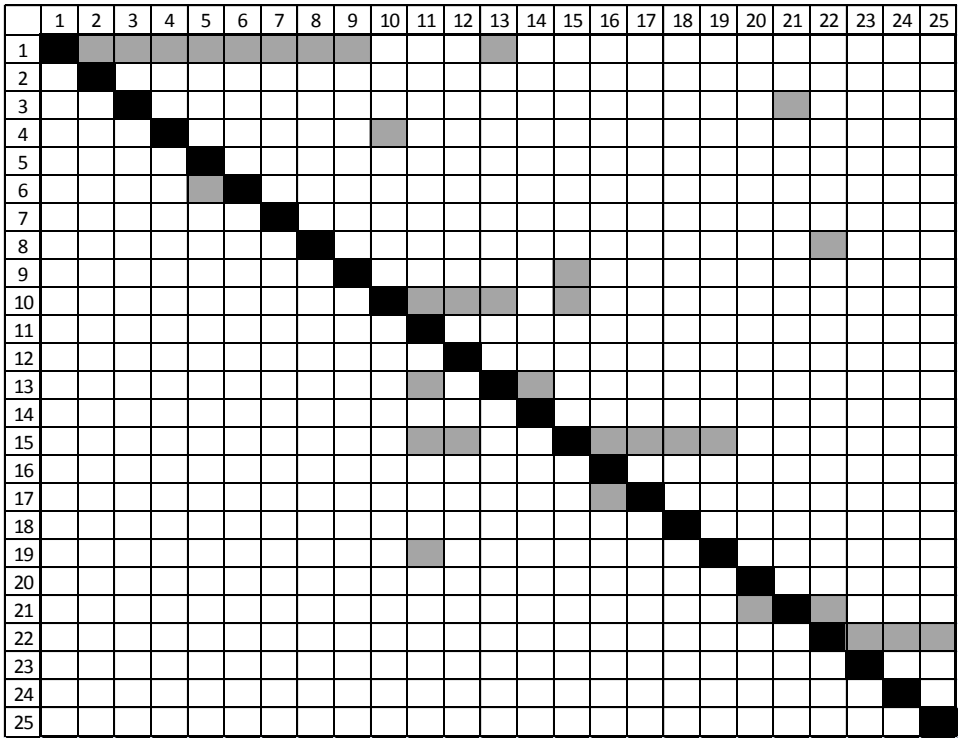

Figure 3: Dependency Matrix of the Example Program

Chapter 3

3. Estimation of Change Propagation

Estimation of the scope of change propagation in software is an important issue in software's change management. Applying changes in software always consumes time and budget. Therefore, predicting and minimizing the scope of change is of great importance.

Changes that may happen to software are result of existence of a bug in the software or need for an improvement or adding a new feature to the software or a new customer request. Since the entities of object oriented software have dependency relationships with each other, changes that happen to one of software's entities may propagate to other entities of the software. Therefore, in farther levels of change propagation, a large scope of software may go under changes. Since the size of change scope has direct relationship with cost and time, finding a method to minimize the scope of change can be helpful in saving time and money.

In this section, we have calculated the probability of change propagation among different entity types through classification of entity changes and analysis of the dependencies that exist among each two types of entities. Then, we will calculate priority number that can

be considered as a representative scale for the scope of change propagation after change application.

3.1. Classification of Changes of Entities

To be able to study changes of software's entities more closely, we should have a precise idea about the type of changes that may happen to each type of entity in OOP. By change type, we mean the actual change that may happen to an entity. For example, the entity type method can be changed through changing its body, passing arguments, return type and etc., that each one can be considered as a type of change.

Table 5: Change Types

Change Type 1 (CT1)	Change in name
Change Type 2 (CT2)	Change in access level(public, protected and private)
Change Type 3 (CT3)	Change in being final or not
Change Type 4 (CT4)	Change in being abstract or not
Change Type 5 (CT5)	Change in type (primitive or user-defined)
Change Type 6 (CT6)	Change in passing arguments
Change Type 7 (CT7)	Change in per-instance or per-class (*static*)
Change Type 8 (CT8)	Change in being constant or variable
Change Type 9 (CT9)	Change in return type (primitive, user-defined, or *void*)
Change Type 10 (CT10)	Change in body
Change Type 11 (CT11)	Add or delete field
Change Type 12 (CT12)	Add or delete method
Change Type 13 (CT13)	Add or delete class

Table 5 has listed the change types that may be cause of change for an entity type. Note that each change type may be or may not be applicable to a specific type of entity. For example, changes in passing arguments can be applicable to entity type “method” but it is meaningless for entity type “field”.

3.2. Change Propagation Mechanism

In OOP, changes start through applying a change type (Table 5) to an entity type (Table 1). Then, it will propagate to another entity through a dependency relationship (Table 3). For each specific change path, the probability of change for the second entity may be 100% (under the situation that we can be sure about changing the second entity) or 50% (under the situation that we cannot be sure about changing the second entity). For example, if we change the name of a class (CT1 × Class), due to dependency relationship Co(F,C), the definition of all the objects that are defined from that class should be changed with the probability of 100 %. Now, suppose that we change the type of return value in a method (CT9 × Method). Due to dependency relationship Ca(F, M), the fields that are equal to return value of that method, may or may not have to be changed. Since we cannot claim the change until we know the exact conditions of the real code, the probability of change for the second entity in this situation is 50%.

Table 6 has listed the change paths in OOP and the probability of change for each affected entity.

Table 6: Change Paths in OOP

Initial Entity	Change Type	Dependency Relationship	Affected Entity	Probability
Method	CT1	Ca(F, M)	Field	100
		Ca(M, M)	Method	50
		Co(M,M)	Method	50
		Int(M,M)	Method	100
	CT9	Ca(F, M)	Field	50
		Ca(M, M)	Method	50
		Co(M,M)	Method	50
		Int(M,M)	Method	100
	CT6	Ca(F, M)	Field	50
		Ca(M, M)	Method	50
		Co(M,M)	Method	50
		Int(M,M)	Method	100
		De(C, M)	Class	50
	CT2	Ca(F, M)	Field	50
		Co(M,M)	Method	50
		Int(M,M)	Method	100
	CT10	Ca(F, M)	Field	50
		Ca(M, M)	Method	50
		Co(M,M)	Method	50
		De(C, M)	Class	50
	CT7	Ca(F, M)	Field	100
		Ca(M, M)	Method	50
		Co(M,M)	Method	50
		Int(M,M)	Method	100
	CT4	De(C, M)	Class	50
		Ca(F, M)	Field	100
		Ca(M, M)	Method	100
		Co(M, M)	Method	100
	CT12	Ca(M, M)	Method	100
		Co(M, M)	Method	100
		Ca(F, M)	Field	100
		Int(M,M)	Method	100
Field	CT1	Ca(M, F)	Method	100
		Co(M,F)	Method	100
		Ca(F, F)	Field	100
		De(M, F)	Method	100
	CT5	Co(C,F)	Class	50

		De(C, F)	Class	50
		Ca(M, F)	Method	50
		Co(M,F)	Method	50
		De(M, F)	Method	50
	CT2	Co(C,F)	Class	50
		Co(M,F)	Method	50
	CT7	Ca(M, F)	Method	50
		Co(M,F)	Method	50
		Ca(F, F)	Field	50
	CT8	Ca(M, F)	Method	50
		Co(M,F)	Method	50
		De(M, F)	Method	50
	CT11	Ca(F, F)	Field	100
		Ca(M,F)	Method	100
		Co(M,F)	Method	100
Class	CT1	Co(F,C)	Field	100
		Inh(C,C)	Class	100
		Int(I,C)	Interface	100
		De(P, C)	Package	100
	CT2	Co(F,C)	Field	50
		De(P, C)	Package	50
	CT3	Inh(C,C)	Class	100
	CT4	Co(F,C)	Field	100
	CT12	De(P, C)	Class	100
		Inh(C,C)	Class	50
		Int(I,C)	Interface	50
	CT11	Inh(C,C)	Class	50
	CT13	De(P, C)	Package	100
		Co(F,C)	Field	100
		Inh(C,C)	Class	50
Interface	CT12	Int(C,I)	Class	100
		De(P, I)	Package	100
	CT11	Inh(I,I)	Interface	50
	CT1	Int(C,I)	Class	100
		Inh(I,I)	Interface	50
		De(P, I)	Package	100
Package	CT1	Ca(M, P)	Method	100
		Ca(C, P)	Class	100

At this stage, by knowing the probability of change propagation among each two specific entities through each particular dependency relationship, we can calculate the probability of change propagation between those two entities by calculating the average of probability numbers through all different dependency relationships. For example, for the probability of change propagation among package and class we have:

Class × CT1 → De(P, C)→ Package (100%)

Class × CT2 → De(P, C)→ Package (50%)

Class × CT12 → De(P, C)→ Package (100%)

Class × CT13 → De(P, C)→ Package (100%)

Package × CT1 → Ca(C, P)→ Class (100%)

Therefore, the probability of change propagation among package and class will be (100+100+100+100+50)/5=90%

In a same way, we can calculate the probability of change propagation among other entities. Table 7 has listed the probability of change propagation between each two specific entities.

Table 7: Change Propagation Probability among Entities

Class - Method	50%	Field-Field	83.33%
Class-Field	71.42%	Class-Interface	87.5%
Field-Method	70.45%	Package-Class	90%
Method-Method	73.80%	Package-Interface	100%
Class-Class	70%	Interface-Interface	50%

3.3. Change Options

Change options can be defined as different ways to address the change request. For example, to decrease the capacity of a box, at least we can have seven different methods.

- ✓ Decrease width
- ✓ Decrease length
- ✓ Decrease height
- ✓ Decrease width and decrease length
- ✓ Decrease width and decrease height
- ✓ Decrease length and decrease height
- ✓ Decrease width and decrease length and decrease height

Each mentioned method can be an option to lead to a same result that is reducing the box capacity. Now, assume that the mentioned box is part of a bigger system. In this situation, choosing each option can have a different impression on the system. In a same way, for software programs to reach to a same result from a change request, we may have more than one option. Also, each option may have different impression on the other parts of the software and as a result, may have different change scopes.

To make this concept clearer, we will continue by going through the mentioned example in Section 2.1. In this program there is a need for change. The reason of change in this program is existence of a bug in the code and we need to modify the program to correct it. Particularly, the bug lies in the lines 12-14 in the class SimpleDotCom (see Appendix for the details of the codes). Due to this bug, the program will increase the number of hits every time the user guesses one of the generated numbers, even if that number had already been guessed. To remove this bug, we need to distinguish the case if the user has repeatedly guessed the same number generated by the program. If this is the case, the

program will not count it as a new hit. To correct this bug, four change options are proposed (Sierra and Bates 2005).

In option 1, a second array is created. At each time when the user makes a guess, the modified program stores the guessed number in the second array. When the user makes another guess, the program will check the second array for any repeated guesses. Figure 4 shows part of the modified program according to this option (Changed lines are highlighted).

```
Public class SimpleDotCom{
   Int[] locationCells;
   Int numOfHits=0;
   boolean[] hitCells=new Boolean[3];
   hitcells[1]=false;
   hitcells[2]=false;
   hitcells[3]=false;
   Public void setLocationCells (int[] locs){
      locationCells=locs;
      }
  Public string checkyourself(string stringGuess){
     Int guess= Integer.parseInt(stringGuess);
     String result="miss";
     For (int i=0;  i<3;  i++){
           Int Cell= locationCells[i];
           If (guess==cell){
              If (hitCells[i]==false){
               result="hit";
               numOfHits++;
               hitCells[i]=true;
               break;
             }
        }
     }
     If (numOfHits==locationCells.length){
          Result="kill";
        }
     System.out.println(result);
       Return result;
        }
}
```

Figure 4: Modified Codes Based on Change Option 1

In option 2, the original array (i.e., locationCells) would be kept, but the values of any correctly guessed numbers in the array would be changed to -1. In this option, there is only one array to check and manipulate. Since the user is only looking for non-negative numbers in the locationCells array, a negative value (i.e., -1) at a particular location of

this array means that the number in that location has already been guessed. Figure 5 shows part of the modified program according to this option (Changed lines are highlighted).

```
Public string checkyourself(string stringGuess){
     Int guess= Integer.parseInt(stringGuess);
     String result="miss";
     For (int i=0;  i<3;  i++){
          Int Cell= locationCells[i];
          If (guess==cell){
                    result="hit";
              numOfHits++;
              locationCells[i]= -1;
              break;
            }
        }
    If (numOfHits==locationCells.length){
         Result="kill";
        }
    System.out.println(result);
      Return result;
       }
```

Figure 5: Modified Codes Based on Change Option 2

In option 3, the location of each number that is guessed correctly, should be removed from the locationCells array and the array should be modified to a smaller one. Since the size of an array cannot be changed, we have to make a new array and copy the remaining cells from the old array to the new and smaller array. Figure 6 shows part of the modified program according to this option (Changed lines are highlighted).

Option 4 is similar to option 3, but the difference is use of Arraylist from the Java library. The Arraylist acts like an array in Java, but it has new features. One of these features is that it can shrink when we remove any cells from it. Therefore, we do not have to make a new array and copy the remaining cells from the old array to the new one (as the case in the option 3). Figure 7 shows part of the modified program according to this option (Changed lines are highlighted).

```
Public class SimpleDotCom{
   Int[] locationCells;
   Int numOfHits=0;
   Int n=3;
   Int length;
      Public void setLocationCells (int[] locs){
      locationCells=locs;
      length= locationCells.length;
      }
  Public string checkyourself(string stringGuess){
     Int guess= Integer.parseInt(stringGuess);
     String result=”miss”;
     For (int i=0;  i<n;  i++){
          Int Cell= locationCells[i];
          If (guess==cell){
            n=n-1;
            int k=0;
            int[] sub=new int[n];
             for (int m=0; m<=n; m++)
                  If (m!=i ){
                        sub[k]= locationCells[m];
                        K++;
                  }
              locationCells=sub;
              result=”hit”;
              numOfHits++;
              break;
            }
      }
    If (numOfHits= =length){
         Result=”kill”;
       }
    System.out.println(result);
      Return result;
       }
}
```

Figure 6: Modified Codes Based on Change Option 3

```
      Import java.util.ArrayList;
Public class SimpleDotCom{
   Private ArrayList<String>  locationCells;
   Public void setLocationCells (ArrayList<String> loc){
      locationCells=locs;
      }
  Public string checkyourself(string stringGuess){
     String result=”miss”;
     Int index=locationCells.indexOf(stringGuess);
    if (index>=0){
          locationCells.remove(index);
          if (locationCells.isEmpty()){
             result=”kill”;
           }else{
             result=”hit”;
          }
    }
        Return result;
          }
    }
```

Figure 7: Modified Codes Based on Change Option 4

3.4. Formulation of Priority Number

Priority number can provide a scale for comparison among the scopes of change that different change options impose to the software system. The purpose of priority number in this context is to estimate the change propagation scope for each change option. The basic concept behind the computation of priority number is to calculate the expected number of modified entities for each change option. The change option that has the smallest value of the priority number will be considered as having a smallest scope of change propagation. To calculate the priority numbers, first we should determine the change proposals (options) and identify the target entities for initial modifications. By target entities, we mean the entities that will go under changes in the first level of change propagation for each change option. Target entities are not the result of change in any other entities. It means that, change in target entities is independent from change in any other entities. After determining the target entities for each change option and by using the dependency matrix, we are able to trace the change propagation paths and estimate the number of potentially affected entities. In practice, if the entities are located farther on a propagation path (or at a higher propagation level), they have less chance to be actually affected after the implementation of all the changes. In this case, we use a probabilistic approach to capture this information.

Without losing the generality of the approach, suppose that we have a set of OOP entities as $\{a, b, c, d\}$, and these entities are interrelated. In a change option, the initial modifications are clearly defined in a sense that we know which entities must be changed initially. Let us denote this kind of entities as target entities. Then, we denote $P(a)$ as the probability of changing the entity a due to the implementation of the change option. If

the entity *a* is a target entity, then $P(a) = 1$ to imply that the entity *a* must be changed in the change option.

In a change option, suppose that the entities *a* and *b* are target entities. Changing the entity *a* will potentially change the entity *c*. To capture the chance of propagation from one entity to another, we denote $P(c \mid a)$ as the propagation probability that the entity *c* will be changed due to the change of the entity *a* (or in a condition that the entity *a* has been changed).

In practice, the modifications of entities can propagate in a path. Suppose that changing the entity *a* will potentially change the entity *c*, which will in turn potentially change the entity *d*. Intuitively, this propagation path can be represented as $a \rightarrow c \rightarrow d$. Then, the probability of changing the entity *d* due to the change of the entity *a* can be computed as follows.

$$P_c(d \mid a) = P(d \mid c) * P(c \mid a) * P(a) \qquad (1)$$

Where the subscript is used to indicate any intermediate entities on the propagation path.

Based on this derivation, we can further define the direct and indirect propagation probabilities. The direct propagation probability is referred to as the probability that the change propagation takes place due to a direct relationship (e.g., $a \rightarrow c$ described above). The direct propagation probability should be determined based on the application's context. That is, as we understand the nature of the entities and their relationships, we should estimate how likely that changes on an entity will affect another one directly. In contrast, the indirect propagation probability is calculated from the direct propagation

probability. In the above formulation, $P(d \mid a)$ is indirect in a sense that its value can be determined via the direct probabilities, i.e., $P(d \mid c)$ and $P(c \mid a)$.

Also, the same entity can be affected by different target entities or from different paths of change propagation. For instance, in addition that the entity *d* can be affected by the target entity *a* (as shown above), the entity *d* can also be affected by the target entity *b* (i.e., $b \rightarrow d$). Then, the probability of changing the entity *d* due to change of the entities *a* and *b* can be computed as follows.

$$P(d \mid a \text{ and } b) = P(d \mid a) + P(d \mid b) - P(d \mid a) * P(d \mid b) \qquad (2)$$

For simplicity, we denote $P(d)$ as the probability that the entity *d* will be changed due to all initial modifications defined in a change option. Given that there are *n* entities in a program $\{a_1, a_2, \ldots, a_n\}$, the priority number (denoted as *PN*) of the change option 1 can be computed as follows.

$$PN(\text{option 1}) = \sum_{i=1}^{n} P(a_i) \qquad (3)$$

$P(a_i)$ can be considered as the probability that the entity a_i will be changed for a given change option. Then, the summation of these probabilities in the computation of *PN* can be viewed as the expected number of entities to be changed given a change option. If several change options are present, we can estimate the scope of change propagation for each option. The change option that has the smallest priority number (i.e., *PN*) will be selected for implementation after considering the detailed specifications of that option.

In this level of research we set the direct propagation probability equal to 0.5 for simplicity. That is, we have a half chance that the change of one entity will directly and

actually change another entity. Another assumption in this demonstration is to limit the length of the change propagation paths up to three levels.

To make the concept of Priority Number clearer, again we will continue through the example that is introduced in Section 2.1. In Section 3.3, we described the bug that exists in the mentioned OOP and also we presented four different options to eliminate this bug. Now, we are going to evaluate each of these options from change propagation aspect. For this evaluation, we need to know the Priority Number for each option. To calculate the priority numbers, we need to know the target entities for each option.

In option 1 as seen in Figure 4, we need to change the entity 15 (i.e., CheckYourSelf method) and the functionality of this method has been changed by adding new lines of code. Also, the entity 10 (i.e., SimpleDotCom class) has been changed by adding new variable (i.e., hitCells). Then, the initial changes of this option belong to the entities 10 and 15.

In option 2 as seen in Figure 5, we only need to change the entity 15 (i.e., CheckYourSelf method) and the functionality of this method has been changed by adding new lines of code. Then, the initial change of this option only belongs to the entity 15.

In option 3 as seen in figure 6, we need to change the entity 15 (i.e., CheckYourSelf method) and the functionality of this method has been changed by adding new lines of code. Also, the entity 10 (i.e., SimpleDotCom class) has been changed by adding new variables (i.e., n and length). Since the entity 13 (i.e., setLocationCells method) is doing something new in addition to its previous task, it has been changed by adding new lines of codes as well. Also the entity 11 (i.e., locationCells) will be changed by being

assigned a new array (i.e., sub). Then, the initial changes of this option belong to the entities 10, 11, 13 and 15.

In option 4 as seen in figure 7, this option requires a change to entity 15 (i.e., CheckYourSelf method), and the functionality of this method is changed by adding new lines of codes. Also, the entities 12 (i.e., numOfHits), 17 (i.e., guess) and 19 (i.e., cell) have been deleted. The entities 14 (i.e., locs) and 11(i.e., locationCells) have been modified since they have been changed from int[] to ArrayList<string>. Then, the initial changes of this option belong to the entities 11, 12, 14, 15, 17and 19.

As discussed, the change option 1 has the target entities 15 and 10. Thus, $P(15) = P(10) = 1$. By checking the column 15 of the dependency matrix in Figure 3 of Section 2.3, it is found that any change of entity 15 will potentially change entities 9 and 10. Similarly, entity 10 will potentially change entity 4. This matrix-based propagation tracing can be conducted similarly for the potentially-affected entities (i.e., the entities 9 and 4). Figure 8 shows the propagation paths of the change option 1 up to three levels.

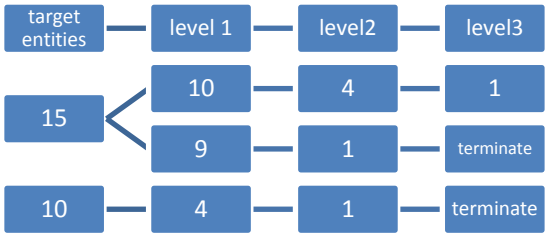

Figure 8: Change Propagation Paths for Option 1

By considering the assumption that the direct propagation probability is equal to 0.5, we can set, for example, $P(10|15) = P(9|15) = P(4|10) = 0.5$. To determine the probability of changing the entity 1 in this change option (i.e., $P(1)$), we first identify the propagation

paths that involve the entity 1. As such, we can determine the following probabilities based on different propagation paths.

$P_{10,4}(1|15) = P(1|4)*P(4|10)*P(10|15) = 0.5*0.5*0.5 = 0.125$ (the first path)

$P_9(1|15) = P(1|4)*P(4|15) = 0.5*0.5 = 0.25$ (the second path)

$P_4(1|10) = P(1|4)*P(4|10) = 0.5*0.5 = 0.25$ (the third path)

To determine the value of $P(1)$, the above probabilities need to be combined by referencing the formulation (2). For simplicity, let $P_{10,4}(1|15) = x$, $P_9(1|15) = y$ and $P_4(1|10) = z$. Then, the calculation of $P(1)$ is shown as follows.

$P(1) = x + y + z - x*y - x*z - y*z + x*y*z = 0.508$

The similar calculation can also be applied to the entities 4 and 9 to determine $P(4)$ and $P(9)$, respectively. As a result, the priority number of the option 1 can be calculated as follows.

$$PN(\text{option 1}) = \sum_{i=1}^{n} P(a_i) = P(15)+P(10)+P(9)+P(4)+P(1)=1+1+0.5+0.625+0.508 = 3.633$$

In change option 2, the only target entity is the entity 15 .Figure 9 shows the change propagation paths of this option up to three levels.

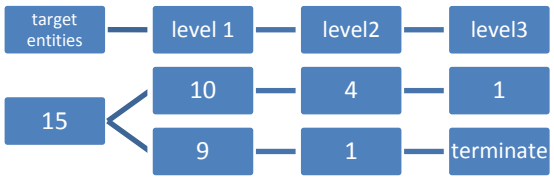

Figure 9: Change Propagation Paths for Option 2

With the same computations as option 1, we will have the following change probability number for each involved entity in option 2:

P(15)=1 , P(10)=0.5 , P(4)=0.25 , P(9)=0.5 , P(1) =0.344

Therefore, for this option we will have the Priority number equal to:

Priority Number=1+0.5+0.25+0.5+0.344= 2.6

In the change option 3, the target entities are entities 15, 13, 10 and 11 .Figure 10 shows the change propagation paths of this option up to three levels.

