

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Yiannis Kanellopoulos

September 2003

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of many people.

I am grateful to my family and Maria the Best, for the continuous support of all kinds they have provided me throughout this year.

I would also like to thank all my friends here in Manchester Imran, Angelos, Helen, Matina and Georgia for their help either in my academic or in my personal life.

I owe special thanks to my supervisor, Christos Tjortjis, whose guidance and speciality have been crucial for the completion of this project.

Abstract

This project investigates the application of data mining techniques such as clustering, in order to facilitate the understanding of programs. Program understanding is a very crucial step of the maintenance of a software system. This report examines the theory of Program Understanding and Knowledge Discovery in Databases, while some previous approaches on this problem are reviewed. The next step is the presentation of the Preprocessing Application, which consists of three parts: the Model of the Input Data, which is the specification of the program entities and their attributes, the Front End (G.U.I.) and the Back-End (Algorithm) that parses the source code and stores the data in the database. This presentation covers the Requirements gathering, Design and Implementation phases. What follows is the testing of the Preprocessing Application and the evaluation of its outcome. Two applications, `CAccessReport` and `CompDB`, are used as samples. Their actual structure is compared with the outcome of the clustering analysis of their respective input models. IBM's Intelligent Miner, is used for this purpose.

Table of Contents

ACKNOWLEDGEMENTS	II
ABSTRACT.....	III
TABLE OF CONTENTS	IV
TABLE OF FIGURES.....	VII
LIST OF ABBREVIATIONS	X
1. INTRODUCTION.....	1
1.1 PROBLEM DOMAIN DESCRIPTION	1
1.2 DISSERTATION OBJECTIVES	2
1.3 DISSERTATION STRUCTURE.....	3
2. BACKGROUND CHAPTER.....	5
2.1 SOFTWARE MAINTENANCE	5
2.1.1 <i>Categories of software maintenance changes</i>	6
2.2 PROGRAM UNDERSTANDING.....	7
2.2.1 <i>Program understanding models</i>	9
2.2.2 <i>Mental Models</i>	10
2.3 PROGRAM UNDERSTANDING STRATEGIES.....	11
2.3.1 <i>Top down model</i>	11
2.3.2 <i>Bottom up model</i>	13
2.3.3 <i>Opportunistic model</i>	14
2.4 KNOWLEDGE DISCOVERY IN DATABASES	16
2.5 KDD AND DATA MINING	17
2.5.1 <i>Steps of the KDD process</i>	18
2.6 DATA MINING TASKS.....	19
2.6.1 <i>Classification</i>	20
2.6.2 <i>Regression</i>	20
2.6.3 <i>Clustering</i>	20
2.7 KDD PROCESS AND PROGRAM UNDERSTANDING: PREVIOUS SOLUTIONS	22
2.7.1 USING DATA MINING TO ASSESS SOFTWARE RELIABILITY	22
2.7.2 A METHOD FOR LEGACY SYSTEMS MAINTENANCE BY MINING DATA EXTRACTED FROM SOURCE CODE	24
2.7.3 USING AUTOMATIC CLUSTERING TO PRODUCE HIGH-LEVEL SYSTEM ORGANISATIONS OF SOURCE CODE	26
2.7.4 A SOFTWARE EVALUATION MODEL USING COMPONENT ASSOCIATION VIEWS.....	29
3. REQUIREMENTS CHAPTER	31

3.1 INTRODUCTION	31
3.2 MODEL OF THE INPUT DATA	32
3.2.1 <i>Requirements of the Model of the Input Data</i>	32
3.3 PREPROCESSING APPLICATION	33
3.3.1 <i>Requirements of the Front-End Part (G.U.I.)</i>	33
3.3.2 <i>Requirements of the Back-End Part (Algorithm)</i>	35
3.4 REQUIREMENTS OF THE DATA MINING TOOL.....	35
4. DESIGN CHAPTER.....	36
4.1 INTRODUCTION	36
4.2 DESIGN OF THE MODEL OF INPUT DATA	37
4.3 DESIGN OF THE PREPROCESSING APPLICATION	41
4.3.1 <i>Functionalities of the Front-End (G.U.I.)</i>	41
4.3.2 <i>Description of the Back-End (Algorithm)</i>	42
4.3.3 <i>Functionalities of the Back-End (Algorithm)</i>	44
5. IMPLEMENTATION CHAPTER.....	49
5.1 IMPLEMENTATION PHASE AND TOOLS	49
5.2 IMPLEMENTATION OF THE MODEL OF DATA	50
5.3 IMPLEMENTATION OF THE PREPROCESSING APPLICATION.....	52
5.3.1 <i>Front-End (G.U.I.)</i>	52
5.3.2 <i>Back-End (Algorithm)</i>	55
6. TESTING CHAPTER	59
6.1 INTRODUCTION	59
6.2 TESTING OF THE PREPROCESSING APPLICATION.....	59
6.2.1 <i>Step 1: Preprocessing the “_Form.h”</i>	59
6.2.2 <i>Step 2: Insertion of data in Classes table</i>	61
6.2.3 <i>Step 3: Insertion of data in Member_Functions table</i>	61
6.2.4 <i>Step 4: Insertion of data in Function_Parameters table</i>	62
7. RESULTS – EVALUATION CHAPTER.....	64
7.1 INTRODUCTION	64
7.2 EVALUATION OF THE PREPROCESSING APPLICATION	65
7.2.1 <i>Description and General Characteristics of the CAccessReport system</i>	65
7.2.1.1 CAccessReport: Classes Analysis.....	65
7.2.1.2 CAccessReport: Member Functions Analysis.....	68
7.2.1.3 CAccessReport: Analysis of the parameters of member functions.....	74
7.2.1.4 CAccessReport: Conclusions	78
7.2.2 <i>Description and General Characteristics of the CompDB system</i>	79
7.2.2.1 CompDB: Classes Analysis	80
7.2.2.2 CompDB: Member Data Analysis	84

7.2.2.3 CompDB: Member Functions Analysis	89
7.2.2.4 CompDB: Analysis of thr Parameters of Member Functions	94
7.2.2.5 CompDB: Conclusions	97
8. CONCLUSIONS – FUTURE WORK CHAPTER	99
8.1 INTRODUCTION	99
8.2 OVERVIEW	99
8.3 LESSONS LEARNED	100
8.4 CONCLUSIONS	101
8.4.1 <i>Conclusions for the methodology</i>	101
8.4.2 <i>Conclusions for the Preprocessing Application</i>	101
8.5 FUTURE WORK	103
REFERENCES	105
PAPERS	105
BOOKS	106
URLS	106
APPENDIX	108
SCREENSHOTS FROM THE ANALYSIS OF COMPDB APPLICATION	108
<i>CompDB: Screenshots from Classes Analysis</i>	108
<i>CompDB: Screenshots from Member Data Analysis</i>	111
<i>CompDB: Screenshots from Member Functions Analysis</i>	114
<i>CompDB: Screenshots from Parameters of Member Functions Analysis</i>	117

Table of Figures

FIGURE 2-1: WATERFALL MODEL OF A SOFTWARE SYSTEM LIFE CYCLE [TAKANG, GRUPP 1996].....	6
FIGURE 2-3: BOTTOM-UP COMPREHENSION PROCESS [TAKANG, GRUPP 1996]	13
FIGURE 2-5: A SIMPLE CLUSTERING OF A LOAN SET INTO 3 CLUSTERS [FAYYAD ET AL. 1996]	21
FIGURE 2-6: AUTOMATIC SOFTWARE MODULARISATION TECHNIQUE (ENVIRONMENT) [MANCORIDIS ET AL. 1998]	27
FIGURE 4-1: BASIC IDEA OF THE DATABASE STORE OF THE MODEL OF THE INPUT DATA.....	37
FIGURE 4-2: OPEN FILE FUNCTION	42
FIGURE 4-3: PROVIDE FEEDBACK TO THE USER FUNCTION	42
FIGURE 4-4: PREPROCESSING ALGORITHM	43
FIGURE 4-5: CLASS DIAGRAM OF THE BACK-END (ALGORITHM) OF THE PREPROCESSING APPLICATION	44
FIGURE 4-6: FILE CHOOSER OF THE PREPROCESSING APPLICATION WITH THE C++ FILES FILTER IMPLEMENTED.....	48
FIGURE 5-1: STRUCTURE OF THE SYSTEM.....	49
FIGURE 5-2: PREPROCESSINGDB DATABASE SCHEMA	50
FIGURE 5-3: “C++ SOURCE CODE PREPROCESSING APPLICATION” WINDOW	53
FIGURE 5-4: “FILE PREPROCESSING” FORM.....	54
FIGURE 5-5: SUCCESSFUL EXECUTION OF THE EXTRACTCLASSDATA FUNCTION	56
FIGURE 5-6: SUCCESSFUL EXECUTION OF THE FUNCTIONS EXTRACTMEMBERDATA AND MEMBERDATACATEGORY.	57
FIGURE 5-7: SUCCESSFUL EXECUTION OF THE FUNCTIONS EXTRACTMEMBERFUNCTIONSDATA AND MEMBERFUNCTIONCATEGORY.	58
FIGURE 6-1: FEEDBACK PROVIDED TO THE USER ABOUT THE PREPROCESSING OF THE _Form.h FILE	60
FIGURE 6-2: INSERTION OF DATA IN THE CLASSES TABLE	61
FIGURE 6-3: INSERTION OF DATA IN THE MEMBER_FUNCTIONS TABLE (IN THE RED CIRCLE THERE IS THE NUMBER OF THE NEW RECORDS)	62
FIGURE 6-4: INSERTION OF DATA IN THE FUNCTION_PARAMETERS TABLE (IN THE RED CIRCLE THERE IS THE NUMBER OF THE NEW RECORDS)	63
FIGURE 7-1: CACCESSREPORT: FIRST CLUSTER OF THE CLASSES TABLE (SMALL SIZE CLASSES).	66
FIGURE 7-2: CACCESSREPORT: SECOND CLUSTER OF THE CLASSES TABLE (MEDIUM SIZE CLASSES).	67
FIGURE 7-3: CACCESSREPORT: THIRD CLUSTER OF THE CLASSES TABLE (LARGE SIZE CLASSES).	67
FIGURE 7-4: CACCESSREPORT: FIRST CLUSTER OF THE MEMBER_FUNCTIONS TABLE. PUBLIC FUNCTIONS WITH PARAMETERS	68
FIGURE 7-5: CACCESSREPORT: FIRST CLUSTER OF THE MEMBER_FUNCTIONS TABLE. PUBLIC FUNCTIONS THAT RETURN VOID OR NULL	69
FIGURE 7-6: CACCESSREPORT: SECOND CLUSTER OF THE MEMBER_FUNCTIONS TABLE. PUBLIC FUNCTIONS WITH NO PARAMETERS	70
FIGURE 7-7: CACCESSREPORT: SECOND CLUSTER OF THE MEMBER_FUNCTIONS TABLE. PUBLIC FUNCTIONS THAT RETURN CSTRING	71