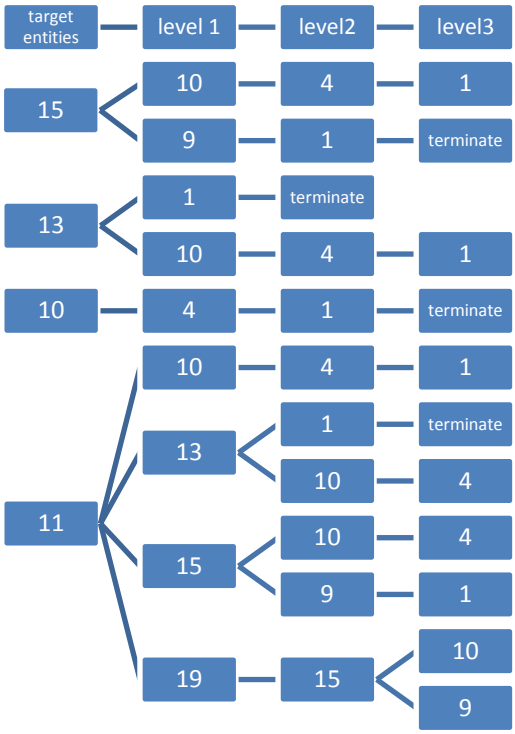

Figure 10: Change Propagation Paths for Option 3

Again with the same computations, we will have the following change probability number for each involved entity in option 3:

P(15)=P(10)=P(11)=P(13)=1, P(9)=0.672, P(4)=0.8395, P(1)= 0.8764

Therefore, for this option we will have the Priority number equal to:

Priority Number=1+1+1+1+0.672+0.8395+0.8764= 6.3879

In change option 4, the target entities are the entities 15, 12, 19, 17, 14 and 11 .Figure 11 shows the change propagation paths of this option up to three levels.

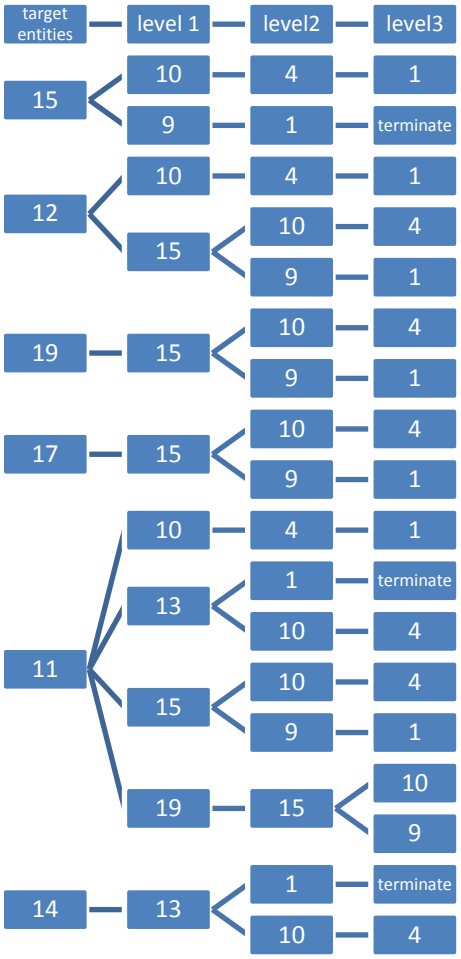

Figure 11: Change Propagation Paths for Option 4

Again with the same computations as previous options, we will have the following change probability number for each involved entity in option 4:

P(15)=P(17)=P(11)=P(14)=P(12)=P(19)=1, P(9)= 0.8625, P(4)= 0.8125, P(1)= 0.834, P(10)=0.98

Therefore, for this option we will have the Priority number equal to:

Priority Number=1+1+1+1+1+1+0.8625+0.8125+0.834+0.98= 9.489

From the obtained priority number for each option, we can conclude that option 2 has the least change propagation on the software system and option 4 has the most change propagation. Thus, option 2 has the first priority for change while option 4 has the last priority.

To validate the effectiveness of the priority numbers to estimate the scope of change propagation, the four change options are implemented. Then, we check the number of lines of codes that are actually modified (i.e., added, removed or changed). The priority numbers and the numbers of modified lines of codes for each change option are listed in Table 8 for comparison.

Table 8: Comparison of Different Change Options

Change Option	Priority Number	Number of Modified Lines of Codes
1	3.633	8
2	2.6	3
3	6.388	14
4	9.489	20

As observed, the rank of the priority numbers corresponds to the rank of the numbers of modified lines of codes. To further analyze, both the priority numbers and the numbers of modified lines of codes are normalized between 0 and 1 via the division with the largest value of their categories. Then, a plot is created in Figure 12, where the y-axis and x-axis mark the values of the normalized priority numbers and the normalized numbers of modified lines of codes, respectively. As seen in Figure 12, both sets of

values are satisfactorily correlated as the corresponding dots are roughly marked around the diagonal.

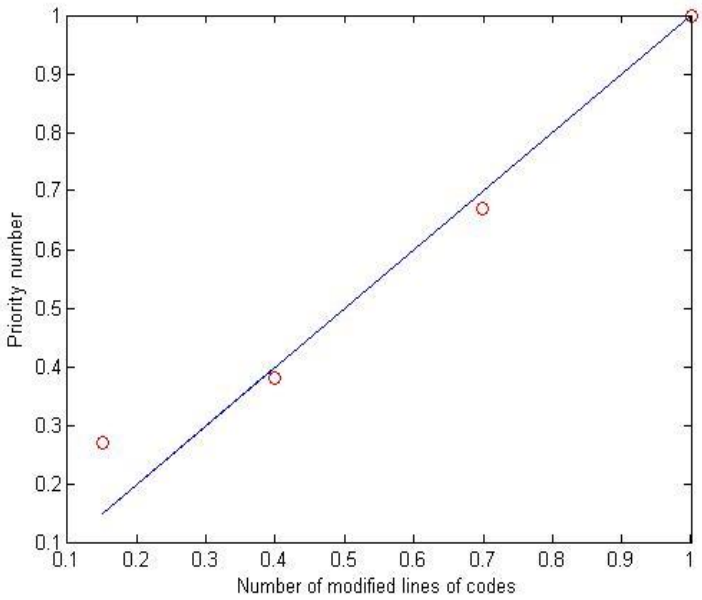

Figure 12: Comparison of Priority Number Scale vs. Modified Line of Code Scale

Chapter 4

4. Case Study

To evaluate and verify the effectiveness of using priority number as a representative scale for the scope of change propagation in a change option, we will introduce a case study in this section. Our case study is about predicting the scope of change propagation in an actuall OOP. As the case study, we have developed software to help the user to do clustering for a dependency matrix. Due to some unefficiencies that exist in our OOP, we need to have some changes in our software inorder to improve it. Our software is an OOP that receives the entries of a dependency matrix as input and set the matrix entries along the main diagonal through reordering of rows and columns. Then, it will return the manipulated matrix as the output. The output of this software can be used for clustering purposes.

Complexity is an inseparable characteristic of most of the systems. Therefore to manage these systems, managing the complexity is inevitable. Dependency matrix is one of the popular tools that have been used recently to perform both the analysis and the management of complex systems. Dependency matrix simply captures the relationships among different parts of the systems and it will let the system managers to study and analyze the dependencies among the entities of the complex systems. Dependency matrix

clustering is a method that partitions the entries of dependency matrix to the categories in a way that the entries of each category have most dependency with each other and less dependency with the entries of other categories. Dependency matrix clustering will help the user to find the subsets of dependency matrix elements that are minimally interacted. One standard method to do the dependency matrix clustering is a two-phase method that in first phase, it will set the elements of the Dependency matrix along the main diagonal of the matrix through reordering of rows and columns. Then, in the second phase, it will choose n-1 points on the diagonal to divide the matrix to n clusters (Li, 2009). Figure 13 has shown these two phases on a regular dependency matrix.

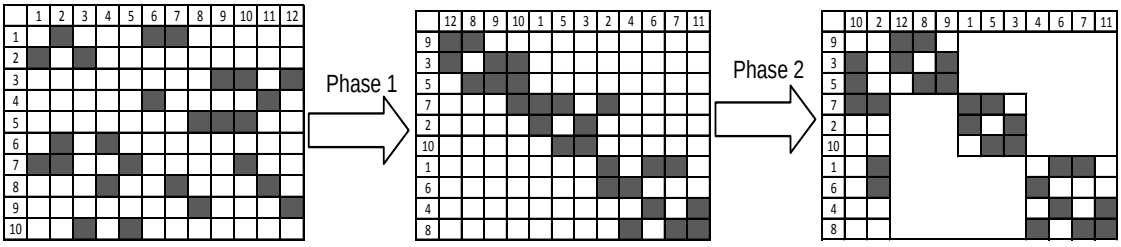

Figure 13: Workflow of the Two-Phase Decomposition Method. Adapted from (Li, 2009.)

Going through phase 1 and reordering the rows and columns in an appropriate manner is not always an easy task especially for larger matrices. As it's mentioned, the software that we have developed receives the entries of a dependency matrix as input and set the matrix entries along the main diagonal and will return the manipulated matrix as the output. The output of this program can be used as input for phase 2 of the mentioned decomposition method.

The code of this OOP is available in Appendix B. By referring to this code, we can distinguish five different classes in this program. Table 9 has listed these classes along

with a brief description for each on. More detailed description for each class will be in the following. Figure 14 has provided the direction of data flow among classes of the mentioned OOP.

Table 9: Case Study's Involved Classes

Classes	Description
Cell	Each object of this class acts as a cell of a matrix.
Node	Each object of class Node represents a branch and two sub branches of the constructed tree.
Matrix	The object of class matrix is a two dimensional array of objects of Cell class and it has a specific number of rows and columns.
Cm	This class is designed for producing the coupling matrix.
Tree	The object of this class has an array of objects of Node class (sequence), which is designed to keep the sequence of nodes in the tree.

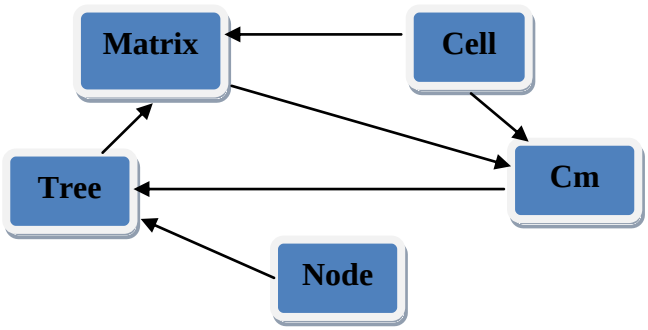

Figure 14: Direction of Data Flow among Classes

Class Cell:

Each object of this class contains the content of a specific location in the matrix and the row number and column number of that location. Also, it contains an extra row number and an extra column number that contain the changes that may eventually happen to the row number and column number or the sequence of row number and column number of that specific location.

Class Node:

Each object of class *Node* represents a branch and two sub branches of the constructed tree based on concatenated coupling matrix.

Class Matrix:

This class has a constructor that asks the user to enter the enteries of matrix, one by one and then, fill in the matrix cells. Also, this class has a method for printing the enteries of the matrix.

This class has a method (*rearrange*) that receives the sequence of rows and columns in form of a [2][n] array, that first row shows the sequence of rows and second row shows the sequence of columns. This method will rearrange the rows and columns of the original matrix based on the received array and will return the new matrix.

Class Cm:

Each *Cm* object has a two dimensional array of objects of class *Cell* for the produced coupling matrix (*cm*), a two dimensional array of objects of class *Cell* for row coupling matrix (cmr), a two dimensional array of objects of class *Cell* for column coupling matrix (cmc), a two dimensional array of objects of class *Cell* for row-column coupling matrix (cmrc), a two dimensional array of objects of class *Cell* for the transposed row-column coupling matrix (trans) and two variables that holds the number of rows and columns, which are equal to the number of rows and columns of the original matrix.

This class has a constructor that accepts the original matrix as an input and produce *cmr*, *cmc* and *cmrc* from the original matrix. Also, it has a method that produces the transposed *cmrc* from the original *cmrc* (*tcmrc*). Also, this class has a method that

produces the final coupling matrix from *cmr*, *cmc*, *cmrc* and *trans* and will return it as a two dimensional array of objects of class *Cell*.

Class Tree:

The object of this class has an array of objects of class *Node* (*sequence*), which is designed to keep the sequence of nodes in the tree and it also has two variables, which contain the original number of rows and columns in the original matrix.

This class has a function (*makeRevised*), which receives the *cm* two dimensional array as the input and produces and returns a lower triangle matrix from the original *cm*, in form of a two dimensional array of objects of class *Cell*, as the output.

This class has a function (*findMax*), which searches and finds the largest entry in the produced lower triangle *cm* matrix and returns it in form of an object of class *Cell*. This function acts as a helping function for the method (*makeTree*).

This class has a function (*makeTree*), which is designed to find linked sub branches and the related branch for those two sub branches and put them in a single object of class *Node* in the sequence array.

Methods (*findLeafs, findDependency* and *reordering*) are designed to put the branches and sub branches of the tree in a correct order in the sequence array.

This class has a method (*findSequence*) that retrieves the sequence of rows and columns from the sequence array, that has been produced in (*makeTree*) method, and reordered through (*findLeafs, findDependency* and *reordering*) methods, and puts them in a

int[2][n] array and then, returns this array. The output of this method will be used as the input for (*rearrange*) method in (*Matrix*) class, to rearrange the original matrix.

The code of this OOP contains 94 entities of type class, method, field and package. The list of these entities along with a short description for each of them is available in Appendix B. Figure 15 shows the DSM related to this code. As it is discussed before, this matrix represents dependencies that exist among the entities of the OOP program.

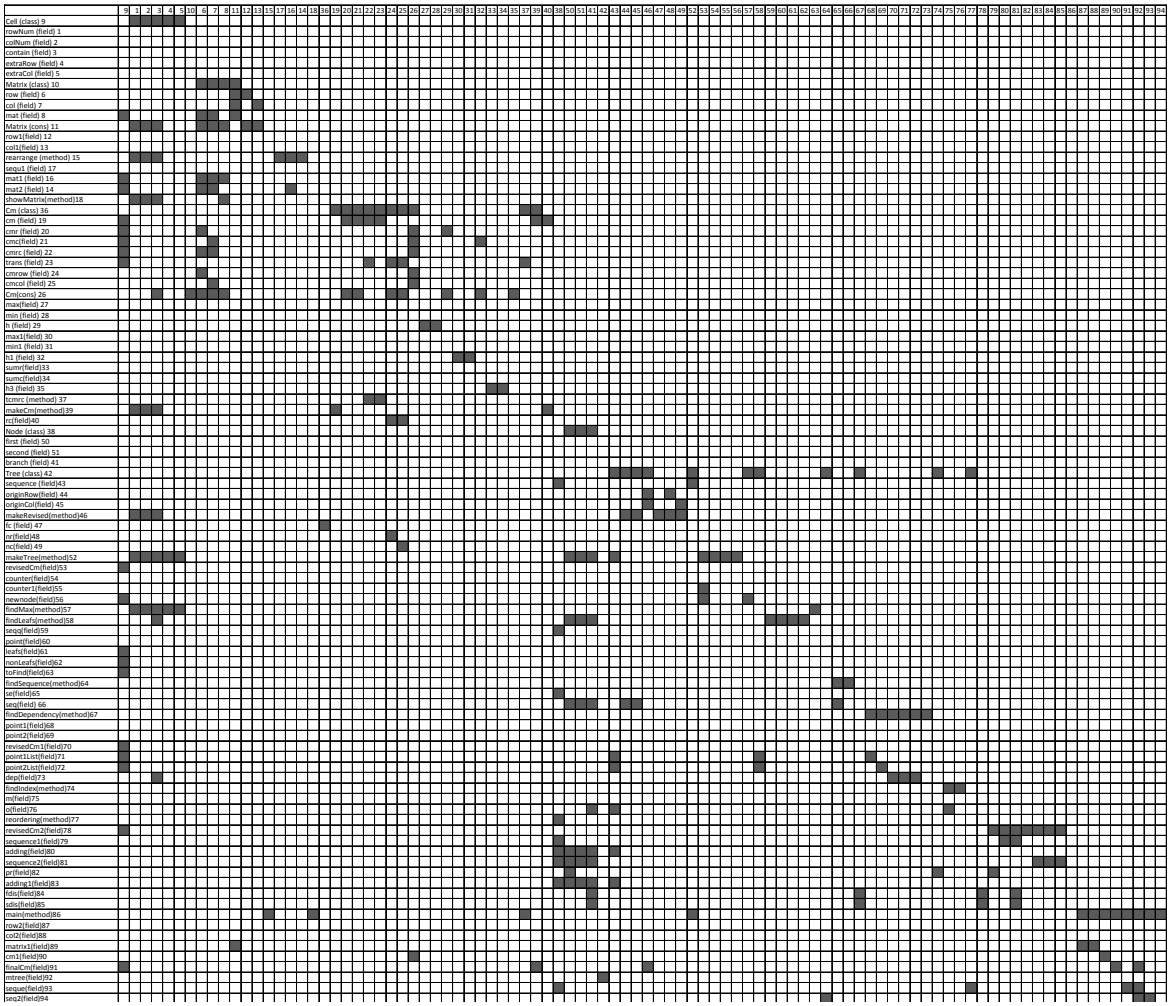

Figure 15: Dependency Matrix of the Case Study

4.1. Problem Descriptions and Change Options

Efficiency is one of the most important characteristics that every software program should have. In the mentioned OOP, there is a problem that may affect the efficiency of the program. The problem is that every single time the code tries to reach the content of each entry of *mat* matrix or *cm, cmr, cmc, cmrc* and *trans* matrices, it should go through an object of class *Cell* and this will reduce the efficiency of the program. To solve this problem, we should stop using class *Cell* in order to keep the order of entries in matrices and we should use the mathematical address of each entry of the matrix instead of that.

To apply this solution, we have suggested three different solutions:

First solution:

In this solution, we eliminate class *Cell* completely and we use two auxiliary int[][] array to keep the order of rows and columns of the *Cm* matrix in *makeTree* method. Here, we need to change *findMax* method in an appropriate way.

Second solution:

In this method, we do not eliminate *Cell* class, but we do not use it in any other classes and methodes except *makeTree* method. In *makeTree* method, we copy the matrix that is based on int[][] to a matrix based on *Cell*[][] and we let the rest of *makeTree* method continue like before. Here, we do not need to change *findMax* method at all.

Third solution:

In this method, we eliminate class *Cell* completely and we use two Arraylists of Integers to keep the branches in *makeTree* method. Here, we eliminate *findMax* method and we enter this method to *makeTree* method.

The code for all three solutions is available in Appendix B.

4.2. Estimation Case 1

In this section, we will calculate the priority number for each suggested solution in Section 4.1., based on the change propagation probability number 0.5 for all dependency relationships.

Priority number calculation for solution 1:

In solution 1, entities number 1, 2, 3, 4, 5, 52 and 15 are the target entities or the entities that change starts from those entities. Figures 16 to 22 demonstrate the trace of these entities up to three levels based on the related dependency matrix of figure 15.