FIGURE7-8: CACCESSREPORT: THIRD CLUSTER OF THE MEMBER_FUNCTIONS TABLE. PUBLIC FUNCTIONS WITH PARAMETERS AND NO PARAMETERS	72
FIGURE7-9: CACCESSREPORT: THIRD CLUSTER OF THE MEMBER_FUNCTIONS TABLE. PUBLIC FUNCTIONS THAT RETURN VARIANT AND LPDISPATCH	73
FIGURE7-10: CACCESSREPORT: FIRST CLUSTER OF THE FUNCTION_PARAMETERS TABLE. PARAMETERS BY VALUE.....	74
FIGURE7-11: CACCESSREPORT: FIRST CLUSTER OF THE FUNCTION_PARAMETERS TABLE. PARAMETERS OF TYPE LPCSTR, LPDISPATCH AND VARIANT	75
FIGURE7-12: CACCESSREPORT: SECOND CLUSTER OF THE FUNCTION_PARAMETERS TABLE. PARAMETERS BY VALUE	76
FIGURE7-13: CACCESSREPORT: SECOND CLUSTER OF THE FUNCTION_PARAMETERS TABLE. PARAMETERS OF TYPE BOOL, SHORT AND LONG	76
FIGURE7-14: CACCESSREPORT: THIRD CLUSTER OF THE FUNCTION_PARAMETERS TABLE. PARAMETERS BY REFERENCE.....	77
FIGURE7-15: CACCESSREPORT: SECOND CLUSTER OF THE FUNCTION_PARAMETERS TABLE. PARAMETERS OF TYPE VARIANT.....	78
FIGURE7-16: COMPDB: FIRST CLUSTER CLASSES TABLE. INFORMATION ABOUT THE SUPER CLASSES....	81
FIGURE7-17: COMPDB: SECOND CLUSTER OF CLASSES TABLE. INFORMATION ABOUT THE SUPER CLASSES.....	82
FIGURE7-18: COMPDB: THIRD CLUSTER OF CLASSES TABLE. INFORMATION THAT DESCRIBES THE SUPER CLASSES.....	83
FIGURE7-19: COMPDB: FIRST CLUSTER OF MEMBER_DATA TABLE. DISTRIBUTION OF THE MEMBER DATA AMONG THE CLASSES OF THE COMPDB APPLICATION.....	85
FIGURE7-20: COMPDB: FIRST CLUSTER OF MEMBE_DATA TABLE. TYPES OF MEMBER DATA.....	85
FIGURE7-21: DISTRIBUTION OF THE MEMBER DATA OF THE SECOND CLUSTER AMONG THE CLASSES OF THE COMPDB APPLICATION.....	86
FIGURE7-22: COMPDB: SECOND CLUSTER OF MEMBER_DATA TABLE. TYPES OF MEMBER DATA.....	87
FIGURE7-23: COMPDB: THIRD CLUSTER OF MEMBER_DATA . DISTRIBUTION OF THE MEMBER DATA AMONG THE CLASSES OF THE COMPDB APPLICATION.....	88
FIGURE7-24: COMPDB: THIRD CLUSTER OF MEMBER_DATA TABLE. TYPES OF MEMBER DATA	89
FIGURE7-25: COMPDB: FIRST CLUSTER OF MEMBER_FUNCTIONS TABLE. RETURN TYPES OF THE MEMBER FUNCTIONS	90
FIGURE7-26: COMPDB: FIRST CLUSTER OF MEMBER_FUNCTIONS TABLE. DISTRIBUTION OF THE MEMBER FUNCTIONS AMONG THE CLASSES OF THE COMPDB APPLICATION	91
FIGURE7-27: COMPDB: SECOND CLUSTER OF MEMBER_FUNCTIONS TABLE. RETURN TYPES OF THE MEMBER FUNCTIONS	92
FIGURE7-28: DISTRIBUTION OF THE MEMBER FUNCTIONS OF THE FIRST CLUSTER AMONG THEIR CLASSES	92
FIGURE7-29: COMPDB: THIRD CLUSTER OF MEMBER_FUNCTIONS TABLE. RETURN TYPES OF THE MEMBER FUNCTIONS	93

FIGURE7-30 COMPDB: THIRD CLUSTER OF MEMBER_FUNCTIONS TABLE. DISTRIBUTION OF THE MEMBER FUNCTIONS AMONG THEIR CLASSES	94
FIGURE7-31: COMPDB: FIRST CLUSTER OF FUNCTION_PARAMETERS TABLE TYPES OF PARAMETERS.....	95
FIGURE7-32: COMPDB: SECOND CLUSTER OF FUNCTION_PARAMETERS TABLE TYPES OF PARAMETERS..	96
FIGURE7-33 COMPDB: THIRD CLUSTER OF FUNCTION_PARAMETERS TABLE. TYPES OF PARAMETERS.....	97
FIGUREA-1: COMPDB: FIRST CLUSTER CLASSES TABLE. DISTRIBUTION OF THE VALUES THAT REPRESENT THE NUMBER OF THE PUBLIC FUNCTIONS	108
FIGUREA-2: COMPDB: FIRST CLUSTER CLASSES TABLE. DISTRIBUTION OF THE VALUES THAT REPRESENT THE NUMBER OF THE PROTECTED FUNCTIONS	109
FIGUREA-3: COMPDB: SECOND CLUSTER OF CLASSES TABLE. DISTRIBUTION OF THE VALUES THAT REPRESENT THE NUMBER OF THE PUBLIC FUNCTIONS.....	109
FIGUREA-4: COMPDB: SECOND CLUSTER OF CLASSES TABLE. DISTRIBUTION OF THE VALUES THAT REPRESENT THE NUMBER OF THE PROTECTED FUNCTIONS.....	110
FIGUREA-5: COMPDB: THIRD CLUSTER OF CLASSES TABLE. DISTRIBUTION OF THE VALUES THAT REPRESENT THE NUMBER OF THE PUBLIC FUNCTIONS.....	110
FIGUREA-6: COMPDB: THIRD CLUSTER OF CLASSES TABLE. DISTRIBUTION OF THE VALUES THAT REPRESENT THE NUMBER OF THE PROTECTED FUNCTIONS.....	111
FIGUREA-7: COMPDB: FIRST CLUSTER OF MEMBER_DATA TABLE. CATEGORY OF MEMBER DATA.....	111
FIGUREA-8: COMPDB: FIRST CLUSTER OF MEMBER_DATA TABLE. INFORMATION THAT DESCRIBES IF THE MEMBER DATA ARE POINTERS OR NOT	112
FIGUREA-9: COMPDB: SECOND CLUSTER OF MEMBER_DATA TABLE. CATEGORY OF MEMBER DATA.....	112
FIGUREA-10: COMPDB: SECOND CLUSTER OF MEMBER_DATA TABLE. INFORMATION THAT DESCRIBES IF THE MEMBER DATA ARE POINTERS OR NOT	113
FIGUREA-11: COMPDB: THIRD CLUSTER OF MEMBER_DATA TABLE CATEGORIES OF MEMBER DATA	113
FIGUREA-12: COMPDB: THIRD CLUSTER OF MEMBER_DATA TABLE. INFORMATION THAT DESCRIBES IF THE MEMBER DATA ARE POINTERS OR NOT	114
FIGUREA-13: COMPDB: FIRST CLUSTER OF MEMBER_FUNCTIONS TABLE. CATEGORIES OF THE MEMBER FUNCTIONS	114
FIGUREA-14: COMPDB: FIRST CLUSTER OF MEMBER_FUNCTIONS TABLE. NUMBER OF PARAMETERS OF THE MEMBER FUNCTIONS	115
FIGUREA-15: COMPDB: SECOND CLUSTER OF MEMBER_FUNCTIONS TABLE. CATEGORIES OF THE MEMBER FUNCTIONS	115
FIGUREA-16: COMPDB: SECOND CLUSTER OF MEMBER_FUNCTIONS TABLE. NUMBER OF PARAMETERS OF THE MEMBER FUNCTIONS	116
FIGUREA-17: COMPDB: THIRD CLUSTER OF MEMBER_FUNCTIONS TABLE. CATEGORIES OF THE MEMBER FUNCTIONS	116
FIGUREA-18 COMPDB: THIRD CLUSTER OF MEMBER_FUNCTIONS TABLE. NUMBER OF PARAMETERS OF THE MEMBER FUNCTIONS	117
FIGUREA-19: COMPDB: FIRST CLUSTER OF FUNCTION_PARAMETERS TABLE. USE OF PARAMETERS	117
FIGUREA-20 COMPDB: SECOND CLUSTER OF FUNCTION_PARAMETERS TABLE USE OF PARAMETERS ...	118

FIGUREA-21 COMPDB: THIRD CLUSTER OF FUNCTION_PARAMETERS TABLE USE OF PARAMETERS..... 118

List of Abbreviations

1. **G.U.I.:** Graphical User Interface
2. **D.B.M.S.:** Database Management System

1. Introduction

1.1 Problem Domain Description

One of the most distinctive things of modern society is the use of software systems in almost every aspect of social and economical life during the last few decades. Manufacturing industries, financial institutions, information services and construction industries are examples of the use of software systems and the increasing reliance on them. In order to be useful and have added value, these systems should meet certain criteria. They have to operate correctly, be flexible, functional and always available. In order to meet these criteria software systems may be subject to changes during their lifetime.

Management and control of these changes are of vital importance, as great amounts of time and effort are required in order to keep software systems operational after release. Several studies, which investigated the costs of changes carried out on a system after delivery, have shown that expenditures on these changes are estimated at about 40%-70% of the entire life cycle of the system [Lientz et al. 1978].

The discipline that is concerned with changes related to a software system after delivery is known as software maintenance. When trying to comprehend a legacy software system, a major problem that software maintainers encounter is that documentation may not be up – to - date or there may be lack of experienced software maintainers. Several techniques and methods have been applied in order to facilitate this time and money consuming activity. This work explores and analyses the use and application of Knowledge Discovery in Databases for maintenance of software systems. In particular, it aspires to elaborate how a data mining technique such as clustering, can help a potential maintainer to understand a software system in order to maintain it.

1.2 Dissertation Objectives

The aim of this research work is to investigate how program understanding and software maintenance can be facilitated with the help of data mining techniques like clustering. The main objectives are:

- Specification of the Model of the Input Data to be preprocessed and stored in a database. This concerns the definition of the program entities and their attributes.
- Design and implementation of the database schema in which the retrieved data is stored. Its design should facilitate the use of clustering analysis.
- Design and implementation of the Preprocessing Application. Here, it has to be stated that the term preprocessing in this work indicates the extraction of data from C++ source code and the insertion of it in a database. The application consists of the following two parts:
 - The Front-End, which is the interface of the application that the user interacts with (G.U.I.); and
 - The Back-End that extracts the data from the source code and store it in a database (Algorithm)
- Use of the Preprocessing Application in order to preprocess C++ source code data.
- Choice and use of the Data Mining Tool, which is going to be the basis of the evaluation of the Preprocessing Application's outcome.
- Evaluation of the feasibility of the Preprocessing Application's outcome in order to produce patterns which are valid, useful and novel to the potential maintainer.

At this point it has to be emphasised that this project is concerned with the development of a Preprocessing system for a semi automated approach to program understanding. This means that it is likely that the user has no expert knowledge of the program which is analysed. For this reason, the Model of the Input Data, which is provided to the Preprocessing Application, is based on the header files of a C++ application.

1.3 Dissertation Structure

The first chapter, *Introduction*, illustrates the principal issues that this dissertation is concerned with.

The second chapter, *Background*, presents the areas of software maintenance and Knowledge Discovery in Databases. A definition of software maintenance and the categories of change that can be implemented in a software system will be also outlined. What is more, an analysis of program understanding and different strategies that are followed will be given. What follows, is an analysis of the field of Knowledge Discovery and Data Mining. The Background chapter's final section is concerned with the investigation and critical analysis of previous solutions in the domain of program understanding. Such solutions make use of the Knowledge Discovery process, and more specifically Data Mining techniques such as clustering, in order to facilitate the understanding of a program.

The third chapter, *Requirements*, outlines the detailed requirements that must be satisfied when developing the system that is going to preprocess the C++ source code data. There are presented the requirements of the Model of Input Data, the Preprocessing Application's G.U.I. and Algorithm. Finally the requirements of the Data Mining Tool are also indicated. This chapter provides the framework for the remainder of this dissertation as subsequent chapters follow the outlined requirements.

The fourth chapter, *Design*, portrays the detailed design of the system that preprocesses C++ source code data. This system consists of the Model of Input Data, the Front-End and the Preprocessing Application's Back-End.

The fifth chapter, *Implementation*, explains the main features of the Model of Input Data, the GUI of the Preprocessing Application, and its Algorithm. There is a discussion of their structure, main functionalities and the implementation of the relevant theory.

The sixth chapter, *Testing*, presents the testing of the Preprocessing Application. The testing of the Preprocessing Application consists of the preprocessing of a header file and the behaviour of the G.U.I. and the Algorithm.

In the seventh chapter, *Results - Evaluation*, an examination of the accuracy of the output of the Preprocessing Application, is undertaken. Two applications, `CAccessReport` and `CompDB`, are used as samples. Their actual structure is

compared with the outcome of the clustering analysis (IBM's Intelligent Miner) of their respective Models of Input Data. The results of this comparison are analysed and the derived conclusions are outlined.

The last chapter, *Conclusion – Future work*, describes the challenges encountered, the implemented solutions and their outcome. In addition, it suggests how to further develop this approach in order to improve the understanding of C++ Source Code.

2. Background Chapter

2.1 Software Maintenance

According to IEEE *software maintenance is the modification of a software product after delivery, in order to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment* [IEEE Software Maintenance Standards 1998].

There are several reasons that motivate the maintenance of a software system. Some of them are:

- Providing continuity of service. This includes fixing bugs (Y2K is a well-known example), recovering from failures and accommodating changes in the operating system and hardware.
- Supporting mandatory upgrades: Some examples of a mandatory upgrade are a government regulation like the one that the banks' and businesses' information systems should be Euro compliant, or an attempt to maintain an advantage over competitors.
- Supporting user requests for improvements: Enhancement of functionality, better performance and customisation to local working patterns are some examples that can lead to the maintenance of a software system.
- Facilitating future maintenance work: This usually involves code and database restructuring, and updating documentation.

As it can be seen from the following figure, software maintenance is a continuation of the development of a software system.

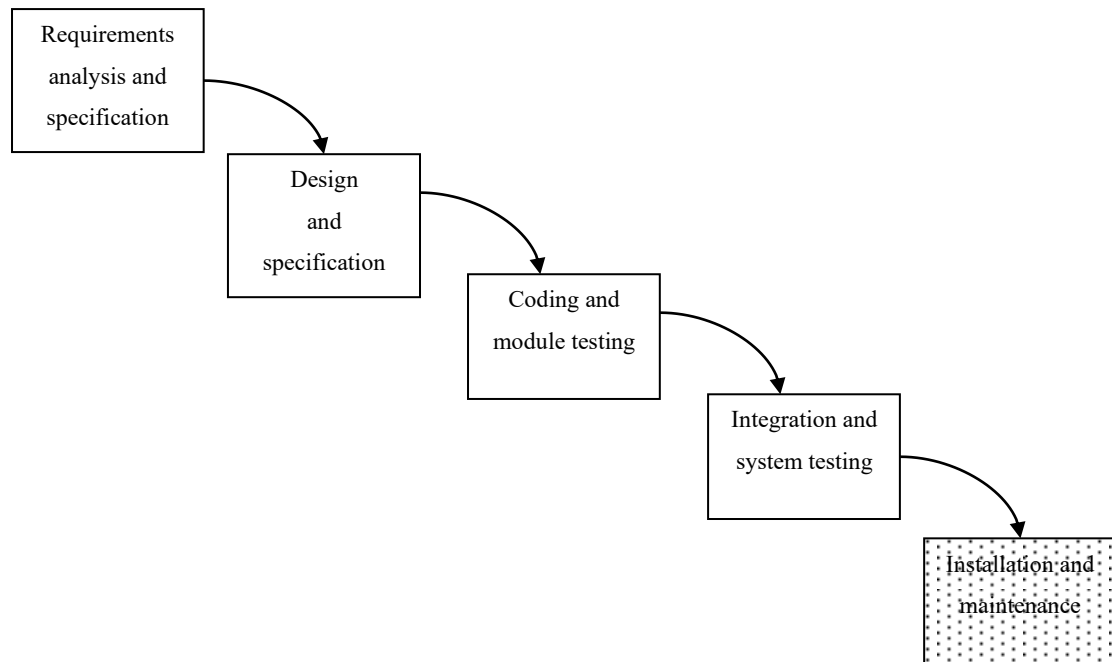


Figure 2-1: Waterfall Model of a software system life cycle [Takang, Grupp 1996]

However, there is a basic difference between the development of a software system and its maintenance. The main reason of that difference is the restrictions that the existing system imposes on its maintenance activities. In order to design an enhancement, the maintainer has to investigate the current system to abstract the architectural and the low-level design. The information retrieved can be used in order to:

- Investigate how the change can be implemented.
- Predict the possible ripple effect of the change.
- Decide what are the skills and the knowledge required in order to perform the changes [Takang, Grupp 1996].

2.1.1 Categories of software maintenance changes

In order for a system to work properly, several kinds of changes are necessary. According to IEEE the categories of changes are the following:

- Adaptive change, which is the modification of a software system performed after delivery, in order to keep a computer program usable in a changed or changing environment.
- Corrective change, which is the reactive modification of a software product, performed after delivery to correct discovered faults.

- Perfective change, which is the modification of a software system performed after delivery, in order to improve performance or maintainability [IEEE Software Maintenance Standards 1998].
- Preventive change, which is the undertaken modification of a software system, in order to prevent malfunctions.

2.2 Program understanding

Prior to implement any kind of change in a software system it is essential for the maintainer to understand it both as a whole and as the programs affected particularly by the change. This activity involves:

- Gaining a general knowledge of what the software system does and how it relates to its environment.
- Identifying where in the system changes are to performed and
- Gaining an in-depth knowledge of how the parts that are going to be corrected or modified, work [Takang, Grupp 1996].

Program understanding is hectic in terms of maintenance effort and resources. At Hewlett Packard, comprehension of source code costs \$200 million per year [Padula 1993]. Reports from industry and other sources also indicate that approximately half of the total effort expended on effecting change is allocated to understand the software system. This expenditure tends to increase especially if the maintainer tries to change code written by someone else, the documentation either does not exist or is not up-to-date, or the program structure has been deteriorated due to several years of ad hoc quick fixes. These are common problems, which the maintainers have to deal with in order to ensure that the systems stay operational and useful.

The main purpose of program reading and understanding is to be capable to implement successfully the requested changes. For this reason, the maintainer has to gain knowledge about the following aspects of the software system:

- Problem domain: This term describes environments like health care sector, telecommunications and finance. In large software systems, problems in these areas are usually broken down into sub-problems or smaller elements, each of which is handled by a different program unit such as a module, a procedure or function. In order to effect change or

simply to estimate the resources required for a maintenance task, knowledge of the problem domain in general and the sub-problems in particular is essential to direct maintenance personnel in the choice of suitable methodologies, algorithms and tools.

- **Execution effect:** This notion means the behaviour of the system during the execution. At a high level of abstraction, the maintainer needs to know or predict the results produced by the program for a given input without having knowledge of which program units contributed to the overall result or how the result was accomplished. On the other hand, on a low level of abstraction, maintainers need to know only the outcome of the execution of the individual program units. Knowledge of data flow, control flow and algorithmic patterns can make easier the accomplishment of these goals.
- **Cause – effect relation:** This term describes the causal relation between an effect and those parts of the program that caused it. Knowledge of this relation is important, especially in large and complex systems. That is because it allows maintainers to make conclusions about how components of a software program interact during execution. It also helps because it makes it easier for the maintainer to predict the scope of a change and any ‘knock – on’ effect that may arise from the change. Another reason that makes the cause – effect relation so useful is that it can be used to trace the flow of the information through the program. An unusual interruption of the flow in a point in a program may signal the source of a bug.
- **Product-environment relation:** That term describes the relation between a software system and elements of its environment. As environment, it can be considered the sum of all conditions and influences which act from outside upon the software system. Examples of these conditions are business rules, government regulations, work patterns, software and hardware operating platforms. Knowledge of this relation is vital for maintainers because it can be used to predict how changes in the elements of the environment will affect the software system in general, and the underlying programs in particular.

- **Decision-support features:** These are the attributes of the software system that guide maintainers in technical and managerial decision making processes such as option analysis, decision-making, budgeting and resource allocation. Two very good examples of these attributes are maintainability and complexity. By measuring the complexity of a system, the components of the system that require more resources for testing can be determined. Maintainability of a system on the other hand may be used as an indicator of its quality.

2.2.1 Program understanding models

Programmers and maintainers may differ in ways of thinking, solving problems and choosing techniques and tools. In general, reading about the program, about its source code and running it are the three actions involved in the process of understanding a program. These actions can be seen in the following figure.

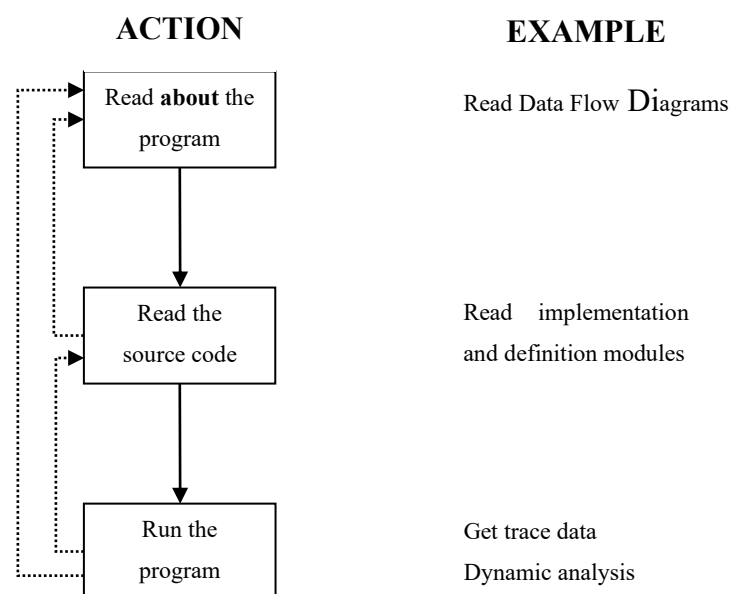


Figure 2-2: A program understanding process model [Takang, Grupp 1996]

At the first stage (read about the program) the maintainer browses, and uses different sources of information about the system, such as system documentation, in

order to develop an overview or an overall understanding of the system. This phase can be omitted if the system documentation is inaccurate, out of date or non-existent.

At the second stage (read the source code) the maintainer obtains the global and the local view of the system. The global view is used to gain a top-level understanding of the system and also to determine the scope of any knock-on effect that the change might have on other parts of the system. On the other hand the local view allows programmers to focus their attention on a specific part of the system. With this view, information about the system's structure, data types and algorithmic patterns is obtained. Bearing in mind that the system documentation may not be reliable, reading program source code is usually the principal way of obtaining information about a software product.

The third stage (run the program) aims to help the maintainer to study the dynamic behaviour of the program 'in action'. The benefit of it is that it can reveal some characteristics of the system which are difficult to obtain by just reading the source code.

2.2.2 Mental Models

The ability of a user to understand a phenomenon depends to some extent on how he/she can form a mental representation, which serves as a 'working model' of the phenomenon that has to be understood.

The phenomenon (for example, how a television set works, the behaviour of liquids, an algorithm) is known as the target system and its mental representation is called a mental model. For instance if a person understands how a television works then he has a mental model which represents this and, based on that model, he can predict the behaviour of the television set when it is turned on or when a different channel is selected. By using the model he can also explain certain observations such as the occurrence of a distorted image. How complete and accurate the model will be depends on a large extent, on the user's information need. In the case of a television set, a user who uses it solely for entertainment does not have to understand its internal composition such as the cathode ray tube and circuits and how they work, in order to be able to use it. On the other hand, a technician, who services the set in the event of a breakdown needs a deeper understanding of how the set works and thus requires a more elaborate and accurate model [Takang, Grupp 1996].

The content and formation of mental models hinges on cognitive structures and cognitive processes. Cognitive structures represent the way in which knowledge is stored in human memory. On the other hand cognitive processes describe how the knowledge is manipulated during the formation and use of mental models.

Observation, inference or interaction with the target system, are the major factors that help the formation of the mental model which changes continuously as more information about the target system is acquired. The completeness and the correctness of a mental model can be influenced by factors like the user's previous experience with similar systems and technical background. In this point it has to be underlined that the mental model may contain insufficient, contradictory or unnecessary information about the target system. But, it is not necessary for it to be complete; it just has to convey key information about the target system. For instance, if a user models a piece of software, it should at least embody the functionality of the software [Takang, Grupp 1996].

2.3 Program Understanding Strategies

A program understanding strategy is a technique used to form a mental model of the target program. As it is mentioned before, the mental model is constructed by combining information contained in the source code and by documentation with the assistance of the expertise and domain knowledge that the maintainer brings to the task. In this point it has to be emphasised that in a lot of cases the documentation of a target system is not updated or simply doesn't exist and therefore the level of the expertise of the maintainer greatly influences how easily the code can be understood. Several techniques and descriptive models have been proposed in order to help maintainers to understand programs [Takang, Grupp 1996]. In the following chapters three of them are going to be presented with the common characteristic the fact that are not based on the documentation but mainly on the source code.

2.3.1 Top down model

In this model the maintainer starts by comprehending the top-level details of a program (what it does), and gradually works towards understanding the low-level

details like data types, control and data flows and algorithmic patterns in a top-down way [Takang, Grupp 1996].

The cognitive structure and cognitive process of a mental model that result from a top-down strategy can be explained in terms of a design metaphor. Software development in its entirety can be considered to be a design task which consists of two fundamental processes; composition that represents the production of a design, and comprehension that is the understanding of that design. In composition, the maintainer maps what the program does, in the problem domain, into a collection of computer instructions of how it works, in the programming domain, using a programming language.

On the other hand, comprehension is the reverse of composition. It is a transformation from the programming domain to the problem domain and involves the reconstruction of knowledge about these domains (including any intermediate domains) and the relationship between them. This reconstruction of knowledge is concerned with the creation, confirmation and successive refinement of hypotheses. It starts with the interception of a vague and general hypothesis, known as the primary hypothesis. This is then confirmed and further refined by acquiring more information about the system, primarily from the program text and other sources like the system documentation [Takang, Grupp 1996].

Usually, the primary hypothesis is generated as soon as the maintainer encounters information concerning any aspect of the program, for example a module name. That is why; the mental model of the program takes form at the outset, even before the maintainer becomes aware of low-level semantic and syntactic details of the program [Takang, Grupp 1996].

The information required for hypothesis generation and refinement is manifested in key features –internal and external to the program- known as beacons, which serve as typical indicators of the presence of a particular structure or operation [Brooks 1983]. The top down model is reminiscent of skimming a piece of text to obtain a general, high-level understanding, and then reading the text again in detail in order to get a deeper understanding [Takang, Grupp 1996].

2.3.2 Bottom up model

The bottom up model can be used when the maintainer lacks hypotheses, or when hypotheses fail, or for close scrutiny of the relevant code [Clements et al. 1996]. Using this model, the maintainer successively recognises patterns in the program. These are iteratively grouped into high-level, semantically more meaningful structures. The process of putting together small units of information like program statements for instance, into larger units like procedures, is called chunking [Takang, Grupp 1996]. Each of these information units consists of a chunk. The high-level structures are then chunked into even bigger structures in a repetitive bottom-up fashion until the program is understood. In the following figure, a diagrammatic representation of the bottom-up model can be seen.