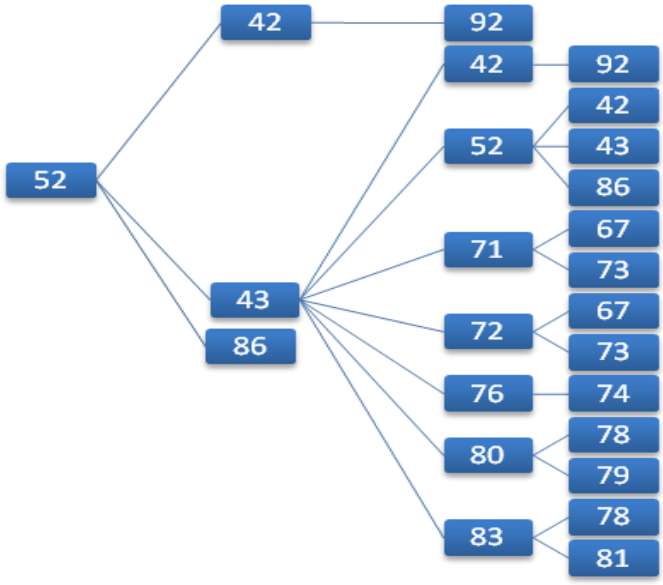

Figure 16: Change Propagation from Target Entity 52

Figure 17: Change Propagation from Target Entity 15

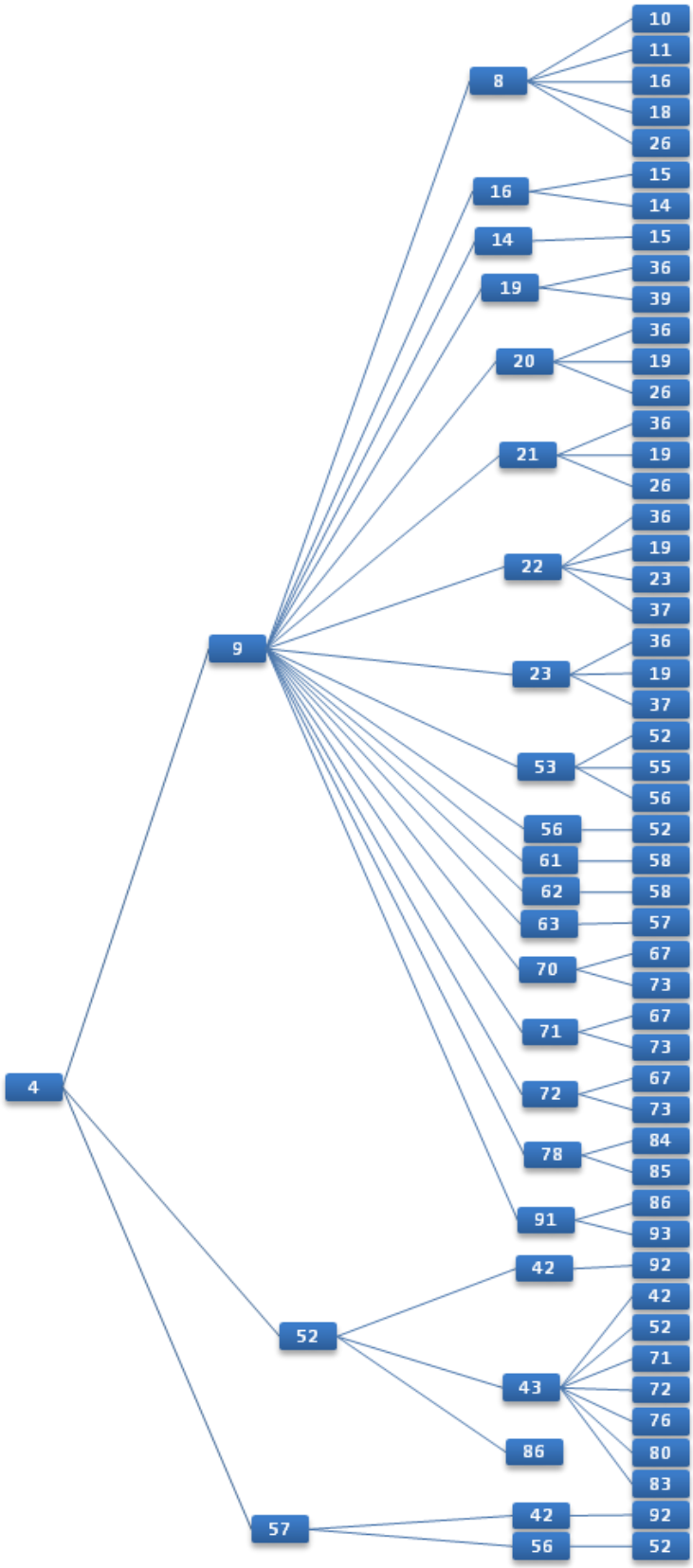

Figure 18: Change Propagation from Target Entity 4

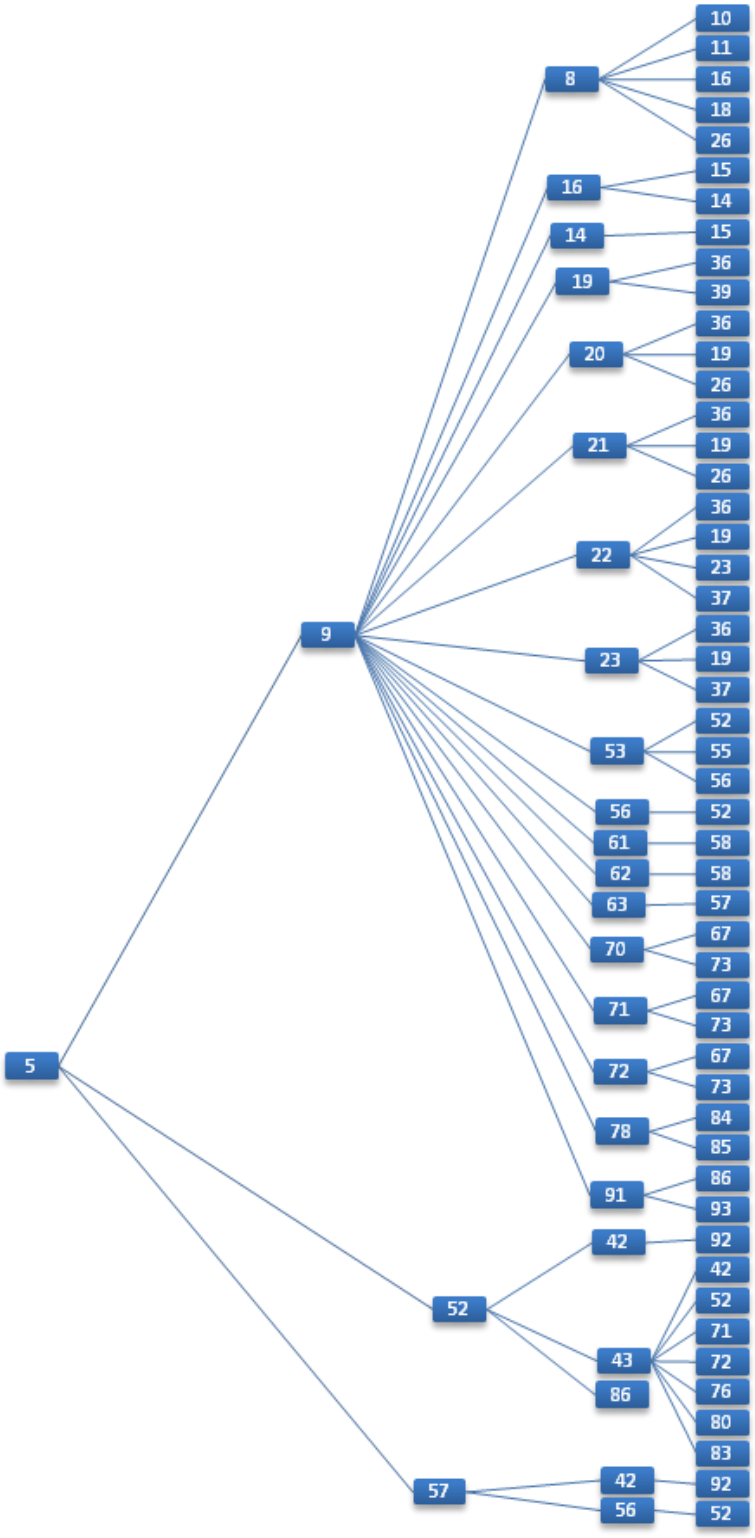

Figure 19: Change Propagation from Target Entity 5

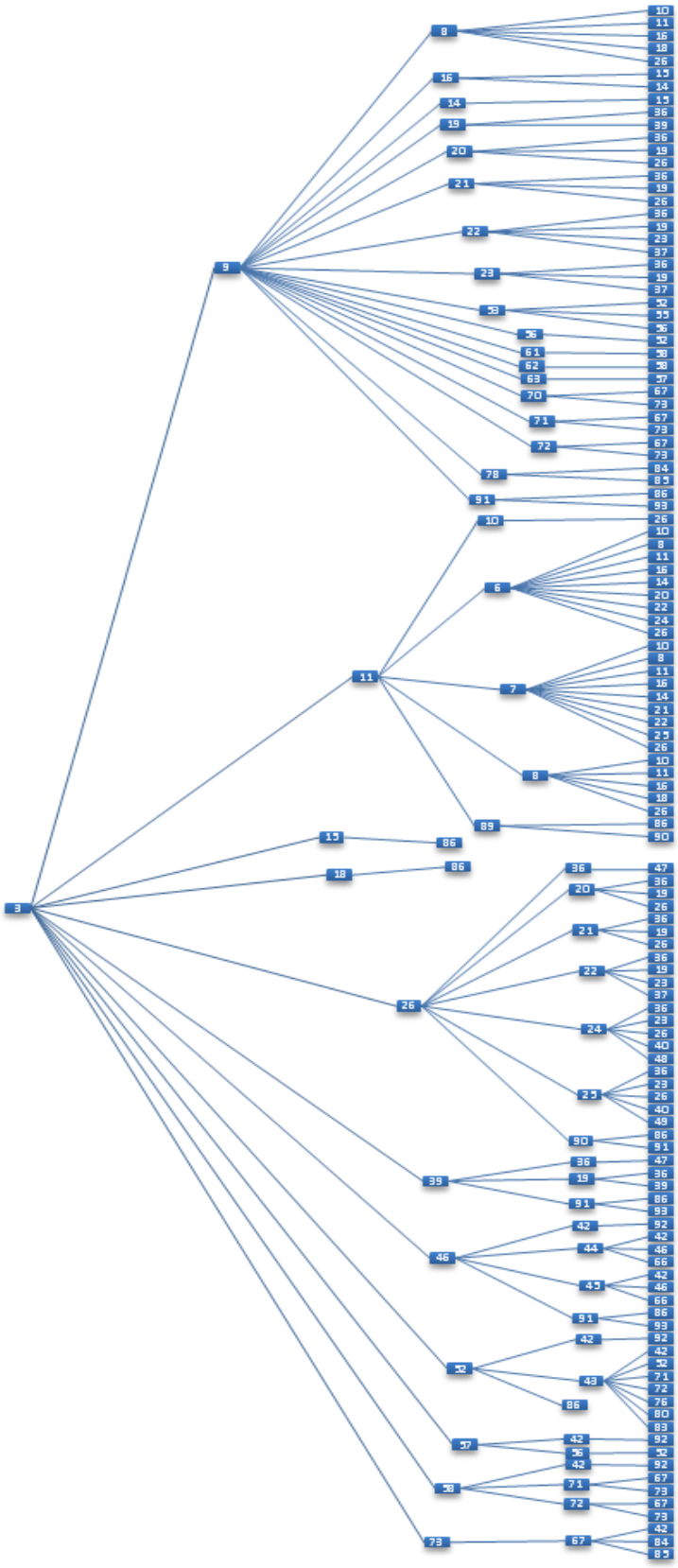

Figure 20: Change Propagation from Target Entity 3

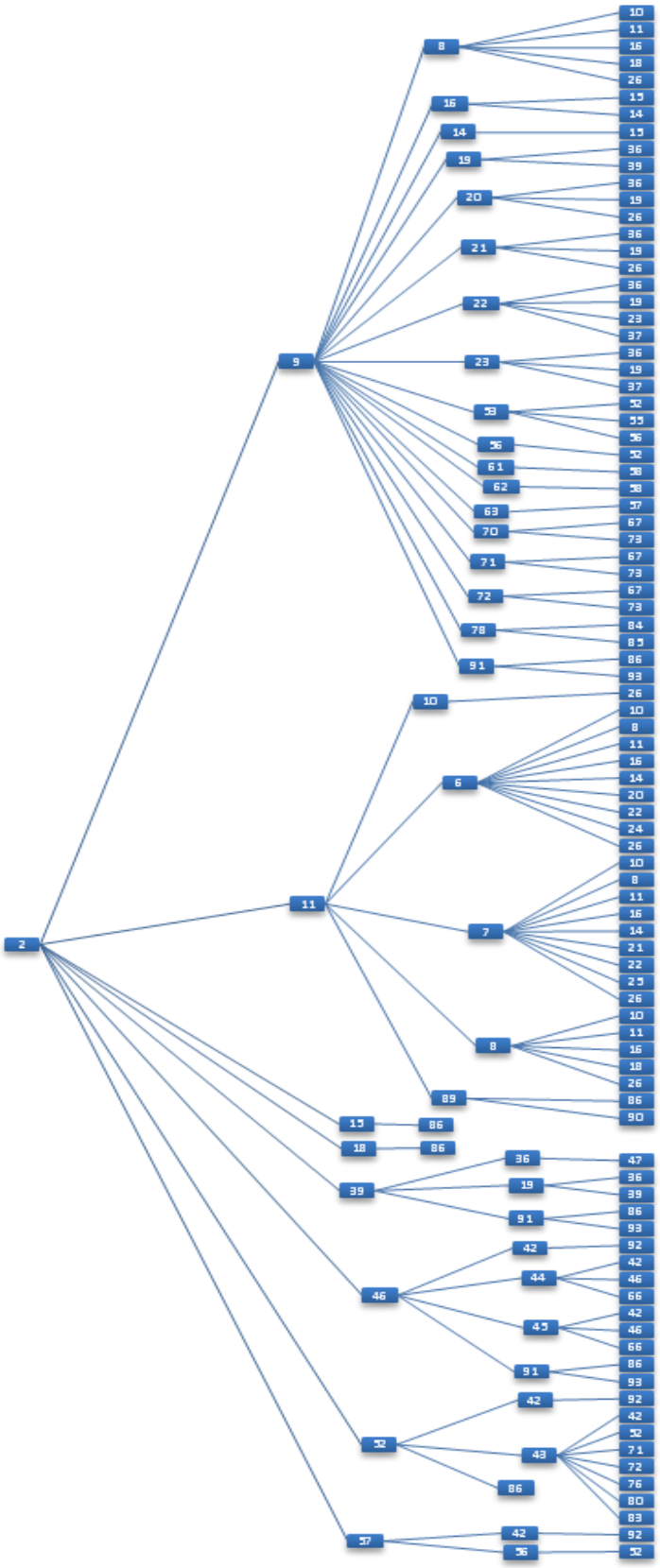

Figure 21: Change Propagation from Target Entity 2

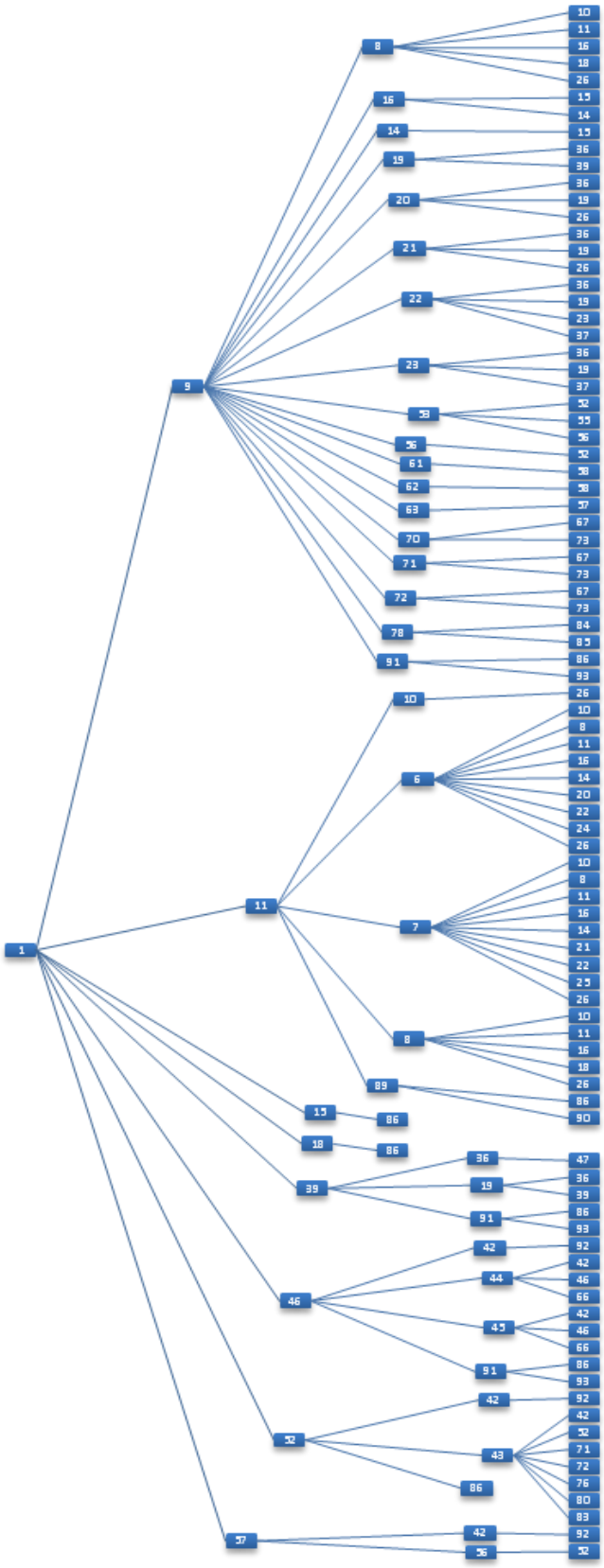

Figure 22: Change Propagation from Target Entity 1

Now, we have other entities that are predicted to be involved in change for solution 1. If we use the change propagation probability number (0.5) to predict the probability of change for each of these entities based on formula 1 and 2 of Section 3.4, we will have the change probability numbers that are listed in Table 10 for each involved entity (note that the probability of change for the target entities is equal to 1).

Table 10: Change Probability Numbers for the Involved Entities in Solution 1 Based on Change Propagation Probability Number 0.5

P(8)= 0.955	P(10)= 0.935	P(11)= 0.981	P(16)= 0.963	P(18)= 0.957
P(26)= 0.97	P(6)= 0.578	P(7)= 0.578	P(89) =0.578	P(14)= 0.945
P(86)= 0.98	P(25)= 0.498	P(90)= 0.498	P(39)= 0.957	P(36)= 0.98
P(20)= 0.881	P(21)= 0.881	P(22)= 0.92	P(24)= 0.498	P(19)= 0.975
P(47)= 0.414	P(91)= 0.968	P(23)= 0.918	P(53)= 0.763	P(56)= 0.971
P(61)= 0.763	P(62)= 0.763	P(63)= 0.763	P(70)= 0.763	P(71)= 0.932
P(72)= 0.931	P(78)= 0.818	P(46)= 0.944	P(57)= 0.984	P(58)= 0.868
P(73)= 0.958	P(42)= 0.98	P(43)= 0.896	P(9)= 0.969	P(37)= 0.77
P(55)= 0.487	P(67)= 0.94	P(84)= 0.551	P(85)= 0.551	P(93)= 0.799
P(44)= 0.578	P(45)= 0.578	P(92)= 0.899	P(76)= 0.615	P(80)= 0.615
P(83)= 0.615	P(81)= 0.125	P(94)= 0.125	P(66)= 0.551	P(40)= 0.234
P(48)= 0.125	P(49)= 0.125	P(74)= 0.125	P(79)= 0.125	

Refer to formula 3 of Section 3.4., if we add the change probability numbers of target entities and change probability numbers of predicted involved entities, we will have the priority number for solution 1, (with change propagation probability number 0.5), which is equal to (49.407).

Priority number calculation for solution 2:

In solution 2, entities number 8, 16, 14, 18, 20, 21, 22, 23, 19, 46, 53, 61, 62, 70, 71, 72 and 78 are the target entities or the entities that change starts from those entities. Appendix C demonstrates the trace of these entities up to three levels based on the related dependency matrix of Figure 15.

Now, we have other entities that are predicted to be involved in change for solution 2. If we use the change propagation probability number (0.5) to predict the probability of change for each of these entities based on formula 1 and 2 of Section 3.4., we will have the change probability numbers that are listed in Table 11 for each involved entity (note that the probability of change for the target entities is equal to 1).

Table 11: Change Probability Numbers for the Involved Entities in Solution 2 Based on Change Propagation Probability Number 0.5

P(10) = 0.749	P(11)= 0.665	P(26) = 0.99	P(15) =0.877	P(86)= 0.99
P(36) = 0.99	P(37) = 0.904	P(39) = 0.879	P(42) = 0.989	P(44) =0.617
P(45) = 0.617	P(91) = 0.914	P(43) = 0.562	P(58) = 0.75	P(67) = 0.976
P(73)= 0.944	P(84) = 0.917	P(85) = 0.917	P(6) = 0.25	P(7) = 0.25
P(89) = 0.25	P(24) = 0.677	P(25) = 0.677	P(90) = 0.677	P(47) = 0.945
P(92) = 0.808	P(66) =0.437	P(93) = 0.498	P(76) = 0.25	P(80) =0.25
P(83) =0.25	P(81) = 0.779	P(40) = 0.551	P(48) = 0.330	P(49) = 0.330
P(94) = 0.330	P(64) = 0.234	P(74) = 0.125	P(79) = 0.330	

Refer to formula 3 of Section 3.4., if we add the change probability numbers of target entities and change probability numbers of predicted involved entities, we will have the priority number for solution 2, (with change propagation probability number 0.5), which is equal to (41.475).

For the third solution, although it is different from the first solution, but the target entities are exactly the same as first solution. Therefore, the predicted entities and the priority number for the third solution will be the same as first solution too.

4.3. Estimation Case 2

In previous section, we considered change propagation probability number equal to (0.5). Therefore, we have the probability of (0.5) for the first level of change trace, (0.25) for the second level of change trace and (0.125) for the third level of change trace. If we need

to have a more precise prediction about the probability of change for each predicted entities, we need to have a more precise change propagation probability number than (0.5). In this regard, by refer to Table 7 of Section 3.2., we will consider the change propagation probability numbers that are obtained for each two type of entities in Section 3.2., as the reference change propagation probability numbers in this section. For simplicity, we have categorized these numbers in three ranges (0% to 62%), (63% to 88%) and (89% to 100%) and we will consider all the numbers in the first range as 50%, the numbers in second range as 75%, and the numbers in third range as 100%. With this consideration, we will have the change propagation probability numbers that are listed in Table 12 for each two entities.

Table 12: Change Propagation Probability among Entities

Class - Method	50%	Field-Field	75%
Class-Field	75%	Class-Interface	75%
Field-Method	75%	Package-Class	100%
Method-Method	75%	Package-Interface	100%
Class-Class	75%	Interface-Interface	50%

Based on the change propagation probability numbers that are listed in Table 12, if we calculate the change probability numbers for predicted involved entities in solution 1, we will have the results that are listed in Table 13. Also, the change probability numbers for predicted involved entities in solution 2 are listed in Table 14. Again for the third solution, since the target entities are exactly the same as first solution, the predicted entities and the priority number for the third solution will be the same as first solution too.

Table 13: Change Probability Numbers for the Involved Entities in Solution 1 Based on Change Propagation Probability Numbers 0.5, 0.75 and 1

P(8)= 0.999	P(10)= 0.999	P(11)= 0.999	P(16)= 0.999	P(18)= 0.999
P(26)= 0.999	P(6)= 0.915	P(7)= 0.915	P(89) =0.915	P(14)= 0.999
P(86)= 0.999	P(25)= 0.914	P(90)= 0.914	P(39)= 0.999	P(36)= 0.999
P(20)= 0.998	P(21)= 0.998	P(22)= 0.999	P(24)= 0.914	P(19)= 0.999
P(47)= 0.731	P(91)= 0.999	P(23)= 0.999	P(53)= 0.983	P(56)= 0.999
P(61)= 0.983	P(62)= 0.983	P(63)= 0.983	P(70)= 0.983	P(71)= 0.999
P(72)= 0.999	P(78)= 0.994	P(46)= 0.999	P(57)= 0.999	P(58)= 0.998
P(73)= 0.999	P(42)= 0.999	P(43)= 0.997	P(9)= 0.999	P(37)= 0.997
P(55)= 0.934	P(67)= 0.999	P(84)= 0.962	P(85)= 0.962	P(93)= 0.998
P(44)= 0.915	P(45)= 0.915	P(92)= 0.999	P(76)= 0.971	P(80)= 0.971
P(83)= 0.971	P(81)= 0.42	P(94)= 0.42	P(66)= 0.962	P(40)= 0.663
P(48)= 0.42	P(49)= 0.42	P(74)= 0.42	P(79)= 0.42	

If we add the change probability numbers of target entities and change probability numbers of predicted involved entities, we will have the priority number for solution 1, (with change propagation probability number 0.5, 0.75 and 1), which is equal to 60.835.

Table 14: Change Probability Numbers for the Involved Entities in Solution 2 Based on Change Propagation Probability Number 0.5, 0.75 and 1

P(10) = 0.970	P(11)= 0.951	P(26) = 0.999	P(15) =0.993	P(86)= 0.999
P(36) = 0.999	P(37) = 0.998	P(39) = 0.997	P(42) = 0.999	P(44) =0.916
P(45) = 0.916	P(91) = 0.999	P(43) = 0.855	P(58) = 0.937	P(67) = 0.999
P(73)= 0.999	P(84) = 0.999	P(85) = 0.999	P(6) = 0.56	P(7) = 0.56
P(89) = 0.56	P(24) = 0.971	P(25) = 0.971	P(90) = 0.971	P(47) = 0.999
P(92) = 0.990	P(66) =0.806	P(93) = 0.914	P(76) = 0.56	P(80) =0.56
P(83) =0.56	P(81) = 0.996	P(40) = 0.962	P(48) = 0.805	P(49) = 0.805
P(94) = 0.805	P(64) = 0.664	P(74) = 0.42	P(79) = 0.805	

If we add the change probability numbers of target entities and change probability numbers of predicted involved entities, we will have the priority number for solution 2, (with change propagation probability number 0.5, 0.75 and 1), which is equal to 50.768.

4.4. Discussion and Verification

To validate the obtained priority numbers for solution 1 and solution 2, from case 1 and case 2, we should have a real change parameter from the actual code. In this work, we have chosen the number of changed lines of code for each solution. The number of changed lines of code means the sum of the number of lines of code which are added, deleted and changed. Therefore, we have 178 changed LOC (Lines Of Code) for the first solution and 140 changed LOC for the second solution. The priority numbers and the numbers of modified lines of code for each change option are listed in Table 15 for comparison.

Table 15: Comparison of Different Change Options

	Number of Modified Line of Code	Priority Number (0.5)	Priority Number (0.5, 0.75, 1)
Solution 1	178	49.407	60.835
Solution 2	140	41.475	50.768

To be able to have a comparison between the obtained priority numbers and LOC which are actually changed, we need to normalize all the priority numbers and changed LOC between 0 and 1. To do this, we have divided each number by the sum of both numbers. For example, for changed LOC, we should divide 178 by (178+140) and also, we should divide 140 by (178+140). Therefore, we have the normalized numbers 0.5597 for the actual changed LOC for the first solution and 0.4402 for the actual changed LOC for the second solution. In a same way, if we normalize the obtained priority numbers with change propagation probability number 0.5, we will have 0.5436 for the first solution and 0.4564 for the second solution. Also, if we normalize the obtained priority numbers with change propagation probability number (0.5, 0.75 and 1), we will have 0.5451 for the

first solution and 0.4549 for the second solution. The difference between the normalized changed LOC is (0.1195). The difference between the normalized priority numbers, with change propagation probability number 0.5, is (0.0872) and the difference between the normalized priority numbers for the improved evaluation is (0.0902). Since the difference between 0.0902 and 0.1195 is less than the difference between 0.0872 and 0.1195, we can say that the priority numbers, which are obtained with change propagation probability number (0.5, 0.75 and 1), are more precise. To make it clearer, we have plotted both sets of priority numbers against the actual changed LOC. We can see that in figure 24, that demonstrates the priority numbers obtained from change propagation probability numbers (0.5, 0.75 and 1), the corresponding dots are closer to the diagonal compare to figure 23, which shows the priority numbers obtained from change propagation probability number 0.5.