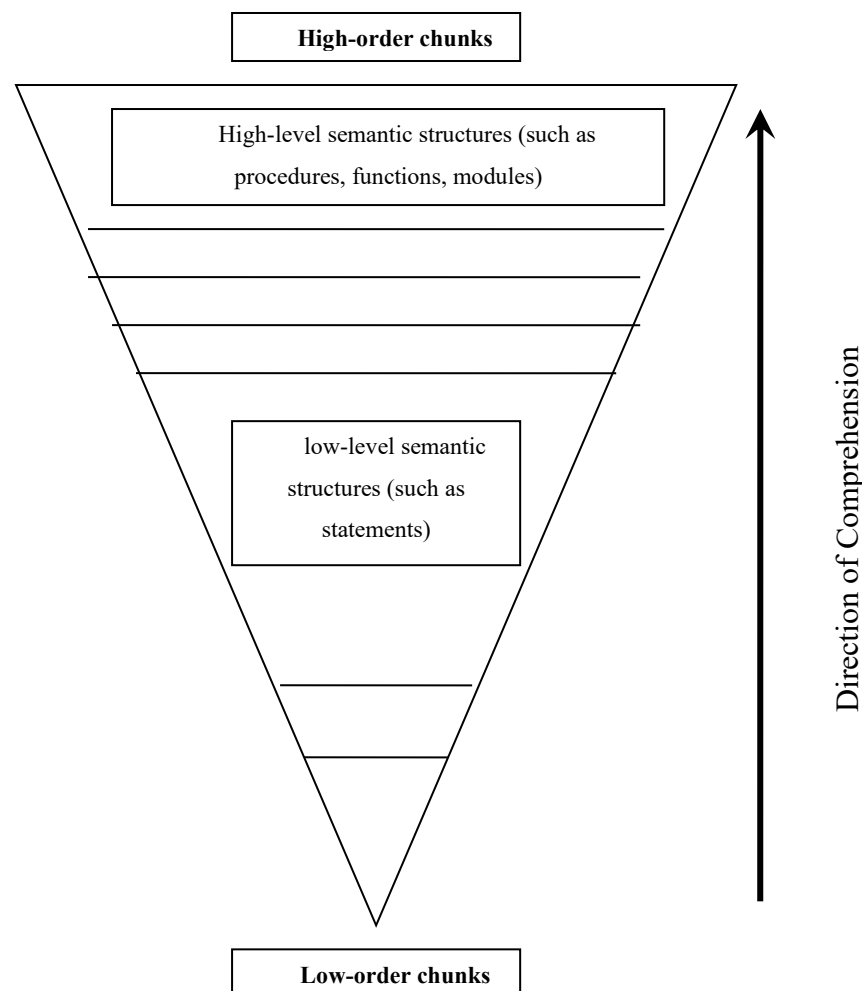


Figure 2-3: Bottom-Up comprehension process [Takang, Grupp 1996]

The chunking process tends to be faster for more experienced programmers than beginners as they recognise patterns more quickly. For instance, the following program statements:

```
MaxValue := Table[1];  
FOR Index := 2 TO 100 DO  
  IF Table[Index] > MaxValue THEN  
    MaxValue := Table[Index];  
END;  
END;
```

would be grouped, by an experienced programmer into a chunk called ‘find maximum element in array’.

In that point here, it has to be emphasised that the process of understanding a program, rarely takes place in such a well-defined fashion as these models portray. Usually, maintainers tend to take advantage of any clues they come across in an opportunistic way.

2.3.3 Opportunistic model

By using this model, the maintainer makes use of both bottom-up and top-down strategies, but not simultaneously. As Letovsky suggests, *‘the program comprehension can best be viewed as an opportunistic application of bottom-up and top-down strategies* [Clements et al. 1996]. According to this model, comprehension hinges on the following three and complementary features:

- A knowledge base, that represents the expertise and the background knowledge that the maintainer brings to the understanding task.
- A mental model, which expresses the maintainer’s current understanding of the target program.
- An assimilation process that describes the procedure used to obtain information from various sources such as source code and system documentation.

When maintainers need to understand a piece of program, the assimilation process enables them to obtain information for the system. This information then triggers the invocation of appropriate plans from the knowledge base in order to enable maintainers to form a mental model of the program that needs to be understood. The

more information is obtained, the more often the mental model changes [Takang, Grupp 1996].

2.4 Knowledge Discovery in Databases

According to Fayyad et al. *Knowledge Discovery in Databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns of data* [Fayyad et al. 1996].

Before going further, it would be useful to analyse these terms in more detail. At first, *data* is a set of facts F (for example cases in a database). The expression E that describes facts in a language L in a subset F_E of F is called *pattern*. For example the expression “If income < \$t, then person has defaulted on the loan” is a pattern for an appropriate choice of t .

In KDD the term *process* means a process consisting multiple steps such as data preparation, search for patterns, knowledge evaluation and refinement involving iteration after modification. The process is assumed to have some degree of search autonomy that is why it is called non-trivial. For instance, computing the mean income of persons in the loan example, while producing a useful result, does not qualify as discovery.

The patterns discovered should be *valid* on new data with some degree of certainty. A measure of that certainty is a function C mapping expressions in L to a partially or totally ordered measurement space M_C . An expression E in L about a subset F_E of F can be assigned a certainty measure $c = C(E, F)$.

The patterns discovered should also be *novel*. Novelty can be measured respectively to changes in data (by comparing current values to previous or expected values) or knowledge (how a new finding is related to old ones). It can be assumed that this can be measured by a function $N(E, F)$, which can be a Boolean function or a measure of degree of novelty or unexpectedness.

Patterns should also potentially lead to some useful actions, as measured by some utility function. Such a function U maps expressions in L to a partially or totally ordered measure space M_U , therefore $u = U(E, F)$.

KDD has to make patterns understandable to humans in order to facilitate a better understanding of the underlying data. Although this is difficult to measure precisely, a substitute for it is the simplicity measure. There are several kinds of simplicity, which range from the purely syntactic, for example the size of a pattern in bits, to the semantic (e.g. easy for humans to comprehend in some setting). This can be

measured, if possible by a function S mapping expressions E in L to a partially or totally ordered measure space M_S , therefore $s = S(E, F)$

In order now to measure altogether the pattern value, combining validity, novelty, usefulness and simplicity, another important notion called *interestingness* can be used. Some KDD systems have an explicit interestingness function $i = I(E, F, C, N, U, S)$ which maps expressions in L to a measure space M_I . Other systems define interestingness indirectly by using an ordering of the discovered patterns.

The definition of what knowledge is can be given as an outcome of all these definitions. The purpose of that definition is to specify what an algorithm used in a KDD process may consider knowledge. Therefore *a pattern $E \in L$ is called knowledge if for some user-specified threshold $i \in M_I$, $I(E, F, C, N, U, S) > i$* [Fayyad et al. 1996].

2.5 KDD and Data Mining

According to Fayyad et al. Data Mining is just *a step in the KDD process consisting of particular data mining algorithms that, under some acceptable computational efficiency limitations produce a particular enumeration of patterns E_j over F* [Fayyad et al. 1996].

It is important here to emphasise that the space of patterns is often not finite, and the enumeration of patterns requires some form of search in this space. The computational efficiency constraints place severe limits on the subspace that can be explored by the algorithm.

From the above-mentioned definition of Data Mining it can be derived that KDD is using data mining methods in order to extract (identify) what is knowledge according to the specifications of measures and thresholds using the database F along with any required preprocessing, sub-sampling, and transformations of F .

From these definitions, it can be seen that there is a difference between Knowledge Discovery in Databases and Data Mining. The main goal of KDD is extracting knowledge from data in the context of large databases. It involves the evaluation and possibly the interpretation of the patterns to make the decision of what constitutes knowledge and what does not. It also includes the choice of encoding schemes, preprocessing, sampling, and projections of the data prior to the data-mining step.

On the other hand, Data Mining is mainly concerned with means by which patterns are extracted and enumerated from the data.

2.5.1 Steps of the KDD process

KDD process is iterative and interactive as it involves many steps with many decisions being made by the user. In the following figure, an overview of the steps comprising the KDD process can be seen:

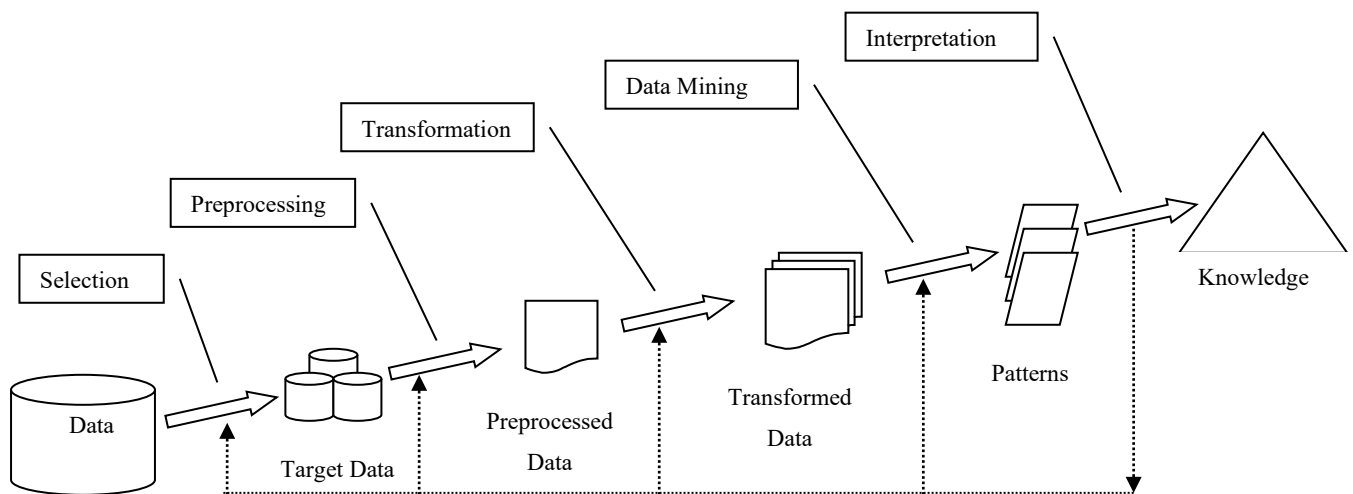


Figure 2-4: An overview of the steps comprising the KDD process [Fayyad et al. 1996]

At first, there has to be developed an understanding of the application domain, the relevant prior knowledge, and the goals of the end user.

Secondly, a target data set has to be created. In other words, a data set has to be selected, or a subset of variables or data samples has to be on focus, on which discovery is to be performed.

After that, data have to be cleaned and preprocessed. That step includes basic operations such as the removal of noise or outliers if appropriate, the collection of the necessary information to model or account for noise, to make a decision on strategies for handling missing data fields, and finally to account for time sequence information and known changes.

The next step is to reduce and project data. It means that useful features to represent the data depending on the goal of the task have to be found. Dimensionality reduction or transformation methods can be used in order to reduce the effective

number of variables under consideration or to find invariant representations for the data.

After that, the data-mining task has to be selected. In other words, there has to be a decision on whether the goal of the KDD process is classification, regression, clustering etc.

The choice of the data-mining algorithm is the next step of the KDD process. Here, the method or methods used for searching for patterns in the data have to be selected. This includes deciding which models and parameters may be appropriate and matching a particular data mining method with the overall criteria of the KDD process.

In order to continue to the next step of the KDD process a search for patterns of interest, in a particular representational form or a set of such representations, has to be conducted. In other words, the chosen data-mining task (such as classification, rules of decision trees, regression, clustering) should be executed.

After performing, the data-mining task the user can interpret mined patterns, with possible return to any of the proceeding steps for further iteration.

The final step is to consolidate the discovered knowledge. This knowledge should be incorporated into the performance system, or simply be documented it and reported to interested parties. This also includes checking for and resolving potential conflicts with previously believed (or extracted knowledge).

It has to be underlined here, that the KDD process can involve significant iteration and may contain loops between any two steps. In figure 2-1, the basic flow of steps is illustrated. Most of the focus of that process is on the data-mining step. However, the other steps are of considerable importance for the successful application of KDD in practice.

2.6 Data Mining Tasks

Practically, the two highest primary goals of data mining tend to be *prediction* and *description* [Fayyad et al. 1996]. The first term (prediction) involves the use of some variables or fields in the database in order to predict unknown or future values of other variables of interest. On the other hand, description focuses on finding human-interpretable patterns describing the data. The importance of prediction and description is relative for particular data mining applications and can vary

considerably. However, in the context of Knowledge Discovery in Databases, description tends to be more important than prediction. This is in contrast to pattern recognition and machine learning applications like speech recognition where prediction is often the primary goal.

The above-mentioned goals of prediction and description can be achieved by using the following primary data mining tasks.

2.6.1 Classification

Classification is learning a function to map (classify) a data item into one of several predefined classes [Fayyad et al. 1996]. Examples of classification methods used as part of knowledge discovery applications include classifying trends in financial markets and automated identification of objects of interest in large image databases.

2.6.2 Regression

Regression is learning a function to map a data item to a real-valued prediction variable [Fayyad et al. 1996]. Examples of regression applications are predicting the amount of biomass present in a forest given remotely-sensed microwave measurements, estimating the probability that a patient will die given the results of a set of diagnostic tests, predicting consumer demand for a new product as a function of advertising expenditure, and predicting time-series where the input variables can be time-lagged versions of the prediction variable.

2.6.3 Clustering

Clustering is a common descriptive task where the user seeks to identify a finite set of categories or clusters in order to describe the data. The categories may be mutually exclusive and exhaustive, or consist of a richer representation like hierarchical or overlapping categories. Examples of a clustering application in a knowledge discovery context include discovering homogeneous sub-populations for consumers in

marketing databases and identification of sub-categories of spectra from infrared sky measurements. In figure 2-5, a possible clustering of a loan data set into three clusters is presented:

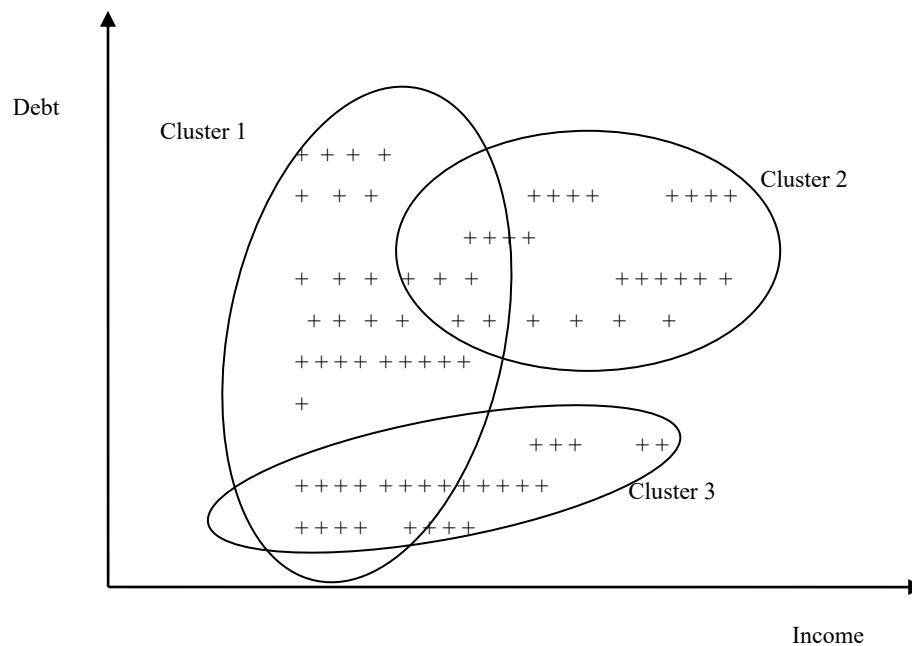


Figure 2-5: A simple clustering of a loan set into 3 clusters [Fayyad et al. 1996]

It can be seen that the clusters overlap allowing data points to belong to more than one cluster. Very close to clustering, is the task of probability density estimation that consists of techniques for estimating from data the joint multi-variant probability density function of all of the variables/fields in the databases.

2.7 KDD Process and Program Understanding: Previous Solutions

In this section, previous solutions to the domain of program understanding are going to be presented. These solutions make use of the Knowledge Discovery process and more specifically Data Mining techniques such as clustering, classification and association rules in order to facilitate the understanding a program.

2.7.1 Using Data Mining to Assess Software Reliability

In this solution, the use of data mining techniques in the assessment and maintenance of software reliability and testability is investigated [Tjortjis and Layzell 2001]. Testability is used in order to measure the structural complexity of a program. The higher the testability the more efficient the validation process is. This also improves the maintenance and the comprehension of the program.

The complexity of a program can be influenced by the coupling and the cohesion of the classes or functions. The systems that are composed by highly coupled classes are more fault-prone and hard to comprehend and maintain. A way to predict fault-prone modules is the use of data mining techniques in order to exploit metrics. Data mining techniques considered suitable in order to support software support reliability assessment, as they achieve results for large collections of data even when limited background knowledge is available [Tjortjis and Layzell 2001].

In order to facilitate the software reliability maintenance, assessment and comprehension, a methodology comprised by three steps is proposed.

In the first step, the input models are defined by selecting parts of the source code, such as functions, routines and variables in order to populate a database suitable for data mining. More specifically, the programs that are to be comprehended have to be represented as a number of entities each of them consisting of several attributes. In this solution, two models were used in order to extract data from code and populate a database [Tjortjis and Layzell 2001]. One model for C/C++ where entities are functions, that their attributes are defined according to the use and types of parameters and variables, and the types of returned values. There is also a model for COBOL source code that caters for a medium level, involving paragraphs as entities, and a low

level understanding where entities are lines of code. In both levels, attributes are binary depending on the presence of user-defined and language-defined identifiers.

In the second step, *clustering is applied to identify sub-sets of source code that are grouped together according to custom-made similarity metrics* [Tjortjis and Layzell 2001]. This approach is taken in order to group C/C++ functions, based on their similarity, into clusters, which represent subsystems. Clustering produces system's overviews, which aid comprehension. The grouping of program components into subsystems facilitates maintenance as it reduces the perceived complexity. This can be detected by the identification of subsystems, which consist of comparatively large number of functions. Large, complex and strongly interrelated subsystems are likely to be fault-prone.

Clustering can also improve systems cohesion and coherence by increasing modularity. This can improve reliability and can be done in two ways. At first, by relocating functions into modules and secondly by adjusting the processing performed within functions to reflect better the functionality designed to be encapsulated within [Tjortjis and Layzell 2001].

The last step is the application of association rules in order to establish inter-group and intra-group relationships. This data mining technique is suitable for binary attributes derived from COBOL programs. It has the potential to identify groups of variables and/or reserved words that have the tendency to appear in the same module, implying in that way that they are interrelated. This technique also identifies programming styles, by exposing patterns related to the presence of variables and reserved words in paragraphs.

Though this solution seems very interesting, is not directly related to program understanding. It has to do more with another step of software maintenance process, the software reliability assessment. However, the framework of this solution, the use of data mining techniques in software maintenance, can be applied to the domain of program understanding. The next solution that is going to be examined is related with the application of data mining techniques in order to facilitate the process of program understanding.

2.7.2 A method for Legacy Systems Maintenance by Mining Data extracted from Source Code.

This solution proposes a method for understanding and maintaining legacy software systems [Chen et al. 2002]. It is based on the use of data mining techniques for extracting interrelationships, patterns and groupings of code elements that range from variables up to modules. This solution aims to address systems both at high and at low level. It is based on the model of data mining in more conventional domains that requires data preprocessing prior to the application of algorithms. This solution was used in COBOL systems because the majority of legacy systems are still in this platform.

The first stage of this method is the systematic data preparation for extracting a number of data models and the relevant databases before applying any data mining technique. More specifically the main aim of this stage is the extraction of both high and low level information from source code and the production of a ‘mineable’ data representation. It comprises in two activities: one is finding and assembling the data set and the other is to manipulate the data in order to enhance its utility for mining. It is required to understand the objective of data mining process in advance and anticipate its likely results. The suitability of several algorithms should be known also in advance and should be reflected on the preparation data in some degree. The data preparation stage should focus on use of semantic and syntactic knowledge in order to include only the necessary data and produce a database of the appropriate size [Chen et al. 2002].

The goal of data preparation in COBOL is to extract information from source code by taking into account the grammar and the syntax of the language, setting up different tables for different data mining algorithms, and evaluating their quality and iteratively improving it. A successful data preparation in COBOL should achieve the following:

- Extract the variables of COBOL programs.
- Generate a clear view of the hierarchy of variables that will facilitate the understanding of programs and their structures.
- Design a database by capturing code structure and variables [Chen et al. 2002].

The second stage of this solution is the application of data mining techniques in the preprocessed data. For example, clustering is the most widely applied technique, focusing on high level modules of source code. Proximity among modules depends on number of accessing, and variables' transfer or 'call' among them. The data tables mainly store the module's relations descriptions. Their contents can be binary showing whether a relationship between modules exists or not, or integer that shows the number of times a relationship occurred. This number can represent the module distance.

Examples of modules relationships can include variable transfer, accessing the same file and using the same sub-modules. Different kinds of relationships may be treated as either the same or having weights according to their kind. For instance, accessing the same file can be more important than transferring a variable when determining module relationship, as file access always involves more than one variable [Chen et al. 2002].

After the use of data mining techniques, there are two types of results that are anticipated. At first there are results that represent the syntactic and semantic content of the source code. Relationships among variables and blocks of code should be identified. For this reason, it is useful to represent code by means of models or graphs, like variable relationship model (similar to an Entity-Relationship model), a variable-block relationship model (similar to the Object-Oriented model) or even models that convey a meaning similar to Data Flow Diagrams and flow charts.

The second type of results represents variables or block relationships acquired by mining association rules. For instance, a rule of the form: 'if *PRICE* exists in paragraph *P*, then *SALES* exists in paragraph *P*' with confidence *x*% and support *y*%. Each rule is characterised by its confidence, which is the percentage of times the rule is true and its support which is the percentage of the task relevant tuples for which the pattern is true [Han, Kamber 2001].

This solution seems also very interesting as is focused in the program understanding domain. But it is designed specific for source code that is written in COBOL. This model also has been tested using a small amount of data. Therefore it can be said that a solution that would be tested using large scale data and designed for different programming languages is needed.

2.7.3 Using Automatic Clustering to Produce High-Level System Organisations of Source Code

This solution proposes a collection of algorithms that were developed and implemented in order to facilitate the automatic recovery of the modular structure of a software system from its source code [Mancoridis et al. 1998]. A very common problem of software maintenance is the existence of out of date system documentation and the lack of the original system designer. Therefore the software maintainer is forced to make modifications on the source code without a thorough understanding of its organisation. This can lead to a situation that the source code organisation is so chaotic that it needs to be radically overhauled or abandoned, especially if the software system is heavily used and its requirements tend to change over time.

According to [Mancoridis et al. 1998] clustering (grouping) of related procedures and their associated data into modules (or classes) is a way for software maintainers to cope with the increasing structure complexity of a software system. In a software system, identifiable clusters of modules called subsystems, which collaborate in order to achieve higher-level system behaviour, can be found. The main problem is that this subsystem structure is not obvious from the source code structure. This solution provides an automatic technique that creates a hierarchical view of the organisation of the system based mainly on the components and the relationships that exist in the source code.

The first step of this technique is the representation of the system modules and the module-level relationships as a module-dependency graph [Mancoridis et al. 1998]. Next step is the application of algorithms in order to partition the graph in a way that the high-level subsystem structure can be derived from the component level relationships that are extracted from the source code.

In the following figure the architecture of the proposed automatic modularisation technique (environment) is illustrated. The first step of the process is the extraction of the module-level dependencies from the source code and the insertion of the resultant information in a database. The AT&T's CIA tool (for C) and Acacia (for C++) were used for this purpose. After all of the module-level dependencies have been stored in a database; the next step is the execution of an AWK script in order to query the database, to filter the query results and to produce a textual representation of the module dependency graph. Then a custom - made clustering tool called Bunch is used

in order to apply the designed clustering algorithms to the module dependency graph and emit a text-based description of the high-level structure of the systems organisation. The last step is the use of AT&T's Dotty visualisation tool to read the output file from Bunch and produce a visualisation of the results.