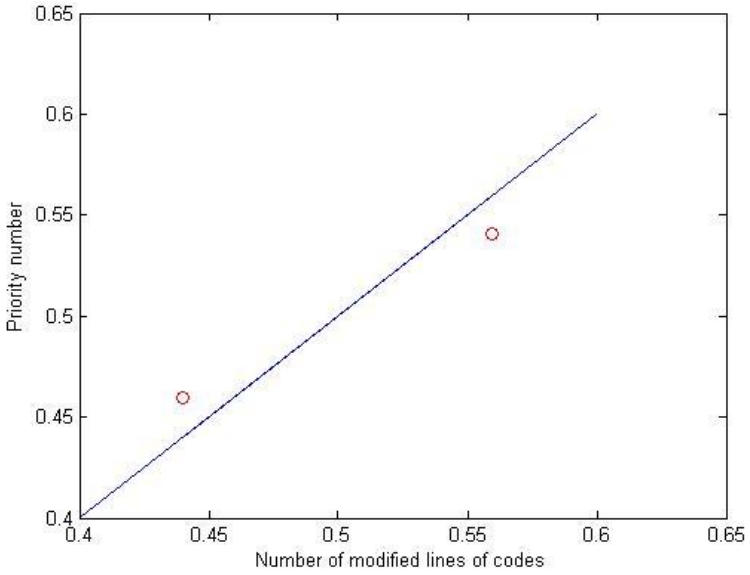

Figure 23: Evaluation of Priority Numbers that are Based on Change Propagation Probability Number 0.5

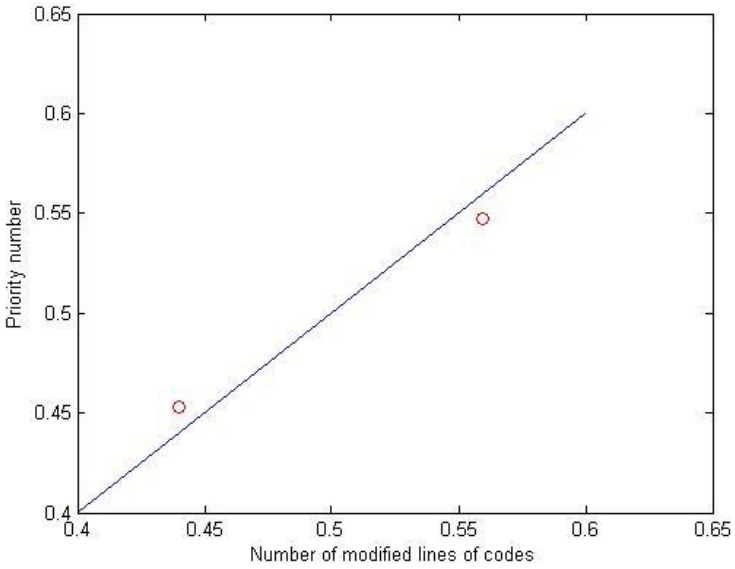

Figure 24: Evaluation of Priority Numbers that are Based on Change Propagation Probability Numbers (0.5, 0.75 and 1)

Chapter 5

5. Conclusion and Future Work

This work reports our effort on the application of matrix-based modeling to manage change propagation in an object-oriented system. The contribution can be viewed in three aspects.

- Firstly, a dependency model is derived for object-oriented programs and the dependency information can be compactly captured in a matrix format. Through the use of matrix, we can trace the propagation paths from the initial changes on a program. The propagation paths can be utilized to estimate of the scope of change propagation.
- Secondly, through the analysis and classification of dependency types among the fundamental entities of OOP and categorization of existing dependencies between each couple of entities, we can calculate the values of direct propagation probabilities for different dependency relationships.
- In third step, the notion of priority number is proposed to estimate the expected number of changed entities based on a change proposal. A case study program has been used to demonstrate and validate the use of matrix-based modeling, the obtained results for direct propagation probability and the priority number.

The future work of this research can include two directions.

- Firstly, as it is mentioned, the primary step in calculation of priority number is detection of target entities. If the detected target entities for two different change options be the same, although the change options may be completely different, the calculated priority numbers will be the same for both of them. In our case study, first suggested solution and third suggested solution can be an example for this situation. As it's seen, although the application of change is completely different for each solution, the target entities are equal for both solutions and we have the same priority numbers for both of them too. However, we need to have a guideline to distinguish the difference among the scope of propagated change in the software for each suggested solution in these specific situations.
- Secondly, in the context of matrix-based modeling, matrix patterns and structuring have been utilized to control the propagation of changes (Li and Chen 2009), and this method of approach is rarely found in the context of OOP. Thus, we need to incorporate the matrix-based structural characteristics to effectively address the issue of change management in OOP.

References

Browning, T.R., 2001, "*Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions,*" IEEE Transaction on Engineering Management, Vol. 48, No. 3, pp. 292-306.

Chen, L., Macwan, A., and Li, S., 2007, "*Model-based Rapid Redesign using Decomposition Patterns,*" ASME Journal of Mechanical Design, (129), pp. 283-294.

Carl A, G., 2000, "*Abstracting dependencies between software configuration items*" ACM Transactions on Software Engineering and Methodology (TOSEM) Journal, Vol.9, Issue 1, pp.94-131.

Clarkson, J., and Simons, C., and Eckert, C., 2001, "*Predicting Change Propagation In Complex Design*", Journal of Mechanical Design, Vol.126, Issue 5, pp. 788-797

Danilovic, M. and Browning, T.R., 2007, "*Managing Complex Product Development Projects with Design Structure Matrices and Domain Mapping Matrices*," International Journal of Project Management, Vol. 25, pp. 300-314.

Dam, K.H., and Winikoff, M., and Padgham, L., 2006, "*An agent-oriented approach to change propagation in software evolution*" Australian Software Engineering Conference, Melbourne, Vic., Australia, pp. 309-318

Fowler, M., 1999, "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA

Gallagher, K.B., and Lyle, J.R., 2002, "*Using program slicing in software maintenance*" IEEE Transactions on Software Engineering, Vol. 17, Issues 8, pp. 751-761

Giffin, M., and Weck, O., and Bounova, G., 2009, "*Change Propagation Analysis in Complex Technical Systems*" Journal of Mechanical Design, Vol.131, Issue 8, 081001.

Greg J, B., 2000, "*JavaML: A Markup Language for Java Source Code*" Computer Networks, Vol. 33, Issues 1-6, pp. 159-177

Han, A.H., and Jeon, S.U., and Bae, D.H., and Hong, J.E., 2008, "*Behavioral Dependency Measurement for Change-proneness Prediction in UML 2.0 Design Models*"

Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, IEEE Computer Society, Washington, DC, USA, pp.76 - 83.

Han, J., 1997, "*Supporting Impact Analysis and Change Propagation in Software Engineering Environments*" Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering, London, UK, pp.172-182.

Hassan, A.E., and Holt, R.C., 2004, "*Predicting Change Propagation in Software Systems*" Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp. 284 – 293.

Kagdi, H., and Maletic, J.I., 2006, "*Software-Change Prediction: Estimated+Actual*" Proceedings of the Second International IEEE Workshop on Software Evolvability, IEEE Computer Society, Washington, DC, USA, pp.38 – 43.

Korn, J., and Yih-Farn, C., and Koutsofios, E., 1999, "*Chava: Reverse Engineering and Tracking of Java Applets*" Sixth Working Conference on Reverse Engineering, Atlanta, GA, USA, pp. 314-325.

Li, S., and Chen, L., 2009, "*Pattern-based Reasoning for Rapid Redesign: A Proactive Approach,*" Research in Engineering Design, 21(1), pp. 25-42.

Li, S., 2010, "*Extensions of the Two-Phase Method for Decomposition of Matrix-based Design Systems,*" ASME Journal of Mechanical Design, Vol. 132, 061003.

Mens, T., and Tourwe, T., 2004, "*A survey of software refactoring*", IEEE Transactions on Software Engineering, Vol. 30, Issues 2, pp. 126 – 139

Mirarab, S., and Hassouna, A., and Tahvildari, L., 2007, "*Using Bayesian Belief Networks to Predict Change Propagation in Software Systems*", Proceedings of the 15th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, pp.177-188.

Rajlich, V., 1997, "*A Model for Change Propagation Based on Graph Rewriting*", Proceedings of the international Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp. 84 – 91.

Rayside, D., and Kontogiannis, K., 1999, "*Extracting Java Library Subsets for Deployment on Embedded Systems*" Proceedings of the Third European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, pp. 102-110

Ren, X., and Shah, F., and Tip, F., and Ryder, B., and Chesley, O., 2004, "*Chianti: a tool for change impact analysis of java programs*" Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vol.39, Issue.10.

Sharafat, A.R., and Tahvildari, L., 2008 "*Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach*" journal of software, 3(5), pp. 26-39.

Sharafat, A.R., and Tahvildari, L., 2007, "*A Probabilistic Approach to Predict Changes in Object-Oriented Software Systems*", Proceedings of the 11th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA,pp. 27-38

Sangal, N., and Jordan, E., and Sinha, V., and Daniel, D., 2005, "*Using Dependency Models to Manage Complex Software Architecture*" Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vol.40, Issue 10, pp. 167 - 176

Sierra, K., and Bates, B., 2005, "*Head First Java*", O'Reilly Media, Inc, Sebastopol, CA, USA, pp.103-112 Chap.5 and pp.127-130 Chap.6.

Sosa, M., 2008, "*A structured approach to predicting and managing technical interactions in software development*" Research in Engineering Design Journal, Vol.19, Number 1, pp.47-70.

Tsantalis, N., and Chatzigeorgiou, A., and Stephanides, G., 2005, "*Predicting the Probability of Changein Object-Oriented Systems*" IEEE Transactions on Software Engineering, 31(7), pp. 601-614.

Tan, X., and Feng, T., and Zhang, J., 2007, "*Mapping Software Design Changes to Source Code Changes*" Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing-Cover, Qingdao, pp. 650-655

Xia, F., and Srikanth, P., 2005, "*A Change Impact Dependency Measure for Predicting the Maintainability of Source Code*" Proceedings of the 28th Annual International Computer Software and Applications Conference, Rolla, MO, USA, Vol.2, pp.22-23.

Zhifeng, Yu., and Rajlich, V., 2001, "*Hidden Dependencies in Program Comprehension and Change Propagation*", Proceedings of IWPC2001, IEEE Computer Society Press, Los Alamitos, CA, pp. 293 - 299.

Appendix A: Details of the Example Program

This appendix includes the original code of the example program, which is shown in Figure A1. In addition, this appendix includes the list of OOP entities of this program, which is shown in Table A1.

```
public static void main(String[] args){

 int numOfGuesses=0;

GameHelper helper=new GameHelper();

   SimpleDotCom theDotCom=new SimpleDotCom();

int  randomNum=(int)(Math.random()*5);  //  generate a random number

int[] locations={ randomNum, RandomNum+1, RandomNum+2};  // put the generated number
and two following number in an int array

theDotCome.setLocationCells(locations);

boolean isAlive=true;

while ( isAlive==true){

    string guess=helper.getUserInput("enter a number"); // ask user for a guess

    string result=theDotCom.checkYourself(guess); // compare the user guess with the
numbers in the int array

    numofGuesses++;  // increase the number of guesses that user made

    if (result.equals("kill")){  // check if all the numbers in the int array are
guessed by the user or not and if yes then

     isAlive=false;

     system.out.println ("you took"+numofGuesses+"guesses"); // print the number of
guesses that the user made

                              }

                        }

}
```

```
public class SimpleDotCom{

   int[] locationCells;

   int numOfHits=0;

   public void setLocationCells (int[] locs){  // this function will receive the
random numbers that are generated by the system and will put them in the int array
related to this class

      locationCells=locs;

      }

  public string checkYourself(string stringGuess){ // receive the guess that is made
by the user

     int guess= Integer.parseInt(stringGuess);

     string result="miss";

     for (int cell: locationCells){   // compare the guess that is made by the user
with all the random numbers that are generated by system

          if (guess==cell){  //  if the guess was equal to each of the generated
numbers then

               result="hit";

               numOfHits++;  //  increase the number of hits

               break;

          }

    }

    if (numOfHits==locationCells.length){  // check if all the numbers that are
generated by the system are guessed by the user

         result="kill";

        }

    system.out.println(result);

      return result;  //  print and return the result as the output of the function

       }

 }
```

```
import java.io.*

public class GameHelper{  // this class will help the system to receive the input
from user

   public String getUserInput(String prompt){

        string inputLine=null;

        system.out.print(prompt+"  ");
```

```
        try{
           BufferedReader is=new BufferedReader(
           new InputStreamReader(System.in));
           inputLine=is.readLine();
           if( inputLine.length()==0) return null;
          }catch(IOException e){
           system.out.println("IOException:"+e);
                       }  return inputLine;
        }  }
```

Figure A1. A Sample of an Object-Oriented Program

Table A1. List of 25 OOP Entities of the Sample Program

	Label	Name	Description
1	M_1	`main`	this method is the main execution part of the program
2	F_1	`numOfGuesses`	a variable (field) of type `int` that is defined in `main` method
3	F_2	`helper`	an object (field) of class `gamehelper` that is defined in `main` method
4	F_3	`theDotCom`	an object (field) of class `simpledotcom` that is defined in `main` method
5	F_4	`randomNum`	a variable (field) of type `int` that is defined in `main` method
6	F_5	`locations`	a variable (field) of type `int` array that is defined in `main` method
7	F_6	`isAlive`	a variable (field) of type `boolean` that is defined in `main` method
8	F_7	`guess`	a variable (field) of type `string` that is defined in `main` method
9	F_8	`result`	a variable (field) of type `string` that is defined in `main` method

10	C_1	SimpleDotCom	a class
11	F_9	locationCells	a variable (field) of type int array that is defined in simpledotcom class
12	F_{10}	numOfHits	a variable (field) of type int that is defined in simpledotcom class
13	M_2	setlocationCells	a method that is defined in simpledotcom class
14	F_{11}	locs	a variable (field) of type int array that is defined in setlocationcells method in *simpledotcom* class
15	M_3	checkYourself	a method that is defined in simpledotcom class
16	F_{12}	stringGuess	a variable (field) of type string that is defined in checkyourself method in simpledotcom class
17	F_{13}	guess	a variable (field) of type int that is defined in checkyourself method in simpledotcom class
18	F_{14}	result	a variable (field) of type string that is defined in checkyourself method in simpledotcom class
19	F_{15}	cell	a variable (field) of type int that is defined in checkyourself method in simpledotcom class
20	P_1	java.io	a package
21	C_2	GameHelper	a class
22	M_4	getUserInput	a method that is defined in gamehelper class
23	F_{16}	prompt	a variable (field) of type string that is defined in getuserinput method in gamehelper class
24	F_{17}	inputLine	a variable (field) of type string that is defined in getuserinput method in gamehelper class
25	F_{18}	is	an object (field) of class BufferedReader* that is defined in getuserinput method in gamehelper class * BufferReader is one of the classes of java.io package

Appendix B: Details of the Case Study

This appendix includes the original code of the case study OOP, which is shown in Figure B1, along with the modified code of the mentioned software based on solution1, solution2 and solution3, which are shown in Figures B2, B3 and B4 respectively. In addition, this appendix includes the list of OOP entities of the mentioned OOP, which is shown in Table B1.

```
package clustering;
import java.text.DecimalFormat;
import java.util.*;

class Cell
        {
    int rowNum;
    int colNum;
    float contain;
    int extraRow;
    int extraCol;
}
```

```
class Matrix
{   int row;
    int col;
    Cell[][] mat;

         public Matrix(int row1, int col1){
         row=row1;
         col=col1;
         mat= new Cell[row][col];
                  for (int i=0; i<row; i++){
                     int p=i+1;
                     System.out.println("enter the entities of row number"+ p );
                  for (int j=0; j<col; j++){
                   Scanner getEntity= new Scanner(System.in);
                   int next;
                   next= getEntity.nextInt();
```

```
                    mat[i][j]=new Cell();
                        mat[i][j].contain=next;
                        mat[i][j].rowNum=i+1;
                        mat[i][j].colNum=j+1;

                   }
                                                  }
                       }

public Cell[][] rearrange(int[][] sequ1)
        {
    Cell[][] mat1=new Cell[row][col];
    for(int d=0; d<row; d++){
                       for (int f=0; f<col; f++){
                           int h=(sequ1[0][d])-1;
                            mat1[d][f]=new Cell();
            mat1[d][f].contain=mat[h][f].contain;
            mat1[d][f].colNum=mat[h][f].colNum;
            mat1[d][f].rowNum=mat[h][f].rowNum;
                   }
                             }

    Cell[][] mat2=new Cell[row][col];
    for(int d1=0; d1<col; d1++){
                       for (int f1=0; f1<row; f1++){
                           int h8=(sequ1[1][d1])-1;
                            mat2[f1][d1]=new Cell();
            mat2[f1][d1].contain=mat1[f1][h8].contain;
            mat2[f1][d1].rowNum=mat1[f1][h8].rowNum;
            mat2[f1][d1].colNum=mat1[f1][h8].colNum;
                   }
                                  }

    for(int d5=0; d5<row; d5++){
                       for (int f5=0; f5<col; f5++){
                System.out.print(mat2[d5][f5].contain+" ");
                System.out.print(mat2[d5][f5].rowNum+" ");
                System.out.print(mat2[d5][f5].colNum+"      ");

        }
                       System.out.println();
                       }

    return mat2;
}

public void showMatrix()
    {    System.out.println("this is the original matrix");
       for (int i=0; i<row; i++){
             for (int j=0; j<col; j++){
                 System.out.print(mat[i][j].rowNum);
                 System.out.print(mat[i][j].colNum);
                 System.out.print((int)mat[i][j].contain+"  ");
             }
             System.out.println();
         }
}
}
```

```
class Cm
{
    Cell[][] cm;
    Cell[][] cmr;
    Cell[][] cmc;
    Cell[][] cmrc;
    Cell[][] trans;
    int cmrow;
    int cmcol;

    public Cm(Matrix matrixIn)
    {
```

```
        cmrow= matrixIn.row;
        cmcol= matrixIn.col;
        cmr=new Cell[matrixIn.row][matrixIn.row];
        for (int m=0; m<matrixIn.row; m++)
            for (int n=0; n<matrixIn.row; n++)
            {
                if (m==n)
                {
                    cmr[m][n]=new Cell();
                    cmr[m][n].contain=0;
                }
                else
                {
                    int max=0;
                    int min=0;
                    for (int k=0; k<matrixIn.col; k++)
                    { if ((matrixIn.mat[m][k].contain==1) ||
(matrixIn.mat[n][k].contain==1))
                            {
                                max++;}
                      if ((matrixIn.mat[m][k].contain==1) &&
(matrixIn.mat[n][k].contain==1))
                      {
                          min++;}
                    }
                    float h=(float)min/max;
                    cmr[m][n]=new Cell();
                    cmr[m][n].contain=h;
                }
                }
        //show the matrix CMr
         System.out.println();
         System.out.println("this is the CMr Matrix");
        for (int q=0; q<matrixIn.row; q++){
                  for (int h=0; h<matrixIn.row; h++){
                   DecimalFormat df1 = new DecimalFormat("#.##");
                   System.out.print(df1.format(cmr[q][h].contain)+"      ");
             }
              System.out.println();
         }

        cmc=new Cell[matrixIn.col][matrixIn.col];
        for (int m1=0; m1<matrixIn.col; m1++)
            for (int n1=0; n1<matrixIn.col; n1++)
            {
                if (m1==n1)
                {
                    cmc[m1][n1]=new Cell();
                    cmc[m1][n1].contain=0;
                }
                else
                {
                    int max1=0;
                    int min1=0;
                    for (int k1=0; k1<matrixIn.row; k1++)
                    { if ((matrixIn.mat[k1][m1].contain==1) ||
(matrixIn.mat[k1][n1].contain==1))
                            {
                                max1++;}
                      if ((matrixIn.mat[k1][m1].contain==1) &&
(matrixIn.mat[k1][n1].contain==1))
                      {
                          min1++;}
                    }
                    float h1=(float)min1/max1;
                    cmc[m1][n1]=new Cell();
                    cmc[m1][n1].contain=h1;
                }
                }
        // show the matrix CMc
```

```
         System.out.println();
         System.out.println("this is the CMc Matrix");
         for (int q1=0; q1<matrixIn.col; q1++){
                 for (int h2=0; h2<matrixIn.col; h2++){
                     DecimalFormat df2 = new DecimalFormat("#.##");
                  System.out.print(df2.format(cmc[q1][h2].contain)+"         ");
             }
              System.out.println();
         }

        cmrc=new Cell[matrixIn.row][matrixIn.col];
        for (int m2=0; m2<matrixIn.row; m2++)
            for (int n2=0; n2<matrixIn.col; n2++)
            {

                    int sumr=0;
                    int sumc=0;
                    for (int k2=0; k2<matrixIn.col; k2++)
                    { if ((matrixIn.mat[m2][k2].contain==1) )
                            {
                                sumr++;}}
                     for (int k3=0; k3<matrixIn.row; k3++)
                    { if ((matrixIn.mat[k3][n2].contain==1) )
                            {
                                sumc++;}}

                    float h3=(float)((2*matrixIn.mat[m2][n2].contain)/(sumr+sumc));
                    cmrc[m2][n2]=new Cell();
                    cmrc[m2][n2].contain=h3;

                }
        //show the matrix CMrc

         System.out.println();
         System.out.println("this is the CMrc Matrix");
        for (int q3=0; q3<matrixIn.row; q3++){
                 for (int h4=0; h4<matrixIn.col; h4++){
                     DecimalFormat df3 = new DecimalFormat("#.##");
                  System.out.print(df3.format(cmrc[q3][h4].contain)+"          ");
             }
              System.out.println();
         }

            }
    public void tcmrc()
    { trans= new Cell[cmcol][cmrow];
      for (int i1=0; i1<cmrow; i1++){
                 for (int j1=0; j1<cmcol; j1++)
                 {trans[j1][i1]=cmrc[i1][j1];}}


             //show the matrix TCMrc
          System.out.println();
          System.out.println("this is the transposed CMrc Matrix");
        for (int q4=0; q4<cmcol; q4++){
                 for (int h5=0; h5<cmrow; h5++){
                     DecimalFormat df4 = new DecimalFormat("#.##");
                  System.out.print(df4.format(trans[q4][h5].contain)+"       ");

             }
              System.out.println();
         }
    }
public Cell[][] makeCm()
    { int rc=cmrow+cmcol;
    cm= new Cell[rc][rc];
    for (int cr=0; cr<cmcol; cr++ )
    {
        for (int cc = 0; cc < cmcol; cc++){
            cm[cr][cc]=new Cell();
            cm[cr][cc].contain=cmc[cr][cc].contain;
```

```
        cm[cr][cc].rowNum=cr+1;
        cm[cr][cc].colNum=cc+1;}}
    for (int tr=0;tr<cmcol; tr++  ){
        for(int tc=cmcol; tc<rc; tc++){
            cm[tr][tc]=new Cell();
            cm[tr][tc].contain=trans[tr][tc-cmcol].contain;
       cm[tr][tc].rowNum=tr+1;
        cm[tr][tc].colNum=tc+1;}}
    for (int rcr=cmcol; rcr<rc; rcr++){
        for(int rcc=0; rcc<cmcol; rcc++){
            cm[rcr][rcc]=new Cell();
            cm[rcr][rcc].contain=cmrc[rcr-cmcol][rcc].contain;
        cm[rcr][rcc].rowNum=rcr+1;
        cm[rcr][rcc].colNum=rcc+1;}}
    for (int rr=cmcol; rr<rc; rr++){
        for(int rc1=cmcol; rc1<rc; rc1++){
            cm[rr][rc1]=new Cell();
            cm[rr][rc1].contain=cmr[rr-cmcol][rc1-cmcol].contain;
       cm[rr][rc1].rowNum=rr+1;
        cm[rr][rc1].colNum=rc1+1;}}


    //show the matrix CM
         System.out.println();
         System.out.println("this is the CM Matrix");
        for (int q5=0; q5<rc; q5++){
                  for (int h6=0; h6<rc; h6++){
                      DecimalFormat df5 = new DecimalFormat("#.##");
                      System.out.print(df5.format(cm[q5][h6].contain)+"          ");
             }
             System.out.println();
         }
         return cm; }    }
```

```
 class Node
        {
            int first;
            int second;
            int branch;
        }
```

```
class Tree
   {

    ArrayList<Node> sequence= new ArrayList<Node>();
    int originRow;
    int originCol;

    public Cell[][] makeRevised(Cm fc)
    {int nr=fc.cmrow;
     int nc=fc.cmcol;
     originRow=nr;
     originCol=nc;
     int to=nr+nc;
     Cell[][] revised=new Cell[to][to];
     for (int i=0; i<to; i++)
         for (int j=0; j<to; j++)
         {if (i<j)

                 {
             revised[i][j]= new Cell();
             revised[i][j].contain=0;
             revised[i][j].rowNum=fc.cm[i][j].rowNum;
             revised[i][j].colNum=fc.cm[i][j].colNum;
             revised[i][j].extraRow=0;
             revised[i][j].extraCol=0;
          }
 else
                 {
              revised[i][j]= new Cell();
             revised[i][j].contain=fc.cm[i][j].contain;
```

```
             revised[i][j].rowNum=fc.cm[i][j].rowNum;
             revised[i][j].colNum=fc.cm[i][j].colNum;
             revised[i][j].extraRow=0;
             revised[i][j].extraCol=0;
             }
 }
//show the revised of matrix CM
         System.out.println();
         System.out.println("this is the revised of CM Matrix");
        for (int q6=0; q6<to; q6++){
                 for (int h7=0; h7<to; h7++){
                     DecimalFormat df6 = new DecimalFormat("#.##");
                   System.out.print(revised[q6][h7].rowNum);
                    System.out.print(revised[q6][h7].colNum+"  ");
                       System.out.print(df6.format(revised[q6][h7].contain)+"
");
             }
             System.out.println();}
     return revised;

    }

    public void makeTree(Cell[][] revisedCm)
    {

        int counter=10000;
        int counter1=revisedCm.length;
        while (counter1 != 1)
        {
        Cell newnode=findMax(revisedCm);
        Node n=new Node();
        if(newnode.extraRow==0)
        {
            n.first = newnode.rowNum;}
        else { n.first=newnode.extraRow;}
        if(newnode.extraCol==0)
        {
            n.second = newnode.colNum;}
        else  {n.second=newnode.extraCol;}
        n.branch=counter;
        counter++;
        sequence.add(n);
        revisedCm[(newnode.rowNum)-1][(newnode.colNum)-1].contain=0;
        revisedCm[(newnode.colNum)-1][(newnode.rowNum)-1].contain=0;
        for(int jn=0; jn<revisedCm.length; jn++)
        {revisedCm[(newnode.colNum)-1][jn].contain=((revisedCm[(newnode.colNum)-
1][jn].contain)+(revisedCm[(newnode.rowNum)-1][jn].contain))/2;
         revisedCm[jn][(newnode.colNum)-1].contain=((revisedCm[jn][(newnode.colNum)-
1].contain)+(revisedCm[jn][(newnode.rowNum)-1].contain))/2;
        revisedCm[(newnode.colNum)-1][jn].extraRow=n.branch;
         revisedCm[jn][(newnode.colNum)-1].extraCol=n.branch;
        }

        for(int in=0; in<revisedCm.length; in++)
        {
            revisedCm[in][(newnode.rowNum) - 1].contain = 0;
            revisedCm[(newnode.rowNum)-1][in].contain=0;
        }

        for (int ir=0; ir<revisedCm.length; ir++){
         for (int jr=0; jr<revisedCm.length; jr++)
         {if (ir<jr)

                 {
             revisedCm[ir][jr].contain=0;


    }
             }
        }
        counter1--;
```

```
    }

        }

   public ArrayList<Cell> findLeafs(ArrayList<Node> seqq, int point)
   {
        int y5=0;
        int fi;
        int se;
        ArrayList<Cell> leafs=new ArrayList<Cell>();
        ArrayList<Cell> nonLeafs=new ArrayList<Cell>();
        if (point>9999){
        for (int x5=0; x5<seqq.size();x5++)
        {
            if (seqq.get(x5).branch==point)
            {
               y5=x5;
               break;
            }
        }

        fi=seqq.get(y5).first;
        se=seqq.get(y5).second;
        if (fi<9999)
        {
            Cell newLeaf = new Cell();
            newLeaf.contain=fi;
            leafs.add(newLeaf);
        }
else
        {
         Cell newNonLeaf=new Cell();
         newNonLeaf.contain=fi;
         nonLeafs.add(newNonLeaf);
}
        if (se<9999)
        {
            Cell newLeaf = new Cell();
            newLeaf.contain=se;
            leafs.add(newLeaf);
        }
        else
        {
         Cell newNonLeaf=new Cell();
         newNonLeaf.contain=se;
         nonLeafs.add(newNonLeaf);
}

             while (!nonLeafs.isEmpty())
              {
                 for (int x9=0; x9<nonLeafs.size(); x9++)
                     {int point1=(int)nonLeafs.get(x9).contain;
                        for(int y9=0; y9<seqq.size(); y9++)
                         {
            if (seqq.get(y9).branch==point1)
            {
                nonLeafs.remove(x9);
               if (seqq.get(y9).first<9999)
               {
                   Cell newLeaf=new Cell();
                   newLeaf.contain=seqq.get(y9).first;
                   leafs.add(newLeaf);
               }
             else
               {
                   Cell newLeaf=new Cell();
                   newLeaf.contain=seqq.get(y9).first;
                   nonLeafs.add(newLeaf);
                }
               if (seqq.get(y9).second<9999)
               {
```

```
                    Cell newLeaf=new Cell();
                    newLeaf.contain=seqq.get(y9).second;
                    leafs.add(newLeaf);
                }
             else
                {
                    Cell newLeaf=new Cell();
                    newLeaf.contain=seqq.get(y9).second;
                    nonLeafs.add(newLeaf);
                 }

                   break;
            }

        }
              }

        }
        }
 else
        {
            Cell newLeaf2=new Cell();
            newLeaf2.contain=point;
            leafs.add(newLeaf2);
 }
        // print leafs
        System.out.println();
for (int x10=0; x10<leafs.size(); x10++)
{
    System.out.println(leafs.get(x10).contain+"   ");
}
return leafs;
    }



    public Cell findMax(Cell[][] toFind)
    {  Cell choose=new Cell();
        float max=toFind[0][0].contain;
        for(int it=0; it<toFind.length; it++)
            for (int jt=0; jt<toFind.length; jt++)
               {
                if (toFind[it][jt].contain>max)
                {
                max = toFind[it][jt].contain;
                choose.colNum=toFind[it][jt].colNum;
                choose.rowNum=toFind[it][jt].rowNum;
                choose.extraRow=toFind[it][jt].extraRow;
                choose.extraCol=toFind[it][jt].extraCol;
                choose.contain=max;
                   }
            }
    return choose;
    }
public int[][] findSequence (ArrayList<Node> se)
    {
         int irow=0;
         int jcol=0;
         int[][] seq;
         if (originRow>originCol)
         {seq=new int[2][originRow];}
         else
         { seq = new int[2][originCol];}
         for (int u = 0; u < se.size(); u++)
         {
             if ((se.get(u).first<= originCol)&&(se.get(u).first>0))
             {
                 seq[1][jcol] = se.get(u).first;
                 jcol++;
             }
```

```
            if (se.get(u).first> originCol && se.get(u).first<9999 &&
se.get(u).first>0 )
            {
                seq[0][irow]= se.get(u).first - originCol;
                irow++;
            }
            if (se.get(u).second<= originCol && se.get(u).second>0)
            {
                seq[1][jcol] = se.get(u).second;
                jcol++;
            }
            if (se.get(u).second> originCol && se.get(u).second<9999 &&
se.get(u).second>0 )
            {
                seq[0][irow]= se.get(u).second - originCol;
                irow++;
            }
            if (se.get(u).branch<= originCol && se.get(u).branch>0)
            {
                seq[1][jcol] = se.get(u).branch;
                jcol++;
            }
            if (se.get(u).branch> originCol && se.get(u).branch<9999 &&
se.get(u).branch>0 )
            {
                seq[0][irow]= se.get(u).branch - originCol;
                irow++;
            }

        }

        // show seq array

         for (int z=0; z<originRow; z++)
         {System.out.print(seq[0][z]+"    ");

         }
        System.out.println();
        for (int y=0; y<originCol; y++)
         {System.out.print(seq[1][y]+"    ");

         }

        return seq;

         }
    public float findDependency(int point1, int point2, Cell[][] revisedCm1)
   {
        float dep=0;
        ArrayList <Cell> point1List=findLeafs(sequence, point1);
        ArrayList <Cell> point2List=findLeafs(sequence, point2);
        for (int k=0; k<point1List.size(); k++)
            for(int k1=0; k1<point2List.size(); k1++)
            {
                dep=dep+revisedCm1[((int)point1List.get(k).contain)-
1][((int)point2List.get(k1).contain)-1].contain;

            }
        return dep;
    }

    public int findIndex(int m)
   {   int o=-1;
        for (int s=0; s<sequence.size(); s++)
            if (sequence.get(s).branch==m)
            {  o=s;
                break;}
        return o;
    }

    public ArrayList<Node> reordering(Cell[][] revisedCm2)
```

```
    {
          ArrayList<Node> sequence1= new ArrayList<Node>();
          int count=1;
          int a=sequence.size();
          Node adding=new Node();
          adding.branch=sequence.get(a-1).branch;
          adding.first=sequence.get(a-1).first;
          adding.second=sequence.get(a-1).second;
          sequence1.add(adding);
          while(count!= sequence.size())
          {  ArrayList<Node> sequence2= new ArrayList<Node>();
              for(int b=0; b<sequence1.size();b++)
                  { if
((sequence1.get(b).first>9999)||(sequence1.get(b).second>9999)){
               if (sequence1.get(b).first>9999)
               {   int pr=findIndex(sequence1.get(b).first);
                   Node adding1=new Node();
                   adding1.branch=sequence.get(pr).branch;
                   adding1.first=sequence.get(pr).first;
                   adding1.second=sequence.get(pr).second;
                   sequence2.add(adding1);
                   count++;
               }
 else
               {Node adding1=new Node();
                   adding1.branch=sequence1.get(b).first;
                   adding1.first=-1;
                   adding1.second=-1;
                   sequence2.add(adding1);

 }
               if (sequence1.get(b).second>9999)
               {   int pr=findIndex(sequence1.get(b).second);
                   Node adding1=new Node();
                   adding1.branch=sequence.get(pr).branch;
                   adding1.first=sequence.get(pr).first;
                   adding1.second=sequence.get(pr).second;
                   sequence2.add(adding1);
                   count++;
               }
               else
               {Node adding1=new Node();
                   adding1.branch=sequence1.get(b).second;
                   adding1.first=-1;
                   adding1.second=-1;
                   sequence2.add(adding1);

 }
                      }

 else
           {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).branch;
                  adding1.first=sequence1.get(b).first;
                  adding1.second=sequence1.get(b).second;
                  sequence2.add(adding1);
                     }

 }
            if(sequence2.size()>2)
            { int co=sequence2.size()+100;
              while(co!=0){
              for (int u=1; u<(sequence2.size())-1;u++)
                {  float fdis;
                   float sdis;
                   fdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u).branch, revisedCm2);
                   sdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u+1).branch, revisedCm2);
                   if(sdis>fdis)
                   {
```

```
                    Node sub=new Node();
                  sub.branch=sequence2.get(u+1).branch;
                  sub.first=sequence2.get(u+1).first;
                  sub.second=sequence2.get(u+1).second;
                  sequence2.get(u+1).branch= sequence2.get(u).branch;
                  sequence2.get(u+1).first=sequence2.get(u).first;
                  sequence2.get(u+1).second=sequence2.get(u).second;
                  sequence2.get(u).branch=sub.branch;
                  sequence2.get(u).first=sub.first;
                  sequence2.get(u).second=sub.second;
                   }
                }
          co--;
              }
            }
            sequence1=sequence2;
          }
         return sequence1;
         }
     }
```

```
public class Main {

        public static void main(String[] args) {
        System.out.println("enter the number of rows");
        Scanner getRow= new Scanner(System.in);
        int row2=getRow.nextInt();
        System.out.println("enter the number of columns");
        Scanner getCol= new Scanner(System.in);
        int col2=getCol.nextInt();
        Matrix matrix1=new Matrix(row2,col2);
        System.out.println();
        matrix1.showMatrix();
        Cm cm1=new Cm(matrix1);
        cm1.tcmrc();
        Cell[][] finalCm=cm1.makeCm();
        Tree mtree= new Tree();
        finalCm=mtree.makeRevised(cm1);
        mtree.makeTree(finalCm);
        mtree.printSequence();
        ArrayList<Node> seque=mtree.reordering(finalCm);
        int[][] seq2=mtree.findSequence(seque);
        System.out.println();
        matrix1.rearrange(seq2);

    }

}
```

Figure B1. The Original Code for the Case Study OOP

```
package clustering1;
import java.text.DecimalFormat;
import java.util.*;

class Matrix
{   int row;
    int col;
    int[][] mat;

         public Matrix(int row1, int col1){
         row=row1;
         col=col1;
         mat= new int[row][col];
                  for (int i=0; i<row; i++){
                     int p=i+1;
                     System.out.println("enter the entities of row number"+ p );
                  for (int j=0; j<col; j++){
                   Scanner getEntity= new Scanner(System.in);
                   int next;
                   next= getEntity.nextInt();
                     mat[i][j]=next;

                  }
                                                    }
                      }
         public int[][] rearrange(int[][] sequ1)
        {
    int[][] mat1=new int[row][col];
    for(int d=0; d<row; d++){
                      for (int f=0; f<col; f++){
                          int h=(sequ1[0][d])-1;
                          mat1[d][f]=mat[h][f];
        }
             }

    int[][] mat2=new int[row][col];
    for(int d1=0; d1<col; d1++){
                      for (int f1=0; f1<row; f1++){
                          int h8=(sequ1[1][d1])-1;
            mat2[f1][d1]=mat1[f1][h8];
        }
             }


    int [][] mat3=new int[row+1][col+1];
    mat3[0][0]=0;
    for (int px=0; px<row; px++)
        mat3[px+1][0]=sequ1[0][px];
    for (int py=0; py<col; py++)
        mat3[0][py+1]=sequ1[1][py];
    for (int xy=0; xy<row; xy++)
        for (int xy1=0; xy1<col; xy1++)
        {
            mat3[xy+1][xy1+1]=mat2[xy][xy1];
        }

    for(int d5=0; d5<row+1; d5++){
                      for (int f5=0; f5<col+1; f5++){
                  System.out.print(mat3[d5][f5]+" ");
        }
                      System.out.println();
                      }

    return mat2;
}

         public void showMatrix()
    {    System.out.println("this is the original matrix");
       for (int i=0; i<row; i++){
             for (int j=0; j<col; j++){
                 System.out.print(mat[i][j]+"  ");
```

```
            }
            System.out.println();
        }
}
}
```

```
class Cm
{
    float[][] cm;
    float[][] cmr;
    float[][] cmc;
    float[][] cmrc;
    float[][] trans;
    int cmrow;
    int cmcol;

    public Cm(Matrix matrixIn)
    {
        cmrow= matrixIn.row;
        cmcol= matrixIn.col;
        cmr=new float[matrixIn.row][matrixIn.row];
        for (int m=0; m<matrixIn.row; m++)
            for (int n=0; n<matrixIn.row; n++)
            {
                if (m==n)
                {
                    cmr[m][n]=0;
                }
                else
                {
                    int max=0;
                    int min=0;
                    for (int k=0; k<matrixIn.col; k++)
                    { if ((matrixIn.mat[m][k]==1) || (matrixIn.mat[n][k]==1))
                            {
                                max++;}
                      if ((matrixIn.mat[m][k]==1) && (matrixIn.mat[n][k]==1))
                      {
                          min++;}
                    }
                    float h=(float)min/max;
                    cmr[m][n]=h;
                }
                }
        //show the matrix CMr
         System.out.println();
         System.out.println("this is the CMr Matrix");
        for (int q=0; q<matrixIn.row; q++){
                 for (int h=0; h<matrixIn.row; h++){
                  DecimalFormat df1 = new DecimalFormat("#.##");
                  System.out.print(df1.format(cmr[q][h])+"      ");
             }
             System.out.println();
         }

        cmc=new float[matrixIn.col][matrixIn.col];
        for (int m1=0; m1<matrixIn.col; m1++)
            for (int n1=0; n1<matrixIn.col; n1++)
            {
                if (m1==n1)
                {
                    cmc[m1][n1]=0;
                }
                else
                {
                    int max1=0;
                    int min1=0;
                    for (int k1=0; k1<matrixIn.row; k1++)
                    { if ((matrixIn.mat[k1][m1]==1) || (matrixIn.mat[k1][n1]==1))
                            {
                                max1++;}
```

```
                    if ((matrixIn.mat[k1][m1]==1) && (matrixIn.mat[k1][n1]==1))
                    {
                        min1++;}
                   }
                   float h1=(float)min1/max1;
                   cmc[m1][n1]=h1;
               }
               }
      // show the matrix CMc

       System.out.println();
       System.out.println("this is the CMc Matrix");
       for (int q1=0; q1<matrixIn.col; q1++){
               for (int h2=0; h2<matrixIn.col; h2++){
                   DecimalFormat df2 = new DecimalFormat("#.##");
                System.out.print(df2.format(cmc[q1][h2])+"         ");
           }
           System.out.println();
       }

      cmrc=new float[matrixIn.row][matrixIn.col];
      for (int m2=0; m2<matrixIn.row; m2++)
          for (int n2=0; n2<matrixIn.col; n2++)
          {

                   int sumr=0;
                   int sumc=0;
                   for (int k2=0; k2<matrixIn.col; k2++)
                   { if (matrixIn.mat[m2][k2]==1)
                           {
                               sumr++;}}
                    for (int k3=0; k3<matrixIn.row; k3++)
                   { if (matrixIn.mat[k3][n2]==1 )
                           {
                               sumc++;}}
                   float h3=matrixIn.mat[m2][n2];
                   cmrc[m2][n2]=(float)((2*h3)/(sumr+sumc));

               }
      //show the matrix CMrc

       System.out.println();
       System.out.println("this is the CMrc Matrix");
      for (int q3=0; q3<matrixIn.row; q3++){
               for (int h4=0; h4<matrixIn.col; h4++){
                   DecimalFormat df3 = new DecimalFormat("#.##");
                System.out.print(df3.format(cmrc[q3][h4])+"          ");
           }
           System.out.println();
       }

          }
    public void tcmrc()
    { trans= new float[cmcol][cmrow];
      for (int i1=0; i1<cmrow; i1++){
               for (int j1=0; j1<cmcol; j1++)
               {trans[j1][i1]=cmrc[i1][j1];}}


           //show the matrix TCMrc
       System.out.println();
       System.out.println("this is the transposed CMrc Matrix");
      for (int q4=0; q4<cmcol; q4++){
               for (int h5=0; h5<cmrow; h5++){
                   DecimalFormat df4 = new DecimalFormat("#.##");
                System.out.print(df4.format(trans[q4][h5])+"       ");

           }
           System.out.println();
       }
```

```
    }
public float[][] makeCm()
    { int rc=cmrow+cmcol;
    cm= new float[rc][rc];
    for (int cr=0; cr<cmcol; cr++ )
    {
        for (int cc = 0; cc < cmcol; cc++){
            cm[cr][cc]=cmc[cr][cc];
        }}
    for (int tr=0;tr<cmcol; tr++  ){
        for(int tc=cmcol; tc<rc; tc++){
            cm[tr][tc]=trans[tr][tc-cmcol];
        }}
    for (int rcr=cmcol; rcr<rc; rcr++){
        for(int rcc=0; rcc<cmcol; rcc++){
            cm[rcr][rcc]=cmrc[rcr-cmcol][rcc];
        }}
    for (int rr=cmcol; rr<rc; rr++){
        for(int rc1=cmcol; rc1<rc; rc1++){
            cm[rr][rc1]=cmr[rr-cmcol][rc1-cmcol];
        }}


    //show the matrix CM
         System.out.println();
         System.out.println("this is the CM Matrix");
        for (int q5=0; q5<rc; q5++){
                  for (int h6=0; h6<rc; h6++){
                      DecimalFormat df5 = new DecimalFormat("#.##");
                      System.out.print(df5.format(cm[q5][h6])+"          ");
             }
              System.out.println();
         }
         return cm;
}

    }
```

```
class Node
        {
            int first;
            int second;
            int branch;
        }
```

```
class Tree
   {

    ArrayList<Node> sequence= new ArrayList<Node>();
    int originRow;
    int originCol;

    public float[][] makeRevised(Cm fc)
    {int nr=fc.cmrow;
     int nc=fc.cmcol;
     originRow=nr;
     originCol=nc;
     int to=nr+nc;
     float[][] revised=new float[to][to];
     for (int i=0; i<to; i++)
         for (int j=0; j<to; j++)
         {if (i<j)

                 {
             revised[i][j]=0;
          }
 else
                 {
             revised[i][j]=fc.cm[i][j];
```

```
            }
 }
//show the revised of matrix CM
         System.out.println();
         System.out.println("this is the revised of CM Matrix");
        for (int q6=0; q6<to; q6++){
                 for (int h7=0; h7<to; h7++){
                     DecimalFormat df6 = new DecimalFormat("#.##");
                     System.out.print(df6.format(revised[q6][h7])+"          ");
            }
            System.out.println();}
     return revised;

    }

     public void makeTree(float[][] revisedCm)
    {

         int y5=revisedCm.length;
         int[][] helpr=new int[y5][y5];
         int[][] helpc=new int[y5][y5];
         for(int l3=0; l3<y5; l3++)
             for (int l4=0; l4<y5; l4++)
             {
                 helpr[l3][l4]=0;
                 helpc[l3][l4]=0;
             }

        int counter=10000;
        int counter1=revisedCm.length;
        while (counter1 != 1)
        {
            int [][] newnode=new int[1][2];

        newnode=findMax(revisedCm);
        int hr=newnode[0][0];
        int hc=newnode[0][1];
        Node n=new Node();
        if(helpr[hr][0]==0)
        {
            n.first = hr+1;}
        else { n.first=helpr[hr][0];}
        if(helpc[0][hc]==0)
        {
            n.second = hc+1;}
        else  {n.second=helpc[0][hc];}
        n.branch=counter;
        counter++;
        sequence.add(n);
        revisedCm[hr][hc]=0;
        revisedCm[hc][hr]=0;
        for(int jn=0; jn<revisedCm.length; jn++)
        {revisedCm[hc][jn]=((revisedCm[hc][jn])+(revisedCm[hr][jn]))/2;
         revisedCm[jn][hc]=((revisedCm[jn][hc])+(revisedCm[jn][hr]))/2;
        helpr[hc][jn]=n.branch;
         helpc[jn][hc]=n.branch;
        }

        for(int in=0; in<revisedCm.length; in++)
        {
            revisedCm[in][hr]= 0;
            revisedCm[hr][in]=0;
        }

        for (int ir=0; ir<revisedCm.length; ir++){
         for (int jr=0; jr<revisedCm.length; jr++)
         {if (ir<jr)

                 {
             revisedCm[ir][jr]=0;
```

```
    }
            }
        }
        counter1--;
    }

        }


     public int[][] findMax(float[][] toFind)
    {  int[][] choose=new int[1][2];
        float max=toFind[0][0];
        for(int it=0; it<toFind.length; it++)
            for (int jt=0; jt<toFind.length; jt++)
               {
                if (toFind[it][jt]>max)
                {
                max = toFind[it][jt];
                choose[0][0]=it;
                choose[0][1]=jt;
                   }
            }
    return choose;
    }

     public ArrayList<Float> findLeafs(ArrayList<Node> seqq, int point)
    {
        int y5=0;
        int fi;
        int se;
        ArrayList<Float> leafs=new ArrayList<Float>();
        ArrayList<Float> nonLeafs=new ArrayList<Float>();
        if (point>9999){
        for (int x5=0; x5<seqq.size();x5++)
        {
            if (seqq.get(x5).branch==point)
            {
               y5=x5;
               break;
            }
        }

        fi=seqq.get(y5).first;
        se=seqq.get(y5).second;
        if (fi<9999)
        {
            float newLeaf = fi;
            leafs.add(newLeaf);
        }
else
        {
         float newNonLeaf=fi;
         nonLeafs.add(newNonLeaf);
}
        if (se<9999)
        {
            float newLeaf =se;
            leafs.add(newLeaf);
        }
        else
        {
         float newNonLeaf=se;
         nonLeafs.add(newNonLeaf);
}

             while (!nonLeafs.isEmpty())
              {
                 for (int x9=0; x9<nonLeafs.size(); x9++)
                     {float point11=nonLeafs.get(x9);
                      int point1=(int)point11;
```

```
                        for(int y9=0; y9<seqq.size(); y9++)
                         {
            if (seqq.get(y9).branch==point1)
            {
                nonLeafs.remove(x9);
               if (seqq.get(y9).first<9999)
               {
                   float newLeaf=seqq.get(y9).first;
                   leafs.add(newLeaf);
               }
             else
               {
                   float newLeaf=seqq.get(y9).first;
                   nonLeafs.add(newLeaf);
                }
               if (seqq.get(y9).second<9999)
               {
                   float newLeaf=seqq.get(y9).second;
                   leafs.add(newLeaf);
               }
             else
               {
                   float newLeaf=seqq.get(y9).second;
                   nonLeafs.add(newLeaf);
                }

                  break;
            }

        }
              }

        }
        }
 else
        {
            float newLeaf2=point;
            leafs.add(newLeaf2);
 }
        // print leafs
        System.out.println();
for (int x10=0; x10<leafs.size(); x10++)
{
    System.out.println(leafs.get(x10)+"   ");
}
return leafs;
    }


     public int[][] findSequence (ArrayList<Node> se)
    {
         int irow=0;
         int jcol=0;
         int[][] seq;
         if (originRow>originCol)
         {seq=new int[2][originRow];}
         else
         { seq = new int[2][originCol];}
         for (int u = 0; u < se.size(); u++)
         {
             if ((se.get(u).first<= originCol)&&(se.get(u).first>0))
             {
                 seq[1][jcol] = se.get(u).first;
                 jcol++;
             }
             if (se.get(u).first> originCol && se.get(u).first<9999 &&
se.get(u).first>0 )
             {
                 seq[0][irow]= se.get(u).first - originCol;
                 irow++;
             }
```

```
            if (se.get(u).second<= originCol && se.get(u).second>0)
            {
                seq[1][jcol] = se.get(u).second;
                jcol++;
            }
            if (se.get(u).second> originCol && se.get(u).second<9999 &&
se.get(u).second>0 )
            {
                seq[0][irow]= se.get(u).second - originCol;
                irow++;
            }
            if (se.get(u).branch<= originCol && se.get(u).branch>0)
            {
                seq[1][jcol] = se.get(u).branch;
                jcol++;
            }
            if (se.get(u).branch> originCol && se.get(u).branch<9999 &&
se.get(u).branch>0 )
            {
                seq[0][irow]= se.get(u).branch - originCol;
                irow++;
            }

        }

        // show seq array

         for (int z=0; z<originRow; z++)
         {System.out.print(seq[0][z]+"    ");

         }
        System.out.println();
        for (int y=0; y<originCol; y++)
         {System.out.print(seq[1][y]+"    ");

         }

        return seq;

         }
    public float findDependency(int point1, int point2, float[][] revisedCm1)
    {
        float dep=0;
        ArrayList <Float> point1List=findLeafs(sequence, point1);
        ArrayList <Float> point2List=findLeafs(sequence, point2);
        for (int k=0; k<point1List.size(); k++)
            for(int k1=0; k1<point2List.size(); k1++)
            {
                float hy=point1List.get(k)-1;
                int hy1=(int)hy;
                float hx=point2List.get(k1)-1;
                int hx1=(int)hx;
                dep=dep+revisedCm1[hy1][hx1];

            }
        return dep;
    }

    public int findIndex(int m)
    {   int o=-1;
        for (int s=0; s<sequence.size(); s++)
            if (sequence.get(s).branch==m)
            {  o=s;
                break;}
        return o;
    }

    public ArrayList<Node> reordering(float[][] revisedCm2)
    {
        ArrayList<Node> sequence1= new ArrayList<Node>();
        int count=1;
```

```
         int a=sequence.size();
         Node adding=new Node();
         adding.branch=sequence.get(a-1).branch;
         adding.first=sequence.get(a-1).first;
         adding.second=sequence.get(a-1).second;
         sequence1.add(adding);
         while(count!= sequence.size())
         {  ArrayList<Node> sequence2= new ArrayList<Node>();
             for(int b=0; b<sequence1.size();b++)
                 { if
((sequence1.get(b).first>9999)||(sequence1.get(b).second>9999)){
              if (sequence1.get(b).first>9999)
              {   int pr=findIndex(sequence1.get(b).first);
                  Node adding1=new Node();
                  adding1.branch=sequence.get(pr).branch;
                  adding1.first=sequence.get(pr).first;
                  adding1.second=sequence.get(pr).second;
                  sequence2.add(adding1);
                  count++;
              }
 else
              {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).first;
                  adding1.first=-1;
                  adding1.second=-1;
                  sequence2.add(adding1);

 }
              if (sequence1.get(b).second>9999)
              {   int pr=findIndex(sequence1.get(b).second);
                  Node adding1=new Node();
                  adding1.branch=sequence.get(pr).branch;
                  adding1.first=sequence.get(pr).first;
                  adding1.second=sequence.get(pr).second;
                  sequence2.add(adding1);
                  count++;
              }
              else
              {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).second;
                  adding1.first=-1;
                  adding1.second=-1;
                  sequence2.add(adding1);

 }
                     }

 else
           {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).branch;
                  adding1.first=sequence1.get(b).first;
                  adding1.second=sequence1.get(b).second;
                  sequence2.add(adding1);
                     }

 }
            if(sequence2.size()>2)
            { int co=sequence2.size()+100;
              while(co!=0){
              for (int u=1; u<(sequence2.size())-1;u++)
                {  float fdis;
                   float sdis;
                   fdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u).branch, revisedCm2);
                   sdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u+1).branch, revisedCm2);
                   if(sdis>fdis)
                   {
                       Node sub=new Node();
                  sub.branch=sequence2.get(u+1).branch;
                  sub.first=sequence2.get(u+1).first;
```

```
                    sub.second=sequence2.get(u+1).second;
                    sequence2.get(u+1).branch= sequence2.get(u).branch;
                    sequence2.get(u+1).first=sequence2.get(u).first;
                    sequence2.get(u+1).second=sequence2.get(u).second;
                    sequence2.get(u).branch=sub.branch;
                    sequence2.get(u).first=sub.first;
                    sequence2.get(u).second=sub.second;
                     }
                  }
           co--;
               }
             }
             sequence1=sequence2;
           }

          // print sequence
          System.out.println();
          for (int kh = 0; kh < sequence1.size(); kh++)
          {
              System.out.println(sequence1.get(kh).first+"
"+sequence1.get(kh).second+"   "+sequence1.get(kh).branch);
          }
          return sequence1;
          }
      }
```

```
public class Main {

    public static void main(String[] args) {
        System.out.println("enter the number of rows");
        Scanner getRow= new Scanner(System.in);
        int row2=getRow.nextInt();
        System.out.println("enter the number of columns");
        Scanner getCol= new Scanner(System.in);
        int col2=getCol.nextInt();
        Matrix matrix1=new Matrix(row2,col2);
        System.out.println();
        matrix1.showMatrix();
        Cm cm1=new Cm(matrix1);
        cm1.tcmrc();
        float[][] finalCm=cm1.makeCm();
        Tree mtree= new Tree();
        finalCm=mtree.makeRevised(cm1);
        mtree.makeTree(finalCm);
        mtree.printSequence();
        ArrayList<Node> seque=mtree.reordering(finalCm);
        int[][] seq2=mtree.findSequence(seque);
        System.out.println();
        matrix1.rearrange(seq2);
    }
}
```