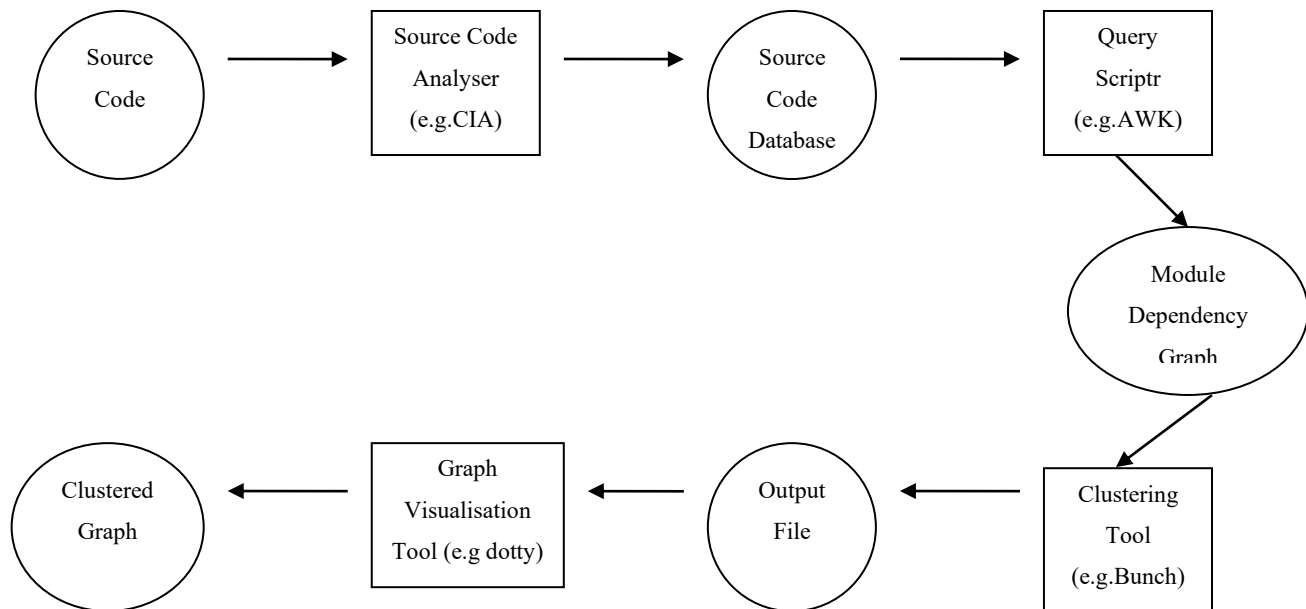


Figure 2-6: Automatic Software Modularisation Technique (Environment) [Mancoridis et al. 1998]

The basic goal [Mancoridis et al. 1998] of this modularisation technique is *to automatically partition the components of a system into clusters (subsystems) so that the resultant organisation concurrently minimises inter-connectivity (that is, connections between the components of two distinct clusters) while maximising intra-connectivity (connections between the components of the same cluster)*. According to [Mancoridis et al. 1998] clustering here can be seen as an optimisation problem that its goal is to maximise an objective function based on a trade off between inter and intra-connectivity. The basic assumption that underlies this approach is that a well-designed software system is organised in to cohesive clusters that are loosely interconnected. Based on the concepts of cohesion and coherence [Mancoridis et al. 1998] introduce in this solution three parameters:

- **Intra-Connectivity:** It is a measure of connectivity between the components that are grouped in the same cluster. A high degree of intra-connectivity indicates a good subsystems partitioning as the modules grouped within a

common subsystem share many software-level components. On the other hand, a low level of intra-connectivity indicates poor subsystem partitioning because the modules assigned to a particular subsystem share few software-level components. By maximising the intra-connectivity measurement, the likelihood that, changes made to a module are localised to the subsystem that contains the module, is increasing.

- **Inter-Connectivity:** It is a measure of connectivity between two distinct clusters. A high degree of inter-connectivity indicates a poor subsystem partitioning. A great number of inter-dependencies make software more complicated because changes to a module may affect many other parts of the system due to the subsystems interrelationships. A low degree of inter-connectivity indicates that the individual clusters of the system are to a large extent, independent.
- **Modularisation Quality:** It is defined as a measurement of the quality of the modularisation in a particular software system. Moreover, it can be said that Modularisation Utility (MQ) is the subtraction between the average intra-connectivity and the average inter-connectivity. It is bounded between -1 (no cohesion between subsystems) and 1 (no coupling between subsystems).

The clustering algorithms that are used on this modularisation technique are the Optimal Clustering Algorithm and the Sub-Optimal Clustering Algorithm. The first one evaluates the MQ for each partition and selects the one with the largest MQ as the optimal solution. The Optimal Algorithm can be successfully applied to systems of up to 15 modules. The second one, the Sub-Optimal Algorithm, generates a random partitioning and then tries to find the neighbouring partitions with the highest MQ as the sub-optimal solution.

This fully automated technique can help the programmers who lack familiarity with a system, and system architects who want to compare documented modularisations with the automatically derived ones or to improve the design of the system by learning from the differences between the modularisations. But the main problem of this solution is that as the number of the produced files exceeded the 20, calculation time was greatly increased.

2.7.4 A Software Evaluation Model Using Component Association Views

This solution proposes a model for the evaluation of the architectural design of a system based on the association between the components of the system. This association is defined as a measure of the overall dependency among high-level system components such as files, modules or subsystems, with regard to a collection of criteria [Sartipi 2001], and is a generalisation of cohesion and coupling metrics.

According to [Sartipi 2001], a component is defined as a named group of system entities. Coupling is a measure of the relative interdependence among the modules of a system and is measured based on the complexity of the interface between the modules [Sartipi 2001]. The cohesion is the measure of the relative functional strength of a module and is usually measured by techniques based on program slicing [Sartipi 2001]. These properties (measures) can be obtained by measuring among components, their inter- and intra-components associations.

This proposed solution allows the measurement of the modularity of the system, as an indication of the quality of the system design and its decomposition into subsystems. For this reason the following three association views of a system are generated:

- Control passing: It represents the correlation among the system components based on function invocation [Sartipi 2001].
- Data exchange: It epitomises the correlation among the system components based on aggregate data types that are either passed as parameters between two functions or are referenced by a function. This view excludes any parameter passing with simple data types such as integer, real, boolean and string since they fail to show enough evidence of correlation between the functions [Sartipi 2001].
- Data sharing view: It signifies the correlation among the system components based on sharing the global variables by the functions [Sartipi 2001].

By using these techniques, the following three design properties of a system based on sharing, passing, or encapsulating the state of the system regardless of the system's adopted design methodology and architectural style are proposed:

- State sharing: In this design property, the components (such as file, module, or subsystem) perform the desired operation of the system through accessing and modifying of a number of global variables. In systems that have this property, common coupling is the dominant association between the components. System sharing is manifested by a large number of references from different system functions to these global variables, while the other design properties are less visible [Sartipi 2001].
- State passing: In this design property, the state of the system is kept in data structures and the system operation is performed by changing and passing these data structures among different modules. In such systems, the coupling between components is mostly of the form stamp and data coupling [Sartipi 2001].
- State encapsulating: In this design property, the state of the system is encapsulated in the modules and the system task is performed by invoking different services to be performed by the modules using their own states. It is known to be the best from the understandability and maintainability perspective. In such systems the dominant couplings are control passing and stamp passing [Sartipi 2001].

In this approach the software system is modelled as an attributed relational graph with system entities as nodes and data-control-dependencies as edges. In this point, the application of data mining techniques, like clustering and association rules (more specifically the application of A priori algorithm) helps the decomposing of the graph into domains of entities based on the association property. The next step is to populate a database of these domains [Sartipi 2001].

3. Requirements Chapter

3.1 Introduction

The aim of this project is to explore how a data mining technique like clustering facilitates the understanding of a software system which is under maintenance. A major problem that software maintainers face when trying to comprehend a legacy software system is the lack of an up – to – date documentation and of experienced software maintainers. By using data mining techniques, such as clustering, this investigation work tries to contribute to this stage of the lifecycle of a software system.

The system designed consists of the following parts:

- *The Model of the Input Data*, which is the specification of the program entities and their attributes.
- *The Preprocessing Application*, which consists of the following two parts:
 - The Front-End, which is the interface of the application that the user interacts with (G.U.I.)
 - The Back-End that extracts the data from the source code and stores them in the database (Algorithm)
- *The Data Mining Tool* that is the backbone of the evaluation of the Preprocessing Application's outcome.

This chapter outlines the characteristics and requirements of the system performing the preprocessing and the clustering analysis of the C++ source code data. From the above brief description of the system's parts, emerge the following categories of requirements:

- Requirements of the Model of Input Data, which are described in the §3.2.1.
- Requirements of the Preprocessing Application, which are going to be explained in the §3.3.
 - Requirements of the Front-End (GUI), which are depicted in the §3.3.1.
 - Requirements of the Back-End (Preprocessing Algorithm), which are described in the §3.3.2.

- Requirements of the Data Mining tool, which are illustrated in the §3.4.

3.2 Model of the Input Data

The Model of the Input Data has to be defined, which means that the program entities and their attributes ought to be specified. This is a very important step because the selected model is going to be the basis for the clustering analysis of the data. Therefore, the Model of the Input Data has to facilitate the application of this data mining technique.

3.2.1 Requirements of the Model of the Input Data

The Model of the Input Data provided to the data mining tool, which performs clustering, is of vital importance and has to be carefully considered. There is a wide range of alternatives considering the exact phase of the Input Model's design, as several possibilities have to be taken under consideration before developing such a model. However, in addition to the model's design there are also some requirements concerning the selection of the program entities that will be analysed in §4. These requirements are the following:

- 1) The number of an entity's attributes should not be small as in such a case can lead to a poor description of the entity. Attributes provide a means of comparison and informative description for an entity. Therefore, a sufficient number of them have to be outlined, in order to avoid misleading comparisons.
- 2) The chosen entities must be described by a common set of attributes in order to achieve homogeneity. This allows the comparison of the entities on the basis of their attributes, which is the main principle of cluster analysis.
- 3) The entities must be applicable to all programs. That means that they have to be clearly named in a program. For instance, functions and classes can be defined as entities as they exist in every program of a software system.
- 4) The entities must also be associated with an appropriate proportion of the source code when the program is modelled as a collection of them. This ensures that most of the program is covered by the analysis.

- 5) The set of attributes describing a program's entities must be clearly defined. Both binary and qualitative attributes can be identified, as they are predominant in a source code application domain.

3.3 Preprocessing Application

The next step is the design of the application that preprocesses the C++ source code. The application consists of two parts. The Front – End, which is the interface that the user interacts with (G.U.I.) and the Back-End (Algorithm), which is the part that is related with the extraction of the data from the source code and their insertion in the database. At this point, it has to be clarified that the term preprocessing refers to the parsing of the C++ source code, the extraction and storing of the retrieved data to a database.

3.3.1 Requirements of the Front-End Part (G.U.I.)

The requirements of the front-end part of the preprocessing application are:

- 1) The application should have a user friendly interface, in order to be easier to interact with. The term friendly implies that the interface must be:
 - i) *Understandable*: It should not require any previous knowledge from the user and it has to be as simple as it can be [<http://developer.kde.org/>]. Possible users of the system can be considered as software maintainers and in general anyone who is related to the area of program comprehension.
 - ii) *Task-suitable*: It must not present an excessive degree of functionality which could confuse the user or harm functionality itself [<http://developer.kde.org/>].
 - iii) *Tolerant of mistakes*: It has to allow users to make mistakes without any harmful consequences (error trapping).
 - iv) *Feedback-rich*: It should always give immediate feedback to the user regarding the actions that are being taken [<http://developer.kde.org/>]. All results in different stages of

processing must be displayed to the user in order to enable him to observe them and restart the process from an early stage in case that something goes wrong.

3.3.2 Requirements of the Back-End Part (Algorithm)

The requirements of the Front-End part that preprocesses source code are:

- 1) The data that is retrieved from the source code must be stored in a mode that facilitates the application of cluster analysis. This is further analysed in the following chapter (§4).
- 2) The application should process a large scale of data and respond to each task performed in a reasonable amount of time. There should be a large scale of data as it is likely that the derived results can be more reliable and have a better quality.

3.4 Requirements of the Data Mining Tool

The data mining tool used in the evaluation of the outcome of the Preprocessing Application has to satisfy the following requirements:

- 1) It has to be reliable and deliver measurable results.
- 2) It has to be easy to learn and use.
- 3) The results have to be easy to understand and store.

4. Design Chapter

4.1 Introduction

The design phase of the project consists of the following three parts (listed according to their importance):

- *The definition and the design of the Model of the Input Data:* This is the most important part, because according to it, the schema of the database is designed and the functions of the application are defined.
- *The design of the database schema:* In this case, the database schema in which the retrieved data is stored has to be designed. This is another vital part because the more flexible and well designed the database schema is, the easier the application of any data mining technique will be.
- *The design of the preprocessing application:* The application consists of two parts: The G.U.I., which is the interface of the application that the user interacts with; and the Algorithm that extracts the data from the source code and stores it in the database: It is an important step because during this phase, data is retrieved from the source code and inserted to the database. However, the successful function of this application is strongly related to the right design of the Model of the Input Data and the schema of the database.
- *The choice of the Data Mining Tool:* This is the final step of the design phase of the project. It is important as it is the basis of the evaluation of the Preprocessing Application's outcome.