Figure B2. The Modified Code of the Case Study OOP Based on Solution 1

```
package clustering2;
import java.text.DecimalFormat;
import java.util.*;


class Cell
        {
    int rowNum;
    int colNum;
    float contain;
    int extraRow;
    int extraCol;
}
```

```
class Matrix
{   int row;
    int col;
    int[][] mat;

         public Matrix(int row1, int col1){
         row=row1;
         col=col1;
         mat= new int[row][col];
                  for (int i=0; i<row; i++){
                     int p=i+1;
                     System.out.println("enter the entities of row number"+ p );
                  for (int j=0; j<col; j++){
                   Scanner getEntity= new Scanner(System.in);
                   int next;
                   next= getEntity.nextInt();
                     mat[i][j]=next;
                  }
                                                  }
                      }
         public int[][] rearrange(int[][] sequ1)
        {
    int[][] mat1=new int[row][col];
    for(int d=0; d<row; d++){
                      for (int f=0; f<col; f++){
                          int h=(sequ1[0][d])-1;
                    mat1[d][f]=mat[h][f];
        }
             }

    int[][] mat2=new int[row][col];
    for(int d1=0; d1<col; d1++){
                      for (int f1=0; f1<row; f1++){
                          int h8=(sequ1[1][d1])-1;
            mat2[f1][d1]=mat1[f1][h8];
        }
             }


    int [][] mat3=new int[row+1][col+1];
    mat3[0][0]=0;
    for (int px=0; px<row; px++)
        mat3[px+1][0]=sequ1[0][px];
    for (int py=0; py<col; py++)
        mat3[0][py+1]=sequ1[1][py];
    for (int xy=0; xy<row; xy++)
        for (int xy1=0; xy1<col; xy1++)
        {
            mat3[xy+1][xy1+1]=mat2[xy][xy1];
        }

    for(int d5=0; d5<row+1; d5++){
                      for (int f5=0; f5<col+1; f5++){
                 System.out.print(mat3[d5][f5]+" ");
        }
                      System.out.println();
                      }
```

```
    return mat2;
}

     public void showMatrix()
    {    System.out.println("this is the original matrix");
       for (int i=0; i<row; i++){
            for (int j=0; j<col; j++){
                System.out.print(mat[i][j]+"  ");
            }
            System.out.println();
        }
}
}
```

```
class Cm
{
    float[][] cm;
    float[][] cmr;
    float[][] cmc;
    float[][] cmrc;
    float[][] trans;
    int cmrow;
    int cmcol;

    public Cm(Matrix matrixIn)
    {
        cmrow= matrixIn.row;
        cmcol= matrixIn.col;
        cmr=new float[matrixIn.row][matrixIn.row];
        for (int m=0; m<matrixIn.row; m++)
            for (int n=0; n<matrixIn.row; n++)
            {
                if (m==n)
                {
                    cmr[m][n]=0;
                }
                else
                {
                    int max=0;
                    int min=0;
                    for (int k=0; k<matrixIn.col; k++)
                    { if ((matrixIn.mat[m][k]==1) || (matrixIn.mat[n][k]==1))
                            {
                                max++;}
                      if ((matrixIn.mat[m][k]==1) && (matrixIn.mat[n][k]==1))
                      {
                          min++;}
                    }
                    float h=(float)min/max;
                    //cmr[m][n]=new Cell();
                    cmr[m][n]=h;
                }
                }
        //show the matrix CMr
         System.out.println();
         System.out.println("this is the CMr Matrix");
        for (int q=0; q<matrixIn.row; q++){
                  for (int h=0; h<matrixIn.row; h++){
                   DecimalFormat df1 = new DecimalFormat("#.##");
                   System.out.print(df1.format(cmr[q][h])+"      ");
             }
             System.out.println();
         }

        cmc=new float[matrixIn.col][matrixIn.col];
        for (int m1=0; m1<matrixIn.col; m1++)
            for (int n1=0; n1<matrixIn.col; n1++)
            {
                if (m1==n1)
                {
```

```
                    cmc[m1][n1]=0;
                }
                else
                {
                    int max1=0;
                    int min1=0;
                    for (int k1=0; k1<matrixIn.row; k1++)
                    { if ((matrixIn.mat[k1][m1]==1) || (matrixIn.mat[k1][n1]==1))
                            {
                                max1++;}
                      if ((matrixIn.mat[k1][m1]==1) && (matrixIn.mat[k1][n1]==1))
                      {
                          min1++;}
                    }
                    float h1=(float)min1/max1;
                    cmc[m1][n1]=h1;
                }
                }
        // show the matrix CMc

         System.out.println();
         System.out.println("this is the CMc Matrix");
         for (int q1=0; q1<matrixIn.col; q1++){
                 for (int h2=0; h2<matrixIn.col; h2++){
                     DecimalFormat df2 = new DecimalFormat("#.##");
                  System.out.print(df2.format(cmc[q1][h2])+"         ");
             }
              System.out.println();
         }

        cmrc=new float[matrixIn.row][matrixIn.col];
        for (int m2=0; m2<matrixIn.row; m2++)
            for (int n2=0; n2<matrixIn.col; n2++)
            {

                    int sumr=0;
                    int sumc=0;
                    for (int k2=0; k2<matrixIn.col; k2++)
                    { if (matrixIn.mat[m2][k2]==1)
                            {
                                sumr++;}}
                     for (int k3=0; k3<matrixIn.row; k3++)
                    { if (matrixIn.mat[k3][n2]==1 )
                            {
                                sumc++;}}
                    float h3=matrixIn.mat[m2][n2];
                    cmrc[m2][n2]=(float)((2*h3)/(sumr+sumc));

                }
        //show the matrix CMrc

         System.out.println();
         System.out.println("this is the CMrc Matrix");
        for (int q3=0; q3<matrixIn.row; q3++){
                 for (int h4=0; h4<matrixIn.col; h4++){
                     DecimalFormat df3 = new DecimalFormat("#.##");
                  System.out.print(df3.format(cmrc[q3][h4])+"          ");
             }
              System.out.println();
         }

            }
    public void tcmrc()
    { trans= new float[cmcol][cmrow];
      for (int i1=0; i1<cmrow; i1++){
                 for (int j1=0; j1<cmcol; j1++)
                 {trans[j1][i1]=cmrc[i1][j1];}}


             //show the matrix TCMrc
          System.out.println();
```

```
         System.out.println("this is the transposed CMrc Matrix");
        for (int q4=0; q4<cmcol; q4++){
                 for (int h5=0; h5<cmrow; h5++){
                     DecimalFormat df4 = new DecimalFormat("#.##");
                  System.out.print(df4.format(trans[q4][h5])+"       ");

             }
             System.out.println();
         }

    }
public float[][] makeCm()
    { int rc=cmrow+cmcol;
    cm= new float[rc][rc];
    for (int cr=0; cr<cmcol; cr++ )
    {
        for (int cc = 0; cc < cmcol; cc++){
        cm[cr][cc]=cmc[cr][cc];
        }}
    for (int tr=0;tr<cmcol; tr++  ){
        for(int tc=cmcol; tc<rc; tc++){
            cm[tr][tc]=trans[tr][tc-cmcol];
        }}
    for (int rcr=cmcol; rcr<rc; rcr++){
        for(int rcc=0; rcc<cmcol; rcc++){
            cm[rcr][rcc]=cmrc[rcr-cmcol][rcc];
        }}
    for (int rr=cmcol; rr<rc; rr++){
        for(int rc1=cmcol; rc1<rc; rc1++){
            cm[rr][rc1]=cmr[rr-cmcol][rc1-cmcol];
        }}

    //show the matrix CM
         System.out.println();
         System.out.println("this is the CM Matrix");
        for (int q5=0; q5<rc; q5++){
                 for (int h6=0; h6<rc; h6++){
                     DecimalFormat df5 = new DecimalFormat("#.##");
             System.out.print(df5.format(cm[q5][h6])+"          ");
             }
             System.out.println();
         }
         return cm;
}

    }
```

```
class Node
        {
            int first;
            int second;
            int branch;
        }
```

```
class Tree
   {

    ArrayList<Node> sequence= new ArrayList<Node>();
    int originRow;
    int originCol;

    public float[][] makeRevised(Cm fc)
    {int nr=fc.cmrow;
     int nc=fc.cmcol;
     originRow=nr;
     originCol=nc;
     int to=nr+nc;
     float[][] revised=new float[to][to];
     for (int i=0; i<to; i++)
         for (int j=0; j<to; j++)
```

```
         {if (i<j)

                 {
             revised[i][j]=0;
          }
 else
                 {
             revised[i][j]=fc.cm[i][j];
             }
 }
//show the revised of matrix CM
         System.out.println();
         System.out.println("this is the revised of CM Matrix");
        for (int q6=0; q6<to; q6++){
                 for (int h7=0; h7<to; h7++){
                     DecimalFormat df6 = new DecimalFormat("#.##");
                     System.out.print(df6.format(revised[q6][h7])+"          ");
             }
             System.out.println();}
     return revised;

    }

    public void makeTree(float[][] revisedCm1)
    {
        int hr1=originRow+originCol;
        Cell[][] revisedCm= new Cell[hr1][hr1];
        for(int ss=0; ss<hr1; ss++)
            for(int yy=0; yy<hr1; yy++)
            {
                revisedCm[ss][yy]=new Cell();
                revisedCm[ss][yy].colNum=yy+1;
                revisedCm[ss][yy].rowNum=ss+1;
                revisedCm[ss][yy].extraRow=0;
                revisedCm[ss][yy].extraCol=0;
                revisedCm[ss][yy].contain=revisedCm1[ss][yy];
            }
        int counter=10000;
        int counter1=revisedCm.length;
        while (counter1 != 1)
        {
        Cell newnode=findMax(revisedCm);
        Node n=new Node();
        if(newnode.extraRow==0)
        {
            n.first = newnode.rowNum;}
        else { n.first=newnode.extraRow;}
        if(newnode.extraCol==0)
        {
            n.second = newnode.colNum;}
        else  {n.second=newnode.extraCol;}
        n.branch=counter;
        counter++;
        sequence.add(n);
        revisedCm[(newnode.rowNum)-1][(newnode.colNum)-1].contain=0;
        revisedCm[(newnode.colNum)-1][(newnode.rowNum)-1].contain=0;
        for(int jn=0; jn<revisedCm.length; jn++)
        {revisedCm[(newnode.colNum)-1][jn].contain=((revisedCm[(newnode.colNum)-
1][jn].contain)+(revisedCm[(newnode.rowNum)-1][jn].contain))/2;
         revisedCm[jn][(newnode.colNum)-1].contain=((revisedCm[jn][(newnode.colNum)-
1].contain)+(revisedCm[jn][(newnode.rowNum)-1].contain))/2;
        revisedCm[(newnode.colNum)-1][jn].extraRow=n.branch;
         revisedCm[jn][(newnode.colNum)-1].extraCol=n.branch;
        }

        for(int in=0; in<revisedCm.length; in++)
        {
            revisedCm[in][(newnode.rowNum) - 1].contain = 0;
            revisedCm[(newnode.rowNum)-1][in].contain=0;
        }
```

```
        for (int ir=0; ir<revisedCm.length; ir++){
         for (int jr=0; jr<revisedCm.length; jr++)
         {if (ir<jr)

                 {
             revisedCm[ir][jr].contain=0;

    }
             }
        }


        counter1--;
    }

        for(int ss1=0; ss1<hr1; ss1++)
            for(int yy1=0; yy1<hr1; yy1++)
            {
                revisedCm1[ss1][yy1]=revisedCm[ss1][yy1].contain;
            }
        }

    public Cell findMax(Cell[][] toFind)
    {  Cell choose=new Cell();
        float max=toFind[0][0].contain;
        for(int it=0; it<toFind.length; it++)
            for (int jt=0; jt<toFind.length; jt++)
               {
                if (toFind[it][jt].contain>max)
                {
                max = toFind[it][jt].contain;
                choose.colNum=toFind[it][jt].colNum;
                choose.rowNum=toFind[it][jt].rowNum;
                choose.extraRow=toFind[it][jt].extraRow;
                choose.extraCol=toFind[it][jt].extraCol;
                choose.contain=max;
                   }
            }
    return choose;
    }


     public ArrayList<Float> findLeafs(ArrayList<Node> seqq, int point)
    {
        int y5=0;
        int fi;
        int se;
        ArrayList<Float> leafs=new ArrayList<Float>();
        ArrayList<Float> nonLeafs=new ArrayList<Float>();
        if (point>9999){
        for (int x5=0; x5<seqq.size();x5++)
        {
            if (seqq.get(x5).branch==point)
            {
               y5=x5;
               break;
            }
        }

        fi=seqq.get(y5).first;
        se=seqq.get(y5).second;
        if (fi<9999)
        {
            float newLeaf = fi;
            leafs.add(newLeaf);
        }
else
        {
         float newNonLeaf=fi;
         nonLeafs.add(newNonLeaf);
}
```

```
        if (se<9999)
        {
            float newLeaf =se;
            leafs.add(newLeaf);
        }
        else
        {
         float newNonLeaf=se;
         nonLeafs.add(newNonLeaf);
 }

             while (!nonLeafs.isEmpty())
              {
                 for (int x9=0; x9<nonLeafs.size(); x9++)
                     {float point11=nonLeafs.get(x9);
                      int point1=(int)point11;
                        for(int y9=0; y9<seqq.size(); y9++)
                         {
            if (seqq.get(y9).branch==point1)
            {
                nonLeafs.remove(x9);
               if (seqq.get(y9).first<9999)
               {
                   float newLeaf=seqq.get(y9).first;
                   leafs.add(newLeaf);
               }
             else
               {
                   float newLeaf=seqq.get(y9).first;
                   nonLeafs.add(newLeaf);
                }
               if (seqq.get(y9).second<9999)
               {
                   float newLeaf=seqq.get(y9).second;
                   leafs.add(newLeaf);
               }
             else
               {
                   float newLeaf=seqq.get(y9).second;
                   nonLeafs.add(newLeaf);
                }

                  break;
            }

        }
              }

        }
        }
 else
        {
            float newLeaf2=point;
            leafs.add(newLeaf2);
 }
return leafs;
    }


     public int[][] findSequence (ArrayList<Node> se)
    {
         int irow=0;
         int jcol=0;
         int[][] seq;
         if (originRow>originCol)
         {seq=new int[2][originRow];}
         else
         { seq = new int[2][originCol];}
         for (int u = 0; u < se.size(); u++)
         {
             if ((se.get(u).first<= originCol)&&(se.get(u).first>0))
```

```
            {
                seq[1][jcol] = se.get(u).first;
                jcol++;
            }
            if (se.get(u).first> originCol && se.get(u).first<9999 &&
se.get(u).first>0 )
            {
                seq[0][irow]= se.get(u).first - originCol;
                irow++;
            }
            if (se.get(u).second<= originCol && se.get(u).second>0)
            {
                seq[1][jcol] = se.get(u).second;
                jcol++;
            }
            if (se.get(u).second> originCol && se.get(u).second<9999 &&
se.get(u).second>0 )
            {
                seq[0][irow]= se.get(u).second - originCol;
                irow++;
            }
            if (se.get(u).branch<= originCol && se.get(u).branch>0)
            {
                seq[1][jcol] = se.get(u).branch;
                jcol++;
            }
            if (se.get(u).branch> originCol && se.get(u).branch<9999 &&
se.get(u).branch>0 )
            {
                seq[0][irow]= se.get(u).branch - originCol;
                irow++;
            }

        }

        // show seq array

         for (int z=0; z<originRow; z++)
         {System.out.print(seq[0][z]+"    ");

         }
        System.out.println();
        for (int y=0; y<originCol; y++)
         {System.out.print(seq[1][y]+"    ");

         }

        return seq;

         }

    public float findDependency(int point1, int point2, float[][] revisedCm5)
   {
        float dep=0;
        ArrayList <Float> point1List=findLeafs(sequence, point1);
        ArrayList <Float> point2List=findLeafs(sequence, point2);
        for (int k=0; k<point1List.size(); k++)
            for(int k1=0; k1<point2List.size(); k1++)
            {
                float hy=point1List.get(k)-1;
                int hy1=(int)hy;
                float hx=point2List.get(k1)-1;
                int hx1=(int)hx;
                dep=dep+revisedCm5[hy1][hx1];

            }
        return dep;
    }

    public int findIndex(int m)
   {   int o=-1;
```

```
         for (int s=0; s<sequence.size(); s++)
             if (sequence.get(s).branch==m)
             {  o=s;
                 break;}
         return o;
     }

     public ArrayList<Node> reordering(float[][] revisedCm6)
    {
         ArrayList<Node> sequence1= new ArrayList<Node>();
         int count=1;
         int a=sequence.size();
         Node adding=new Node();
         adding.branch=sequence.get(a-1).branch;
         adding.first=sequence.get(a-1).first;
         adding.second=sequence.get(a-1).second;
         sequence1.add(adding);
         while(count!= sequence.size())
         {  ArrayList<Node> sequence2= new ArrayList<Node>();
             for(int b=0; b<sequence1.size();b++)
                 { if
((sequence1.get(b).first>9999)||(sequence1.get(b).second>9999)){
              if (sequence1.get(b).first>9999)
              {   int pr=findIndex(sequence1.get(b).first);
                  Node adding1=new Node();
                  adding1.branch=sequence.get(pr).branch;
                  adding1.first=sequence.get(pr).first;
                  adding1.second=sequence.get(pr).second;
                  sequence2.add(adding1);
                  count++;
              }
 else
              {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).first;
                  adding1.first=-1;
                  adding1.second=-1;
                  sequence2.add(adding1);

 }
              if (sequence1.get(b).second>9999)
              {   int pr=findIndex(sequence1.get(b).second);
                  Node adding1=new Node();
                  adding1.branch=sequence.get(pr).branch;
                  adding1.first=sequence.get(pr).first;
                  adding1.second=sequence.get(pr).second;
                  sequence2.add(adding1);
                  count++;
              }
              else
              {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).second;
                  adding1.first=-1;
                  adding1.second=-1;
                  sequence2.add(adding1);

 }
                     }

 else
           {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).branch;
                  adding1.first=sequence1.get(b).first;
                  adding1.second=sequence1.get(b).second;
                  sequence2.add(adding1);
                     }

 }
            if(sequence2.size()>2)
            { int co=sequence2.size()+100;
              while(co!=0){
              for (int u=1; u<(sequence2.size())-1;u++)
```

```
                {  float fdis;
                   float sdis;
                   fdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u).branch, revisedCm6);
                   sdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u+1).branch, revisedCm6);
                   if(sdis>fdis)
                   {
                       Node sub=new Node();
                  sub.branch=sequence2.get(u+1).branch;
                  sub.first=sequence2.get(u+1).first;
                  sub.second=sequence2.get(u+1).second;
                  sequence2.get(u+1).branch= sequence2.get(u).branch;
                  sequence2.get(u+1).first=sequence2.get(u).first;
                  sequence2.get(u+1).second=sequence2.get(u).second;
                  sequence2.get(u).branch=sub.branch;
                  sequence2.get(u).first=sub.first;
                  sequence2.get(u).second=sub.second;
                   }
                }
          co--;
              }
            }
            sequence1=sequence2;
          }

         // print sequence
         System.out.println();
         for (int kh = 0; kh < sequence1.size(); kh++)
         {
             System.out.println(sequence1.get(kh).first+"
"+sequence1.get(kh).second+"   "+sequence1.get(kh).branch);
         }
         return sequence1;
         }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println("enter the number of rows");
        Scanner getRow= new Scanner(System.in);
        int row2=getRow.nextInt();
        System.out.println("enter the number of columns");
        Scanner getCol= new Scanner(System.in);
        int col2=getCol.nextInt();
        Matrix matrix1=new Matrix(row2,col2);
        System.out.println();
        matrix1.showMatrix();
        Cm cm1=new Cm(matrix1);
        cm1.tcmrc();
        float[][] finalCm=cm1.makeCm();
        Tree mtree= new Tree();
        finalCm=mtree.makeRevised(cm1);
        mtree.makeTree(finalCm);
        mtree.printSequence();
        ArrayList<Node> seque=mtree.reordering(finalCm);
        int[][] seq2=mtree.findSequence(seque);
        System.out.println();
        matrix1.rearrange(seq2);
    }
}
```

Figure B3. The Modified Code of the Case Study OOP Based on Solution 2

```
package clustering3;
import java.text.DecimalFormat;
import java.util.*;
class Matrix
{   int row;
    int col;
    int[][] mat;

         public Matrix(int row1, int col1){
         row=row1;
         col=col1;
         mat= new int[row][col];
                  for (int i=0; i<row; i++){
                     int p=i+1;
                     System.out.println("enter the entities of row number"+ p );
                  for (int j=0; j<col; j++){
                   Scanner getEntity= new Scanner(System.in);
                   int next;
                   next= getEntity.nextInt();
                     mat[i][j]=next;

                  }
                                                 }
                       }
         public int[][] rearrange(int[][] sequ1)
        {
    int[][] mat1=new int[row][col];
    for(int d=0; d<row; d++){
                     for (int f=0; f<col; f++){
                         int h=(sequ1[0][d])-1;
                    mat1[d][f]=mat[h][f];
        }
             }

    int[][] mat2=new int[row][col];
    for(int d1=0; d1<col; d1++){
                     for (int f1=0; f1<row; f1++){
                         int h8=(sequ1[1][d1])-1;
            mat2[f1][d1]=mat1[f1][h8];
        }
             }


    int [][] mat3=new int[row+1][col+1];
    mat3[0][0]=0;
    for (int px=0; px<row; px++)
        mat3[px+1][0]=sequ1[0][px];
    for (int py=0; py<col; py++)
        mat3[0][py+1]=sequ1[1][py];
    for (int xy=0; xy<row; xy++)
        for (int xy1=0; xy1<col; xy1++)
        {
            mat3[xy+1][xy1+1]=mat2[xy][xy1];
        }

    for(int d5=0; d5<row+1; d5++){
                     for (int f5=0; f5<col+1; f5++){
                  System.out.print(mat3[d5][f5]+" ");
        }
                     System.out.println();
                     }

    return mat2;
}

         public void showMatrix()
    {    System.out.println("this is the original matrix");
       for (int i=0; i<row; i++){
             for (int j=0; j<col; j++){
                 System.out.print(mat[i][j]+"  ");
             }
```

```
            System.out.println();
        }
}
}
```

```
class Cm
{
    float[][] cm;
    float[][] cmr;
    float[][] cmc;
    float[][] cmrc;
    float[][] trans;
    int cmrow;
    int cmcol;

    public Cm(Matrix matrixIn)
    {
        cmrow= matrixIn.row;
        cmcol= matrixIn.col;
        cmr=new float[matrixIn.row][matrixIn.row];
        for (int m=0; m<matrixIn.row; m++)
            for (int n=0; n<matrixIn.row; n++)
            {
                if (m==n)
                {
                    cmr[m][n]=0;
                }
                else
                {
                    int max=0;
                    int min=0;
                    for (int k=0; k<matrixIn.col; k++)
                    { if ((matrixIn.mat[m][k]==1) || (matrixIn.mat[n][k]==1))
                            {
                                max++;}
                      if ((matrixIn.mat[m][k]==1) && (matrixIn.mat[n][k]==1))
                      {
                          min++;}
                    }
                    float h=(float)min/max;
                    cmr[m][n]=h;
                }
                }
        //show the matrix CMr
         System.out.println();
         System.out.println("this is the CMr Matrix");
        for (int q=0; q<matrixIn.row; q++){
                  for (int h=0; h<matrixIn.row; h++){
                   DecimalFormat df1 = new DecimalFormat("#.##");
                   System.out.print(df1.format(cmr[q][h])+"      ");
             }
              System.out.println();
         }

        cmc=new float[matrixIn.col][matrixIn.col];
        for (int m1=0; m1<matrixIn.col; m1++)
            for (int n1=0; n1<matrixIn.col; n1++)
            {
                if (m1==n1)
                {
                    cmc[m1][n1]=0;
                }
                else
                {
                    int max1=0;
                    int min1=0;
                    for (int k1=0; k1<matrixIn.row; k1++)
                    { if ((matrixIn.mat[k1][m1]==1) || (matrixIn.mat[k1][n1]==1))
                            {
                                max1++;}
                      if ((matrixIn.mat[k1][m1]==1) && (matrixIn.mat[k1][n1]==1))
```

```
                        {
                            min1++;}
                        }
                        float h1=(float)min1/max1;
                        cmc[m1][n1]=h1;
                    }
                    }
            // show the matrix CMc

             System.out.println();
             System.out.println("this is the CMc Matrix");
             for (int q1=0; q1<matrixIn.col; q1++){
                     for (int h2=0; h2<matrixIn.col; h2++){
                         DecimalFormat df2 = new DecimalFormat("#.##");
                      System.out.print(df2.format(cmc[q1][h2])+"         ");
                 }
                 System.out.println();
             }

            cmrc=new float[matrixIn.row][matrixIn.col];
            for (int m2=0; m2<matrixIn.row; m2++)
                for (int n2=0; n2<matrixIn.col; n2++)
                {

                        int sumr=0;
                        int sumc=0;
                        for (int k2=0; k2<matrixIn.col; k2++)
                        { if (matrixIn.mat[m2][k2]==1)
                                {
                                    sumr++;}}
                         for (int k3=0; k3<matrixIn.row; k3++)
                        { if (matrixIn.mat[k3][n2]==1 )
                                {
                                    sumc++;}}
                        float h3=matrixIn.mat[m2][n2];
                        cmrc[m2][n2]=(float)((2*h3)/(sumr+sumc));
                    }
            //show the matrix CMrc

             System.out.println();
             System.out.println("this is the CMrc Matrix");
            for (int q3=0; q3<matrixIn.row; q3++){
                     for (int h4=0; h4<matrixIn.col; h4++){
                         DecimalFormat df3 = new DecimalFormat("#.##");
                      System.out.print(df3.format(cmrc[q3][h4])+"          ");
                 }
                 System.out.println();
             }

                }
        public void tcmrc()
        { trans= new float[cmcol][cmrow];
          for (int i1=0; i1<cmrow; i1++){
                     for (int j1=0; j1<cmcol; j1++)
                     {trans[j1][i1]=cmrc[i1][j1];}}


                 //show the matrix TCMrc
             System.out.println();
             System.out.println("this is the transposed CMrc Matrix");
            for (int q4=0; q4<cmcol; q4++){
                     for (int h5=0; h5<cmrow; h5++){
                         DecimalFormat df4 = new DecimalFormat("#.##");
                      System.out.print(df4.format(trans[q4][h5])+"       ");

                 }
                 System.out.println();
             }

        }
    public float[][] makeCm()
```

```
    { int rc=cmrow+cmcol;
    cm= new float[rc][rc];
    for (int cr=0; cr<cmcol; cr++ )
    {
        for (int cc = 0; cc < cmcol; cc++){
            cm[cr][cc]=cmc[cr][cc];
        }}
    for (int tr=0;tr<cmcol; tr++  ){
        for(int tc=cmcol; tc<rc; tc++){
            cm[tr][tc]=trans[tr][tc-cmcol];
        }}
    for (int rcr=cmcol; rcr<rc; rcr++){
        for(int rcc=0; rcc<cmcol; rcc++){
            cm[rcr][rcc]=cmrc[rcr-cmcol][rcc];
        }}
    for (int rr=cmcol; rr<rc; rr++){
        for(int rc1=cmcol; rc1<rc; rc1++){
            cm[rr][rc1]=cmr[rr-cmcol][rc1-cmcol];
        }}

    //show the matrix CM
         System.out.println();
         System.out.println("this is the CM Matrix");
        for (int q5=0; q5<rc; q5++){
                  for (int h6=0; h6<rc; h6++){
                      DecimalFormat df5 = new DecimalFormat("#.##");
                      System.out.print(df5.format(cm[q5][h6])+"          ");
             }
              System.out.println();
         }
         return cm;
}
    }
```

```
class Node
        {
            int first;
            int second;
            int branch;
        }
```

```
class Tree
   {
    ArrayList<Node> sequence= new ArrayList<Node>();
    int originRow;
    int originCol;

    public float[][] makeRevised(Cm fc)
    {int nr=fc.cmrow;
     int nc=fc.cmcol;
     originRow=nr;
     originCol=nc;
     int to=nr+nc;
     float[][] revised=new float[to][to];
     for (int i=0; i<to; i++)
         for (int j=0; j<to; j++)
         {if (i<j)

                 {
            revised[i][j]=0;
          }
 else
                 {
            revised[i][j]=fc.cm[i][j];
            }
 }
//show the revised of matrix CM
         System.out.println();
         System.out.println("this is the revised of CM Matrix");
        for (int q6=0; q6<to; q6++){
```

```
                for (int h7=0; h7<to; h7++){
                    DecimalFormat df6 = new DecimalFormat("#.##");
                    System.out.print(df6.format(revised[q6][h7])+"          ");
            }
            System.out.println();}
     return revised;

    }

      public void makeTree(float[][] revisedCm)
    {
        ArrayList<Integer> helpRow= new ArrayList<Integer>();
        ArrayList<Integer> helpCol= new ArrayList<Integer>();
        for (int oi=0; oi<revisedCm.length; oi++)
        {
            helpRow.add(0);
            helpCol.add(0);

        }

        int counter=10000;
        int counter1=revisedCm.length;
        while (counter1 != 1)
        {
        int[][] newnode=new int[1][2];
        float max=revisedCm[0][0];
        for(int it=0; it<revisedCm.length; it++)
            for (int jt=0; jt<revisedCm.length; jt++)
               {
                if (revisedCm[it][jt]>max)
                {
                max = revisedCm[it][jt];
                newnode[0][0]=it;
                newnode[0][1]=jt;
                   }
            }
        Node n=new Node();
        if(helpRow.get(newnode[0][0])==0)
        {
            n.first = newnode[0][0]+1;}
        else { n.first=helpRow.get(newnode[0][0]);}
        if(helpCol.get(newnode[0][1])==0)
        {
            n.second = newnode[0][1]+1;}
        else  {n.second=helpCol.get(newnode[0][1]);}
        n.branch=counter;
        counter++;
        sequence.add(n);
        helpRow.set(newnode[0][1],n.branch);
        helpCol.set(newnode[0][1],n.branch);
        helpRow.remove(newnode[0][0]);
        helpCol.remove(newnode[0][0]);
        int kk=helpCol.size();
        float[][] helpRevised=new float[kk][kk];
        for(int yy=0; yy<kk+1; yy++)
            for(int xx=0; xx<kk+1; xx++)
            {
                if (xx<newnode[0][0] && yy<newnode[0][0])
                     helpRevised[yy][xx]=revisedCm[yy][xx];
                if (yy<newnode[0][0] && xx>newnode[0][0])
                    helpRevised[yy][xx-1]=revisedCm[yy][xx];
                if (yy>newnode[0][0] && xx<newnode[0][0])
                    helpRevised[yy-1][xx]=revisedCm[yy][xx];
                if (yy>newnode[0][0] && xx>newnode[0][0])
                    helpRevised[yy-1][xx-1]=revisedCm[yy][xx];

            }
        revisedCm=helpRevised;


        for (int ir=0; ir<revisedCm.length; ir++){
```

```
       for (int jr=0; jr<revisedCm.length; jr++)
         {if (ir<jr)

               {
            revisedCm[ir][jr]=0;

   }
          }
        }
        counter1--;
   }

        }


public ArrayList<Float> findLeafs(ArrayList<Node> seqq, int point)
   {
        int y5=0;
        int fi;
        int se;
        ArrayList<Float> leafs=new ArrayList<Float>();
        ArrayList<Float> nonLeafs=new ArrayList<Float>();
        if (point>9999){
        for (int x5=0; x5<seqq.size();x5++)
        {
            if (seqq.get(x5).branch==point)
            {
               y5=x5;
               break;
            }
        }

        fi=seqq.get(y5).first;
        se=seqq.get(y5).second;
        if (fi<9999)
        {
            float newLeaf = fi;
            leafs.add(newLeaf);
        }
else
        {
         float newNonLeaf=fi;
         nonLeafs.add(newNonLeaf);
}
        if (se<9999)
        {
            float newLeaf =se;
            leafs.add(newLeaf);
        }
        else
        {
         float newNonLeaf=se;
         nonLeafs.add(newNonLeaf);
}

             while (!nonLeafs.isEmpty())
              {
                 for (int x9=0; x9<nonLeafs.size(); x9++)
                     {float point11=nonLeafs.get(x9);
                      int point1=(int)point11;
                        for(int y9=0; y9<seqq.size(); y9++)
                          {
            if (seqq.get(y9).branch==point1)
            {
                nonLeafs.remove(x9);
               if (seqq.get(y9).first<9999)
               {
                   float newLeaf=seqq.get(y9).first;
                   leafs.add(newLeaf);
               }
             else
```

```
                {
                    float newLeaf=seqq.get(y9).first;
                    nonLeafs.add(newLeaf);
                 }
                 if (seqq.get(y9).second<9999)
                {
                    float newLeaf=seqq.get(y9).second;
                    leafs.add(newLeaf);
                }
              else
                {
                    float newLeaf=seqq.get(y9).second;
                    nonLeafs.add(newLeaf);
                 }

                   break;
            }

        }
              }

        }
        }
 else
        {
            float newLeaf2=point;
            leafs.add(newLeaf2);
 }

return leafs;
    }


     public int[][] findSequence (ArrayList<Node> se)
    {
         int irow=0;
         int jcol=0;
         int[][] seq;
         if (originRow>originCol)
         {seq=new int[2][originRow];}
         else
         { seq = new int[2][originCol];}
         for (int u = 0; u < se.size(); u++)
         {
             if ((se.get(u).first<= originCol)&&(se.get(u).first>0))
             {
                 seq[1][jcol] = se.get(u).first;
                 jcol++;
             }
             if (se.get(u).first> originCol && se.get(u).first<9999 &&
se.get(u).first>0 )
             {
                 seq[0][irow]= se.get(u).first - originCol;
                 irow++;
             }
             if (se.get(u).second<= originCol && se.get(u).second>0)
             {
                 seq[1][jcol] = se.get(u).second;
                 jcol++;
             }
             if (se.get(u).second> originCol && se.get(u).second<9999 &&
se.get(u).second>0 )
             {
                 seq[0][irow]= se.get(u).second - originCol;
                 irow++;
             }
             if (se.get(u).branch<= originCol && se.get(u).branch>0)
             {
                 seq[1][jcol] = se.get(u).branch;
                 jcol++;
             }
```

```
            if (se.get(u).branch> originCol && se.get(u).branch<9999 &&
se.get(u).branch>0 )
            {
                seq[0][irow]= se.get(u).branch - originCol;
                irow++;
            }

        }

        // show seq array

         for (int z=0; z<originRow; z++)
         {System.out.print(seq[0][z]+"    ");

         }
        System.out.println();
        for (int y=0; y<originCol; y++)
         {System.out.print(seq[1][y]+"    ");

         }

        return seq;

         }
     public float findDependency(int point1, int point2, float[][] revisedCm5)
    {
         float dep=0;
         ArrayList <Float> point1List=findLeafs(sequence, point1);
         ArrayList <Float> point2List=findLeafs(sequence, point2);
         for (int k=0; k<point1List.size(); k++)
             for(int k1=0; k1<point2List.size(); k1++)
             {
                 float hy=point1List.get(k)-1;
                 int hy1=(int)hy;
                 float hx=point2List.get(k1)-1;
                 int hx1=(int)hx;
                 dep=dep+revisedCm5[hy1][hx1];

             }
         return dep;
     }

     public int findIndex(int m)
    {   int o=-1;
         for (int s=0; s<sequence.size(); s++)
             if (sequence.get(s).branch==m)
             {  o=s;
                 break;}
         return o;
     }

     public ArrayList<Node> reordering(float[][] revisedCm6)
    {
         ArrayList<Node> sequence1= new ArrayList<Node>();
         int count=1;
         int a=sequence.size();
         Node adding=new Node();
         adding.branch=sequence.get(a-1).branch;
         adding.first=sequence.get(a-1).first;
         adding.second=sequence.get(a-1).second;
         sequence1.add(adding);
         while(count!= sequence.size())
         {  ArrayList<Node> sequence2= new ArrayList<Node>();
             for(int b=0; b<sequence1.size();b++)
                 { if
((sequence1.get(b).first>9999)||(sequence1.get(b).second>9999)){
              if (sequence1.get(b).first>9999)
              {   int pr=findIndex(sequence1.get(b).first);
                  Node adding1=new Node();
                  adding1.branch=sequence.get(pr).branch;
                  adding1.first=sequence.get(pr).first;
```

```
                    adding1.second=sequence.get(pr).second;
                    sequence2.add(adding1);
                    count++;
                }
 else
                {Node adding1=new Node();
                    adding1.branch=sequence1.get(b).first;
                    adding1.first=-1;
                    adding1.second=-1;
                    sequence2.add(adding1);

 }
                if (sequence1.get(b).second>9999)
                {   int pr=findIndex(sequence1.get(b).second);
                    Node adding1=new Node();
                    adding1.branch=sequence.get(pr).branch;
                    adding1.first=sequence.get(pr).first;
                    adding1.second=sequence.get(pr).second;
                    sequence2.add(adding1);
                    count++;
                }
                else
                {Node adding1=new Node();
                    adding1.branch=sequence1.get(b).second;
                    adding1.first=-1;
                    adding1.second=-1;
                    sequence2.add(adding1);

 }
                       }

 else
           {Node adding1=new Node();
                  adding1.branch=sequence1.get(b).branch;
                  adding1.first=sequence1.get(b).first;
                  adding1.second=sequence1.get(b).second;
                  sequence2.add(adding1);
                     }

 }
            if(sequence2.size()>2)
            { int co=sequence2.size()+100;
              while(co!=0){
              for (int u=1; u<(sequence2.size())-1;u++)
                {  float fdis;
                   float sdis;
                   fdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u).branch, revisedCm6);
                    sdis=findDependency(sequence2.get(u-1).branch,
sequence2.get(u+1).branch, revisedCm6);
                    if(sdis>fdis)
                    {
                        Node sub=new Node();
                   sub.branch=sequence2.get(u+1).branch;
                   sub.first=sequence2.get(u+1).first;
                   sub.second=sequence2.get(u+1).second;
                   sequence2.get(u+1).branch= sequence2.get(u).branch;
                   sequence2.get(u+1).first=sequence2.get(u).first;
                   sequence2.get(u+1).second=sequence2.get(u).second;
                   sequence2.get(u).branch=sub.branch;
                   sequence2.get(u).first=sub.first;
                   sequence2.get(u).second=sub.second;
                    }
                }
          co--;
              }
            }
            sequence1=sequence2;
           }

         // print sequence
```

```
         System.out.println();
         for (int kh = 0; kh < sequence1.size(); kh++)
         {
             System.out.println(sequence1.get(kh).first+"
"+sequence1.get(kh).second+"   "+sequence1.get(kh).branch);
         }
         return sequence1;
         }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println("enter the number of rows");
        Scanner getRow= new Scanner(System.in);
        int row2=getRow.nextInt();
        System.out.println("enter the number of columns");
        Scanner getCol= new Scanner(System.in);
        int col2=getCol.nextInt();
        Matrix matrix1=new Matrix(row2,col2);
        System.out.println();
        matrix1.showMatrix();
        Cm cm1=new Cm(matrix1);
        cm1.tcmrc();
        float[][] finalCm=cm1.makeCm();
        Tree mtree= new Tree();
        finalCm=mtree.makeRevised(cm1);
        mtree.makeTree(finalCm);
        mtree.printSequence();
        ArrayList<Node> seque=mtree.reordering(finalCm);
        int[][] seq2=mtree.findSequence(seque);
        System.out.println();
        matrix1.rearrange(seq2);
    }

}
```

Figure B4. The Modified Code of the Case Study OOP Based on Solution 3

Label	Name	Description
1	rowNum	a variable (field) of type `int` that is defined in `Cell` class
2	colNum	a variable (field) of type `int` that is defined in `Cell` class
3	contain	a variable (field) of type `float` that is defined in `Cell` class
4	extraRow	a variable (field) of type `int` that is defined in `Cell` class
5	extraCol	a variable (field) of type `int` that is defined in `Cell` class
6	row	a variable (field) of type `int` that is defined in `Matrix` class
7	col	a variable (field) of type `int` that is defined in `Matrix` class
8	mat	a variable (object) of type `Cell 2D array` that is defined in `Matrix` class
9	Cell	a class
10	Matrix	a class
11	Matrix	a method (constructor) that is defined in `Matrix` class
12	row1	a variable (field) of type `int` that is defined in `Matrix` method
13	col1	a variable (field) of type `int` that is defined in `Matrix` method
14	mat2	a variable (object) of type `Cell 2D array` that is defined in `rearrange` method
15	rearrange	a method that is defined in `Matrix` class
16	mat1	a variable (object) of type `Cell 2D array` that is defined in `rearrange` method
17	sequ1	a variable (field) of type `int 2D array` that is defined in `rearrange` method
18	showMatrix	a method that is defined in `Matrix` class
19	cm	a variable (object) of type `Cell 2D array` that is defined in `Cm` class
20	cmr	a variable (object) of type `Cell 2D array` that is defined in `Cm` class
21	cmc	a variable (object) of type `Cell 2D array` that is defined in `Cm` class
22	cmrc	a variable (object) of type `Cell 2D array` that is defined in `Cm` class
23	trans	a variable (object) of type `Cell 2D array` that is defined in `Cm` class
24	cmrow	a variable (field) of type `int` that is defined in `Cm` class
25	cmcol	a variable (field) of type `int` that is defined in `Cm` class
26	Cm	a method (constructor) that is defined in `Cm` class
27	max	a variable (field) of type `int` that is defined in `Cm` method
28	min	a variable (field) of type `int` that is defined in `Cm` method
29	h	a variable (field) of type `float` that is defined in `Cm` method
30	max1	a variable (field) of type `int` that is defined in `Cm` method
31	min1	a variable (field) of type `int` that is defined in `Cm` method
32	h1	a variable (field) of type `float` that is defined in `Cm` method
33	sumr	a variable (field) of type `int` that is defined in `Cm` method
34	sumc	a variable (field) of type `int` that is defined in `Cm` method
35	h3	a variable (field) of type `float` that is defined in `Cm` method
36	Cm	a class
37	tcmrc	a method that is defined in `Cm` class
38	Node	a class
39	makeCm	a method that is defined in `Cm` class
40	rc	a variable (field) of type `int` that is defined in `makeCm` method
41	branch	a variable (field) of type `int` that is defined in `Node` class
42	Tree	a class