4.2 Design of the Model of Input Data

The basic idea for the Model of the Input Data is indicated in the following figure:

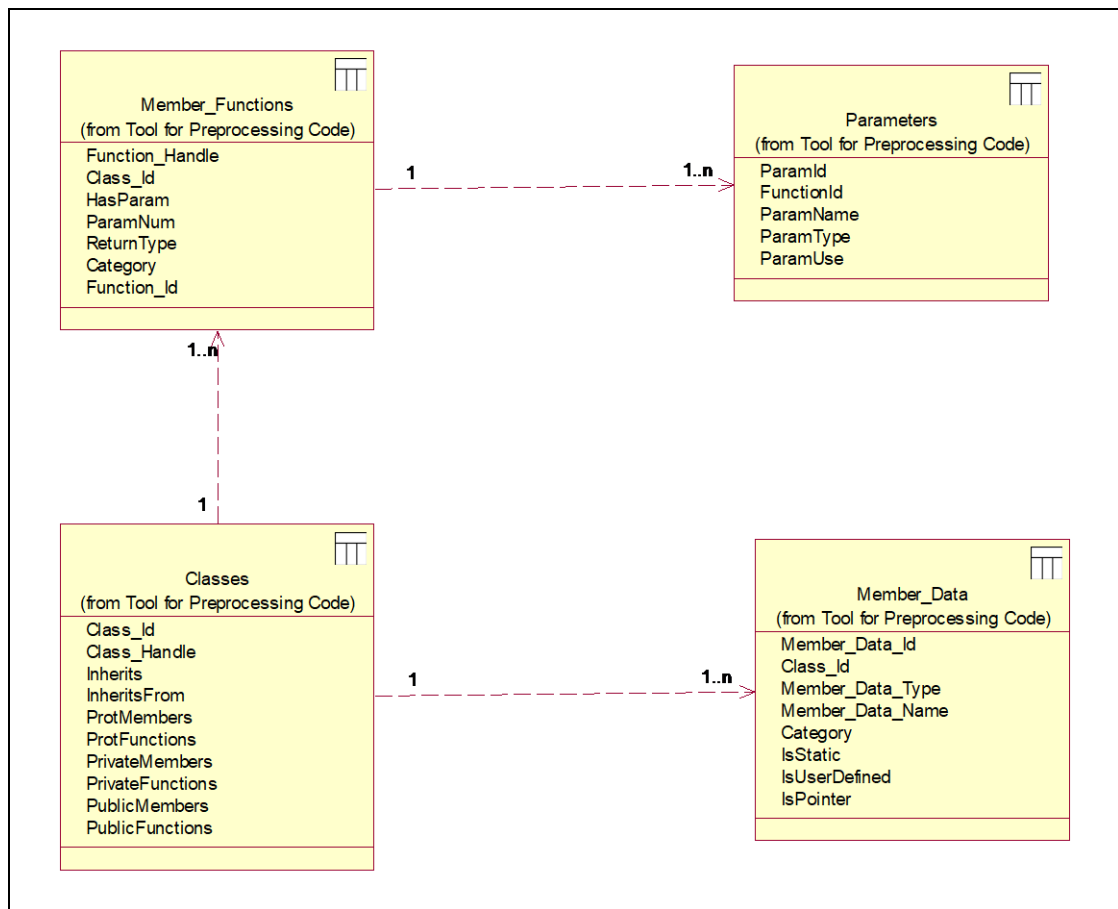


Figure 4-1: Basic Idea of the Database Store of the Model of the Input data

The above relational schema consists of the following four entities:

- Classes
- Member Functions
- Parameters
- Member Data

Each entity is described by certain attributes that are stored in respective tables of a database. In order to understand in depth the Model of the Input Data a description of the entities and their respective attributes is undertaken:

- 1) Classes: It is a representation of a conceptual grouping of similar terms, which combines data and functions in order to describe an individual entity [http://www-ksl-svc.stanford.edu:5915]. This entity can be chosen in order

to achieve a high level of detail as it can serve as a plan or a template [Lafore 1999]. The following entity class attributes can be identified in:

- i) The name of the class (**Class_Name**)
- ii) The property of inheritance (**Inherits**): It is a binary attribute, which implicitly defines whether a class is a subclass or not.
- iii) The name of the super class, if there is one (**InheritsFrom**). This is a qualitative attribute.
- iv) The number of the protected member data (**ProtMembers**), where in this case it is a quantitative attribute. Member data in general is the data items within a class [Lafore 1999]. There can be any number of data members in a class.
- v) The number of the protected member functions (**ProtFunctions**). This is a quantitative attribute. Member functions in general, are the functions that are included in a class [Lafore 1999]. They are also called methods.
- vi) The number of the private member data (**PrivateMembers**). This is a quantitative attribute.
- vii) The number of the private member functions (**PrivateFunctions**), which is a quantitative attribute.
- viii) The number of the public member data (**PublicMembers**), which is a quantitative attribute as well.
- ix) The number of the public member functions (**PublicFunctions**) that is a quantitative attribute.

2) Member Functions: It groups a number of program statements into a unit. This entity can be chosen in order to achieve a medium level of detail as it contributes to the conceptual organisation of a program. The following attributes of the entity function can be defined as:

- i) The name of the function (**Function_Handle**).
- ii) The information describing whether the function has parameters or not (**HasParam**). This is a qualitative attribute.
- iii) The number of the parameters (**ParamNum**). This is a quantitative attribute and describes the number of the parameters that a function has.

- iv) The type of the returned value of the function (**ReturnType**), which is also a qualitative attribute. It can be found before the function name in both the declaration and definition parts. Every return value of a function has a specified data type such as **char**, **short**, **int**, **long**, **float**, **double** and **long double**. If the function does not return any value, then the return type is **void**.
 - v) The category of the member function (**Category**). This is also a qualitative attribute. Member functions can be either private (they can only be accessed from inside the class) or public (they can only be accessed from outside the class) or protected (they can only be accessed by member functions in its own class or in any class derived from the particular class). Usually member functions are public. In this point, there has to be emphasised that the first idea of the design of the Model of the Input Data was to have three binary fields (**IsPublic**, **IsPrivate**, **IsProtected**) in order to define the category of the member function. However, something like that would cause unnecessary increment of the similarity of the records of the Member Functions entity.
- 3) Function parameters (**Parameters**): A parameter is a piece of data passed from a program to the function [Lafore 1999]. There can be different types of parameters like constants, variables of several data types (such as **char**, **int**, **long**), or even entire structures. The data types of variables and constants, which are used as parameters, must match those specified in the function's declaration and definition. The following attributes are describing the characteristics of this entity:
- i) The name of the parameter (**ParamName**): This is a qualitative attribute.
 - ii) The type of the parameter (arguments) (**ParamType**), which is also a qualitative attribute.
 - iii) The use of parameter (**ParamUse**), which is qualitative still. It describes how the parameters are passing to the function. There are two ways of doing that: passing by value and passing by reference. In the first one, the function creates copies of the arguments passed

to it. On the other hand, the mode of passing arguments by reference makes use of a different mechanism. Instead of passing a value to the function, a reference to the original variable in the calling program is passed. More specifically, it is the memory address of the variable that is passed [Lafore 1999].

- 4) Member Data: This is the data items within a class. There can be any number of member data in a class [Lafore 1999]. The following attributes of the entity member data are defined as:
- a. The type of the member data (**Member_Data_Type**), which is a qualitative argument. This refers to the different data types that a data member can be assigned to, such as `char`, `short`, `int`, `long`, `float`, `double`, `long double`, and `bool`.
 - b. The name of the data member, which is also a qualitative attribute.
 - c. The category of the member data (**Category**) is a qualitative attribute as well. Member data can be either private (they can only be accessed from inside the class) or public (they can only be accessed from outside the class) or protected (they can only be accessed by member functions in its own class or in any class derived from this particular class). Usually member data is private. In this point, there has to be stated that the first idea of the design of the Model of the Input Data was to have three binary fields (`IsPublic`, `IsPrivate`, `IsProtected`) in order to define the category of the data member. However, something like that would cause unnecessary increment of the similarity of the records of the Member Data entity.
 - d. The information which describes whether a data member is static (**IsStatic**) or not. This is a binary attribute. If a data item in a class is declared as static, then only one such item is created for the entire class, no matter how many objects there are. Static data items are useful when all objects of the same class must share a common item of information. They are visible only within the class, but their lifetime is the entire program. They continue to exist even if there are no items of the class [Lafore 1999].
 - e. The information that describes whether a data member is a pointer (**IsPointer**) or not. This is a binary attribute as well. A pointer is a

variable that holds an address value [Lafore 1999]. It is one of the most basic characteristics of C++ and its use is essential in order to best profit from the language.

- f. The information which describes if a data member is user-defined (`IsUserDefined`) or not. This is also a binary attribute. *Enumeration* is an example of a user-defined type. It functions when the programmer desires to know in advance a finite (usually short) list of values that a data type can take on [Lafore 1999]. An enumeration example is the following:

```
Enum days_of_week {Sun, Mon, Tue, wed, Thu, Fri,  
Sat}
```

4.3 Design of the Preprocessing Application

The particular application consists of two parts: The Front-End, which is the interface of the application that the user interacts with (G.U.I.); and the Back-End that extracts the data from the source code and stores it in the database (Preprocessing Algorithm). This is an important step because during this step the data is retrieved from the source code and inserted to the database. However, the successful function of this step is strongly related to the right design of the Model of the Input Data and the schema of the database. In this phase it has to be defined that the term preprocessing means the parsing of the C++ source code, the extraction and the storing of the retrieved data to a database. The following paragraphs will describe the main functionalities of both the Front-End and Back-End parts of the application.

4.3.1 Functionalities of the Front-End (G.U.I.)

The Front-End of the application is simple and easy for the user to understand it. The functions of it are:

- Start Preprocessing, which is depicted in the following diagram:



Figure 4-2: Open File Function

In this function the user has the ability to choose a header file (files of type “*.h”) and open it in order to start the preprocessing of it. In particular, this functionality starts the execution of the Preprocessing Algorithm.

- Provide Feedback to the User for the progress of the preprocessing, which is presented in the following diagram:

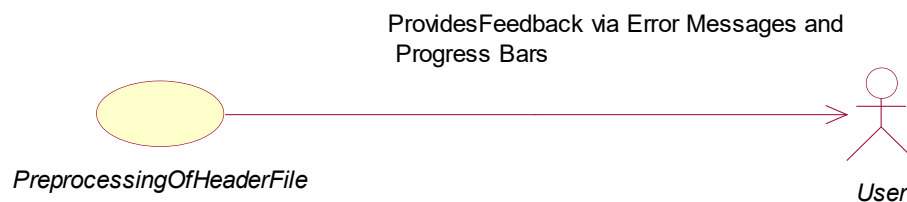


Figure 4-3: Provide Feedback to the User Function

As long as the preprocessing of the header files progresses, the user is provided with feedback. This feedback can be:

- Error messages, in order to ensure that the user will be able to observe the progress of the preprocessing process and restart it from an early stage in case that something goes wrong
- Progress Bar, which depicts the progress of the extraction and the insertion of the data in the database
- Suitable messages in text labels that will inform the user about the stage of the Preprocessing Algorithm

4.3.2 Description of the Back-End (Algorithm)

The Back-End (Preprocessing Algorithm) of the Preprocessing Application is the core of it, as it is responsible for the extraction of the data from the source code and their storing in the database. In other words, the Back-End is the part, which performs the actual preprocessing and sends feedback to the Front-End.

The steps of the Preprocessing Algorithm are illustrated in the following diagram:

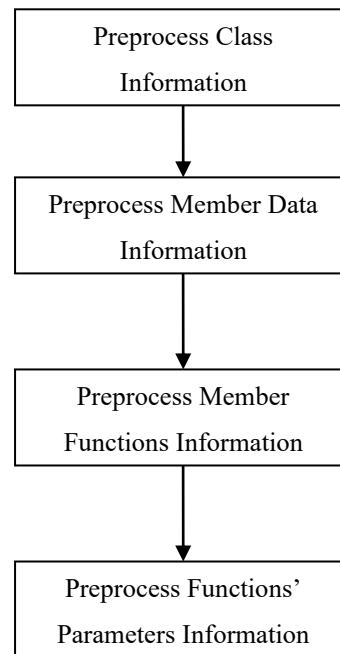


Figure 4-4: Preprocessing Algorithm

According to the above diagram, this Algorithm is based on the top-down model of program understanding. It starts by preprocessing the information that concerns the top-level details of the program (such as information about the class), and gradually works towards understanding lower-level details such member functions and their parameters, and member data in a top-down way.

More specifically the Preprocessing Algorithm will start by preprocessing the information that describes the Class entity. It extracts the handle of the class, the name of its super-class (if exists) and the number of the member data and functions. If the first step is successful then the algorithm continues into the next stage.

The second step of the Preprocessing Algorithm is the preprocessing of the information that describes the member data of the class. The name of the variable, its type and category (public, protected, private) are extracted, and information describing whether the variable is static, pointer or user-defined is also derived. As soon as this step finishes successfully, the algorithm moves on to the third step.

This step of the Preprocessing Algorithm is the preprocessing of the information that is related to the member functions of the class. More specifically the name of the function, its return type and category (public, protected, private), as well as the number of its parameters (if it has any) are extracted and inserted into the database.

When this step is accomplished with success, the algorithm continues onto the last step.

The last step of the algorithm is related to the preprocessing of the information that is related with the parameters of the member functions of the class. The name of the parameter, its type and use (by value, by reference) are extracted and inserted into the database.