43	sequence	a variable (object) of type `Node ArrayList` that is defined in `Tree` class
44	originRow	a variable (field) of type `int` that is defined in `Tree` class
45	originCol	a variable (field) of type `int` that is defined in `Tree` class
46	makeRevised	a method that is defined in `Tree` class
47	fc	a variable (object) of type `Cm` that is defined in `makeRevised` method
48	nr	a variable (field) of type `int` that is defined in `makeRevised` method
49	nc	a variable (field) of type `int` that is defined in `makeRevised` method
50	first	a variable (field) of type `int` that is defined in `Node` class
51	second	a variable (field) of type `int` that is defined in `Node` class
52	makeTree	a method that is defined in `Tree` class
53	revisedCm	a variable (object) of type `Cell 2D array` that is defined in `makeTree` method
54	counter	a variable (field) of type `int` that is defined in `makeTree` method
55	counter1	a variable (field) of type `int` that is defined in `makeTree` method
56	newnode	a variable (object) of type `Cell` that is defined in `makeTree` method
57	findMax	a method that is defined in `Tree` class
58	findLeafs	a method that is defined in `Tree` class
59	seqq	a variable (object) of type `Node ArrayList` that is defined in `findLeafs` method
60	point	a variable (field) of type `int` that is defined in `findLeafs` method
61	leafs	a variable (object) of type `Cell ArrayList` that is defined in `findLeafs` method
62	nonLeafs	a variable (object) of type `Cell ArrayList` that is defined in `findLeafs` method
63	toFind	a variable (object) of type `Cell 2D array` that is defined in `findMax` method
64	findSequence	a method that is defined in `Tree` class
65	se	a variable (object) of type `Node ArrayList` that is defined in `findSequence` method
66	seq	a variable (field) of type `int 2D array` that is defined in `findSequence` method
67	findDependency	a method that is defined in `Tree` class
68	point1	a variable (field) of type `int` that is defined in `findDependency` method
69	point2	a variable (field) of type `int` that is defined in `findDependency` method
70	revisedCm1	a variable (object) of type `Cell 2D array` that is defined in `findDependency` method
71	point1List	a variable (object) of type `Cell ArrayList` that is defined in `findDependency` method
72	point2List	a variable (object) of type `Cell ArrayList` that is defined in `findDependency` method
73	dep	a variable (field) of type `float` that is defined in `findDependency` method
74	findIndex	a method that is defined in `Tree` class
75	m	a variable (field) of type `int` that is defined in `findIndex` method
76	o	a variable (field) of type `int` that is defined in `findIndex` method

77	reordering	a method that is defined in `Tree` class
78	revisedCm2	a variable (object) of type `Cell 2D array` that is defined in `reordering` method
79	sequence1	a variable (object) of type `Node ArrayList` that is defined in `reordering` method
80	adding	a variable (object) of type `Node` that is defined in `reordering` method
81	sequence2	a variable (object) of type `Node ArrayList` that is defined in `reordering` method
82	pr	a variable (field) of type `int` that is defined in `reordering` method
83	adding1	a variable (object) of type `Node` that is defined in `reordering` method
84	fdis	a variable (field) of type `float` that is defined in `reordering` method
85	sdis	a variable (field) of type `float` that is defined in `reordering` method
86	main	this method is the main execution part of the program
87	row2	a variable (field) of type `int` that is defined in `main` method
88	col2	a variable (field) of type `int` that is defined in `main` method
89	matrix1	a variable (object) of type `Matrix` that is defined in `main` method
90	cm1	a variable (object) of type `Cm` that is defined in `main` method
91	finalCm	a variable (object) of type `Cell 2D array` that is defined in `main` method
92	mtree	a variable (object) of type `Tree` that is defined in `main` method
93	seque	a variable (object) of type `Node ArrayList` that is defined in `main` method
94	seq2	a variable (field) of type `int 2D array` that is defined in `main` method

Table B1. List of 94 OOP Entities of the Case Study OOP

Appendix C: Trace of Target Entities Related to Solution 2

This Appendix demonstrates the trace of entities number 8, 16, 14, 18, 20, 21, 22, 23, 19, 46, 53, 61, 62, 70, 71, 72 and 78, from the case study OOP, up to three levels

Figure C1: Change Propagation from Target Entity 16

Figure C2: Change Propagation from Target Entity 14

Figure C3: Change Propagation from Target Entity 18

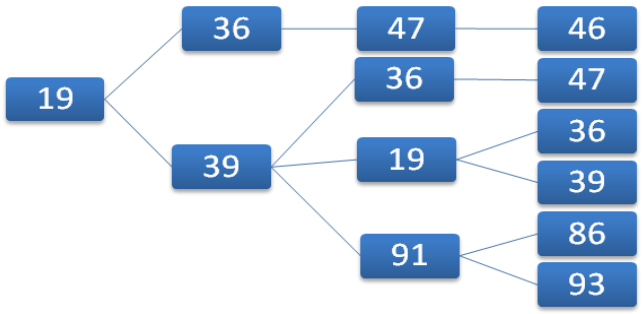

Figure C4: Change Propagation from Target Entity 19

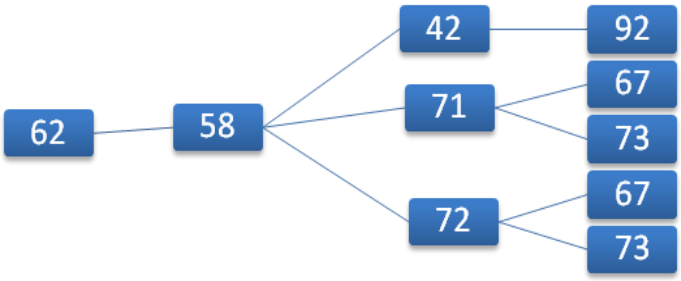

Figure C5: Change Propagation from Target Entity 62

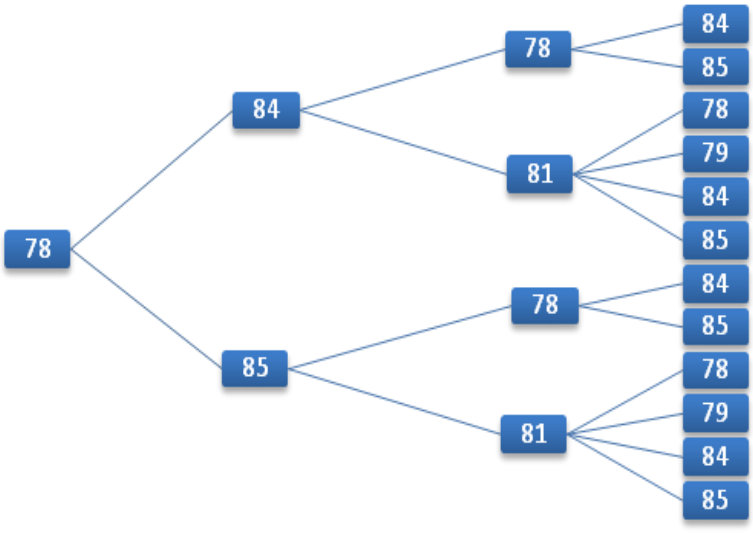

Figure C6: Change Propagation from Target Entity 78

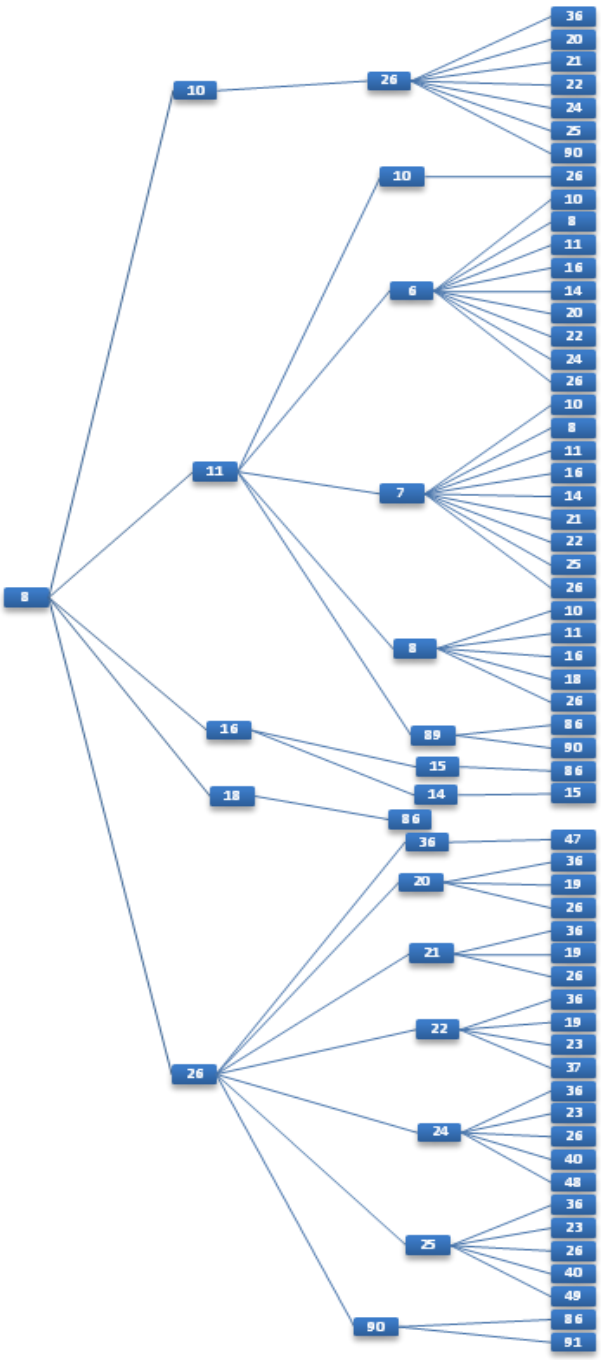

Figure C7: Change Propagation from Target Entity 8

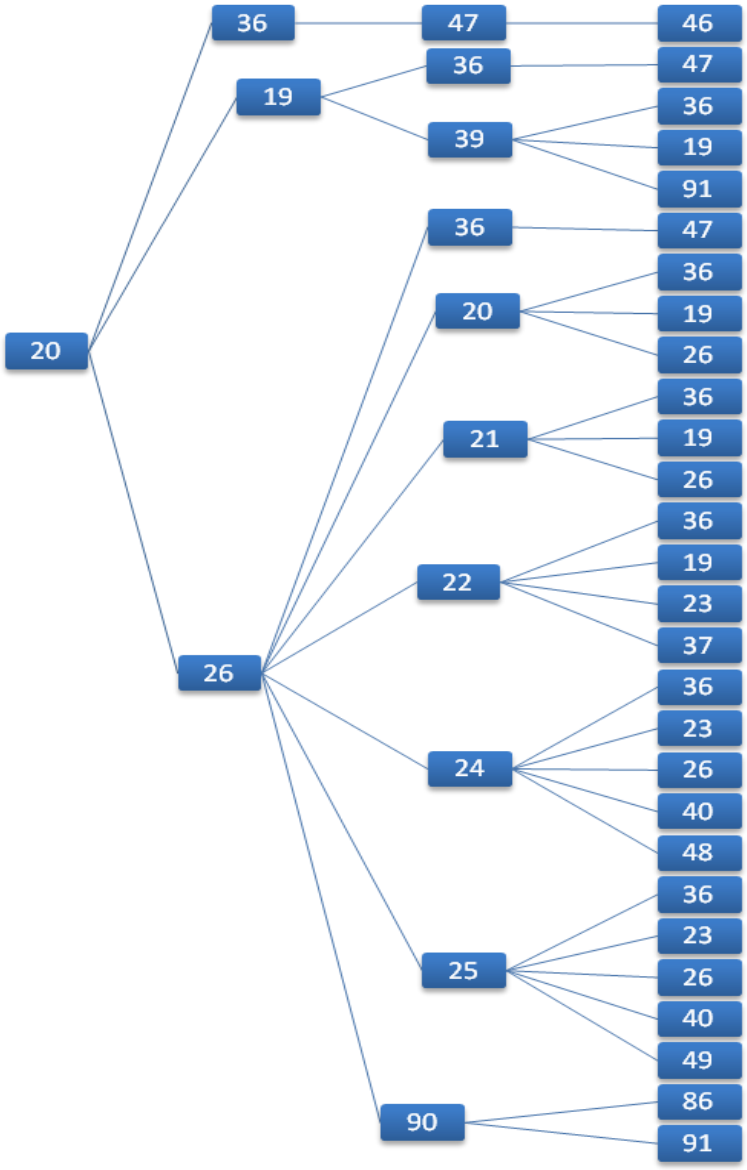

Figure C8: Change Propagation from Target Entity 20

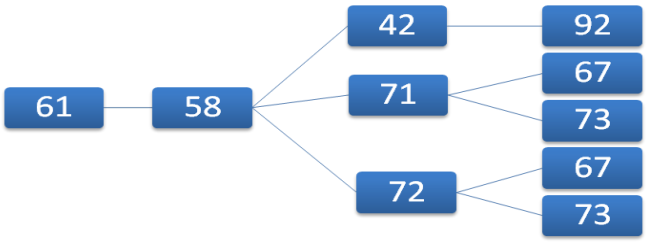

Figure C9: Change Propagation from Target Entity 61

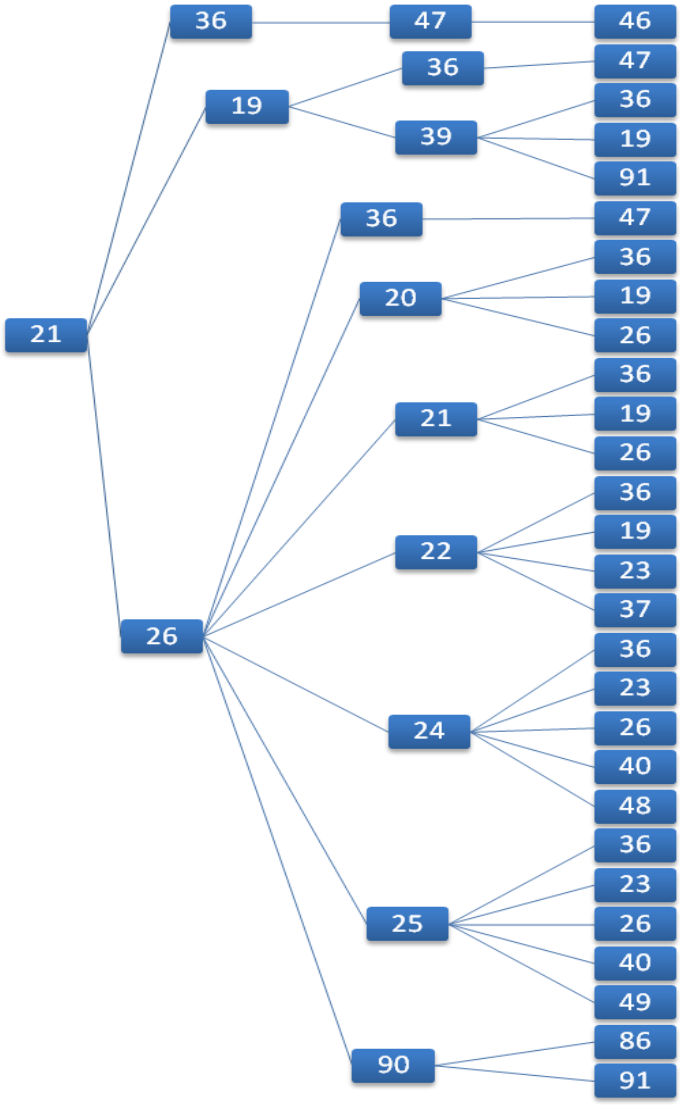

Figure C10: Change Propagation from Target Entity 21

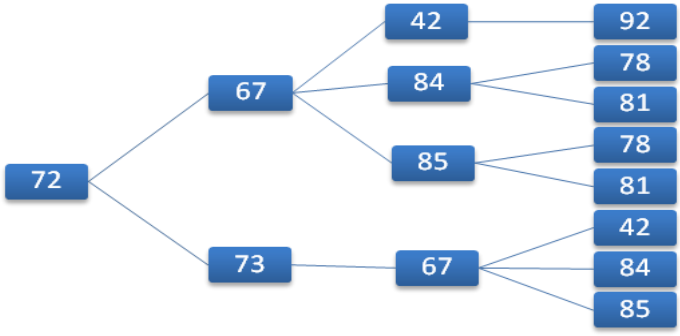

Figure C11: Change Propagation from Target Entity 72

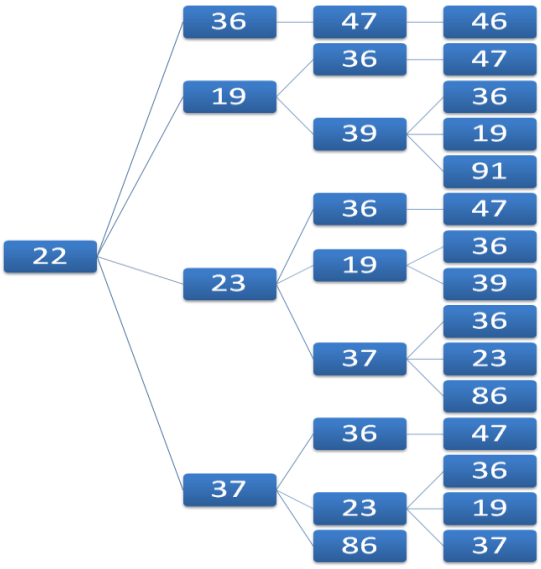

Figure C12: Change Propagation from Target Entity 22

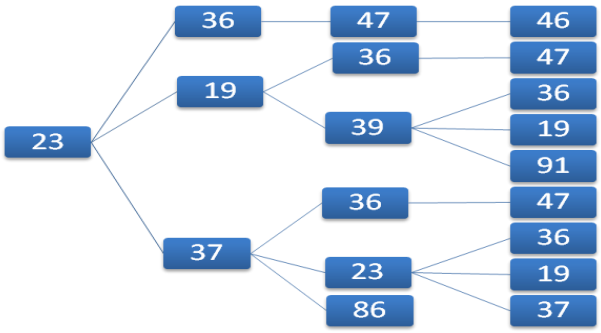

Figure C13: Change Propagation from Target Entity 23

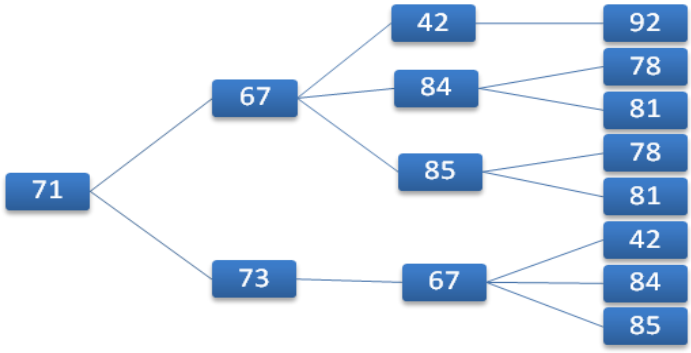

Figure C14: Change Propagation from Target Entity 71

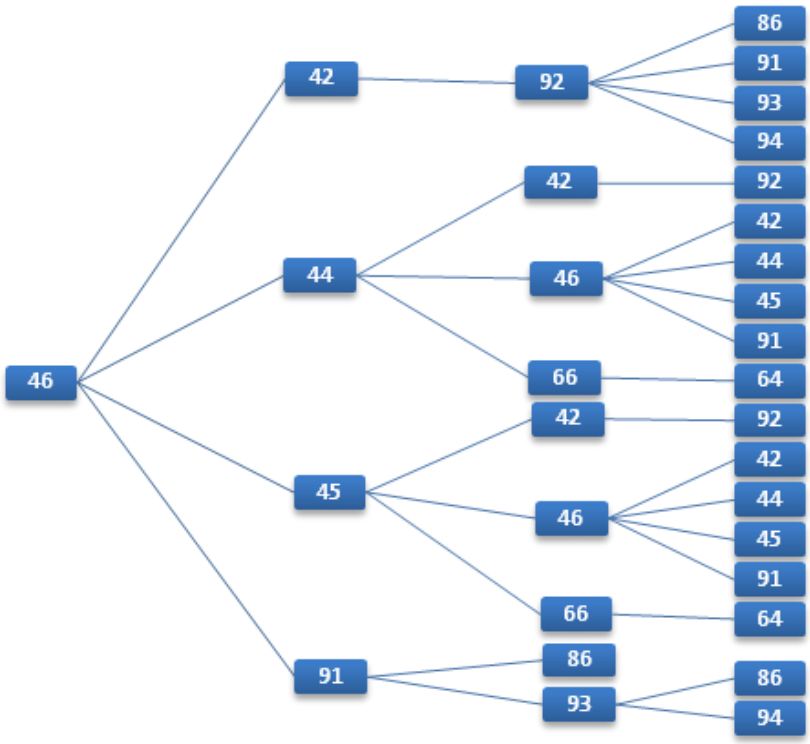

Figure C15: Change Propagation from Target Entity 46

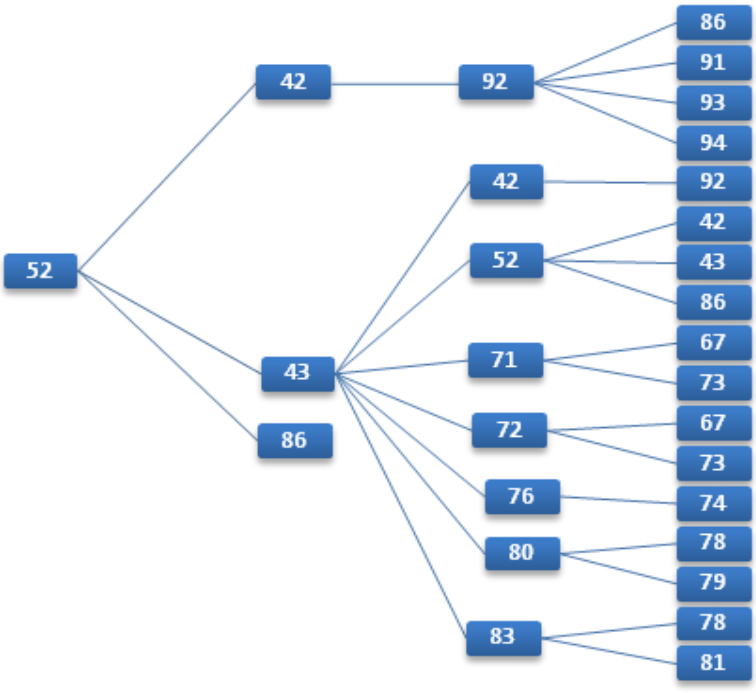

Figure C16: Change Propagation from Target Entity 52

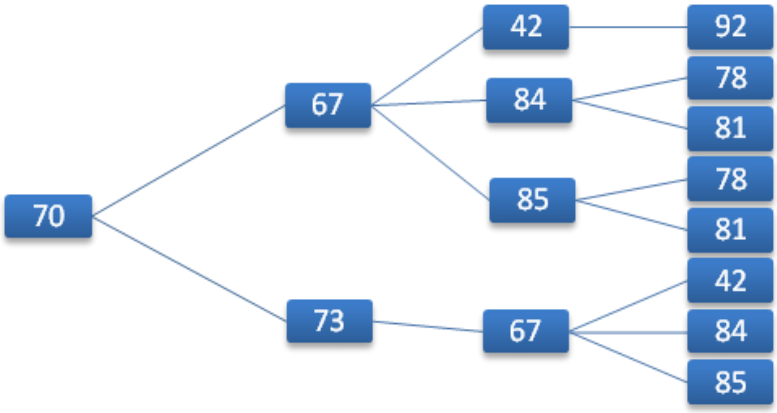

Figure C17: Change Propagation from Target Entity 70