4.3.3 Functionalities of the Back-End (Algorithm)

The structure of the Back-End (Algorithm) of the Preprocessing Application is presented in the diagram 4-5:

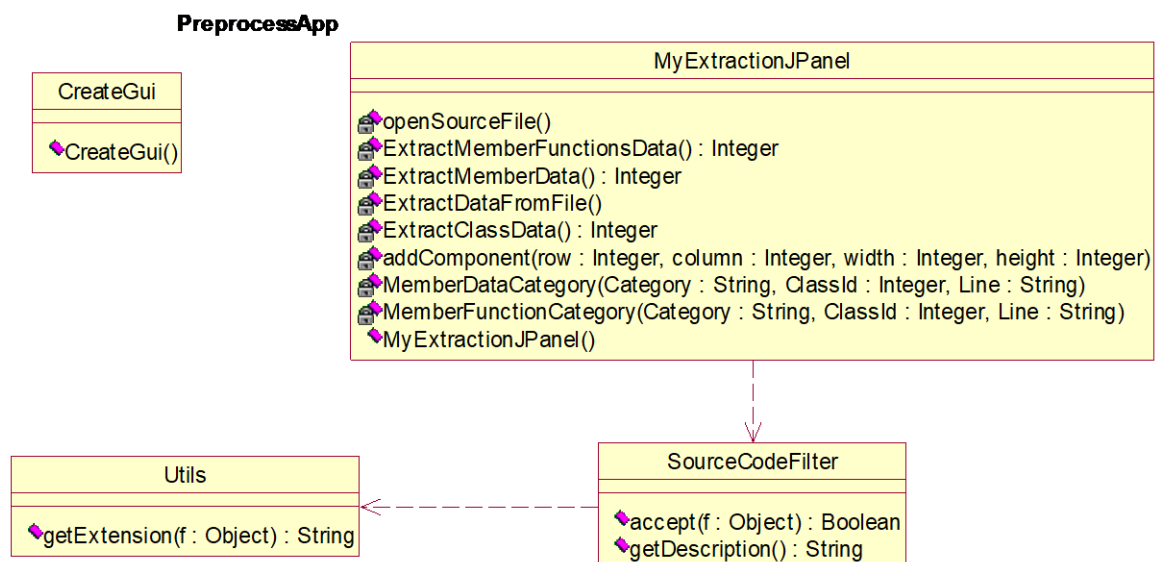


Figure 4-5: Class Diagram of the Back-End (Algorithm) of the Preprocessing Application

As illustrated above, the Back-End consists of the **PreprocessApp** package that includes the following classes:

- 1) **CreateGUI**: It is the class that creates the basic elements of the Front-End (G.U.I.) of the Preprocessing Application. In particular, it creates the desktop pane, which has the menu of the functions of the Preprocessing Application. **CreateGUI** has the following operations (Member Functions):
 - i) **CreateGui**, which is the constructor of the class.

- 2) **MyExtractionJPanel**: It is the most significant class, as it implements the Back-End (Algorithm) of the Preprocessing Application. It consists of the following Member Functions:
- i) **ExtractDataFromFile**: It is the function which implements the Algorithm as depicted in the diagram 4-4. It opens the file that is chosen by the **OpenSourceFile**, and calls **ExtractClassData** first. After the successful completion of **ExtractClassData**, it calls the **ExtractMemberData** and if this call is also successful then continues with **ExtractMemberFunctionsData**.
 - ii) **openSourceFile**: It is the function implementing the filtering of the files. The user can not choose any type of file he desires. There is a filter defined by the creation of an object of type of the class **SourceCodeFilter** that allows the opening of only header (*.h), and the C++ source code files (*.cpp).
 - iii) **ExtractClassData**: It is the function that implements the preprocessing of the data describing a class. It reads the header file line by line and extracts the handle of the class, information about inheritance (if the class inherits and the name of the superclass), the number of the protected member data and functions, the number of private member data and functions and finally takes out the number of the public member data and functions.
 - iv) **ExtractMemberData**: It is used in order to implement the preprocessing of the information that describes the member data of the class. It extracts the class ID where the member data belongs to and reads the header file line by line in order to extract the category (private, protected, public) of the member data. This is defined according to which section the line of the header file resides on. If the line resides on the private section then the category is private, if on public then the category is public and so on. As soon as the class ID and the category of the function are extracted the function **MemberDataCategory** is invoked in order to continue with the preprocessing of the rest of the information describing the member data.

- v) **MemberDataCategory**: It is the function which implements the preprocessing of the information describing the member data of the class (except of the category and the class ID). It takes as parameters the Category of the data (protected, private, and public) and the class ID that belongs to. It extracts the name of the variable, its type; and the information that defines whether it is static, pointer and user defined.
- vi) **ExtractMemberFunctionsData**: It is used in order to implement the preprocessing of the information that concerns the member functions of the class. It extracts the class ID where the member function belongs, and reads the header file line by line in order to extract the category (protected, private, and public) of the member function. This is defined according to which section resides the line of the header file. If the line resides in the private section then the category is private, if in public then the category is public and so on. As soon as the class ID and the category of the function are extracted, the function **MemberFunctionCategory** is called to continue with the preprocessing of the rest of the information describing the member function.
- vii) **MemberFunctionCategory**: It is the function implementing the preprocessing of two types of information: The first one describes the member functions of the class and the second describes their respective parameters (if they have any). It takes as parameters the Category of the function (protected, private, and public), and the ID of the class where it belongs. It extracts the name of the function, its return type, and the information that defines whether it has parameters and if any it extracts their number. Following that and in case that the member function has parameters, the **MemberFunctionCategory** continues by preprocessing the information that describing the member function's parameters. It extracts the name of the parameter, its type and its use (by value or by reference). However, if the member function does not have any parameters then the **MemberFunctionCategory** ends and returns the control to the **ExtractMemberFunctionsData**.

- viii) **addComponent**: It is used to add the components of the “File Preprocessing” form and has four parameters: The row and the column that define where the component will be placed, as well as its width and height defining its size.
 - ix) **MyExtractionJPanel**: As its name suggests it is the constructor of the **MyExtractionJPanel** class. According to [Deitel & Deitel 1999] a constructor is a method with the same name as the class. It is invoked automatically each time an object of that class is instantiated and it cannot specify return types or return values.
- 3) **SourceCodeFilter**: It is the class that implements the filter that is used in the opening of the C++ header and source code files. It consists of the following functions:
- i) **accept**: It defines whether the given file is accepted by the filter or not. This is done by creating an object of type of the class **Utils** and by using its **getExtension** method in order to get the extension from the given file and compare it with the approved ones.
 - ii) **getDescription**: It provides the description of the file. In the file chooser of the Preprocessing Application, for example, the description is “Only C++ Files”. The following picture (4-6) presents the outcome of the use of the filter that is implemented by the **SourceCodeFilter** class:

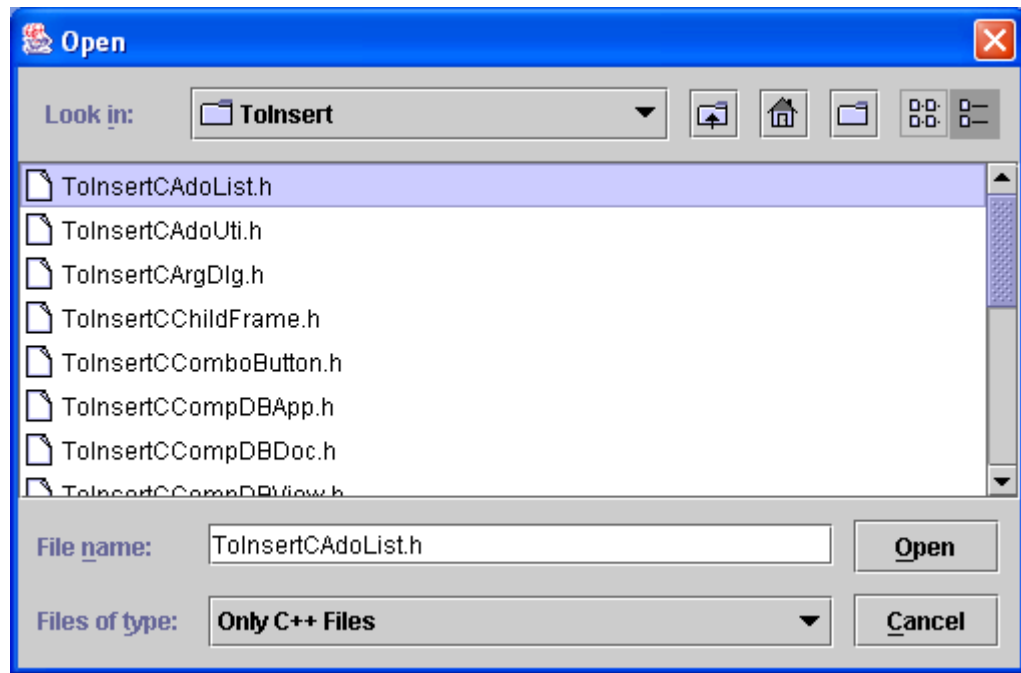


Figure 4-6: File Chooser of the Preprocessing Application with the C++ Files Filter implemented

As presented in the above picture, the user can only see the C++ type of files (header files or source code files).

- 4) **utils:** It is the class which objects are used by **SourceCodeFilter** in order to provide the extension of the file. This extension is used to define whether the file is approved or not. The particular class has the following function:
 - i) **getExtension:** It gets the extension of the file.

5. Implementation Chapter

5.1 Implementation phase and tools

As the previous chapter suggested, the system will include the Preprocessing Application that parses the C++ source code and a database management system where the data is stored in a mode capable for performing cluster analysis. The DBMS (Database Management System) used is the SQL Server 2000, which can scale up easily and can handle great amounts of data. The following figure (5-1) highlights the structure of the system:

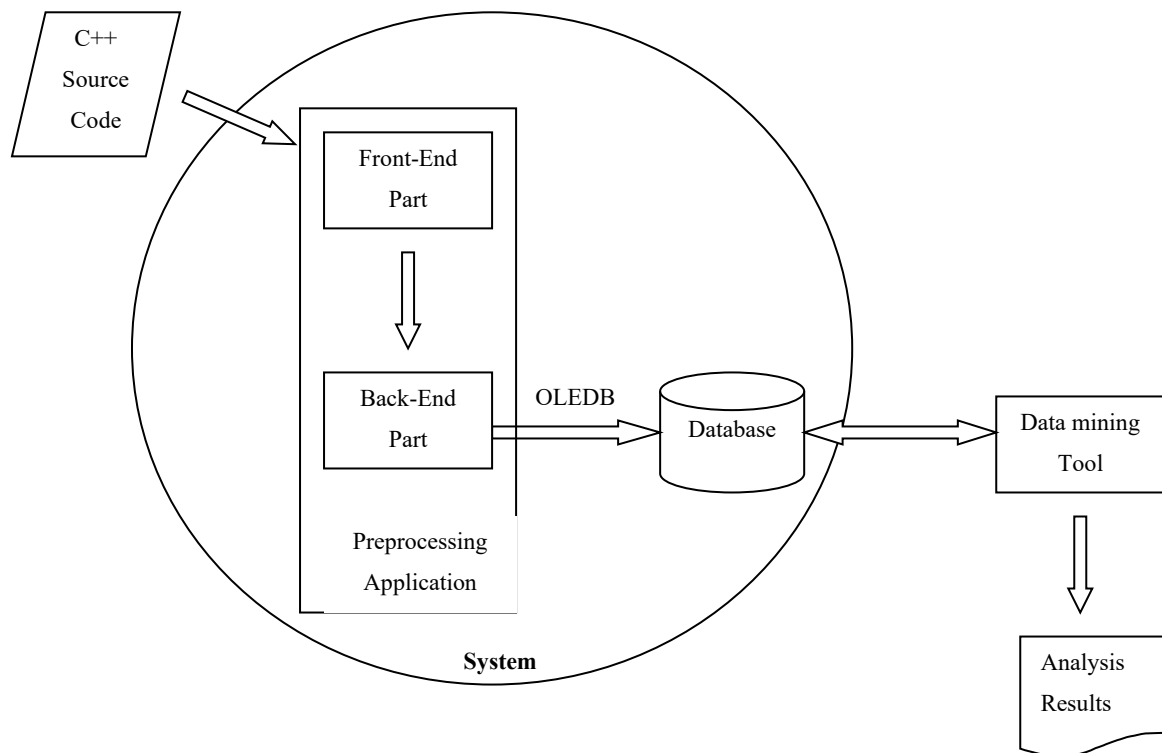


Figure 5-1: Structure of the system

The programming language that is used in order to implement the Preprocessing Application is Java. It is a widely used object oriented language that enables the quick development of a great range of applications. Gaining knowledge and experience on such a language, functions as a further motivation for using it. The chosen data mining tool for the analysis of the preprocessed data is the IBM's Intelligent Miner. It is easy to use and it presents the results in a format easy to store and understand. Moreover, another advantage of the Intelligent Miner is that it is widely used in the market and uses proven algorithms.

5.2 Implementation of the Model of Data

The schema of the PreprocessingDB database, which implements the Model of the Input Data is portrayed in the following picture (5-2):

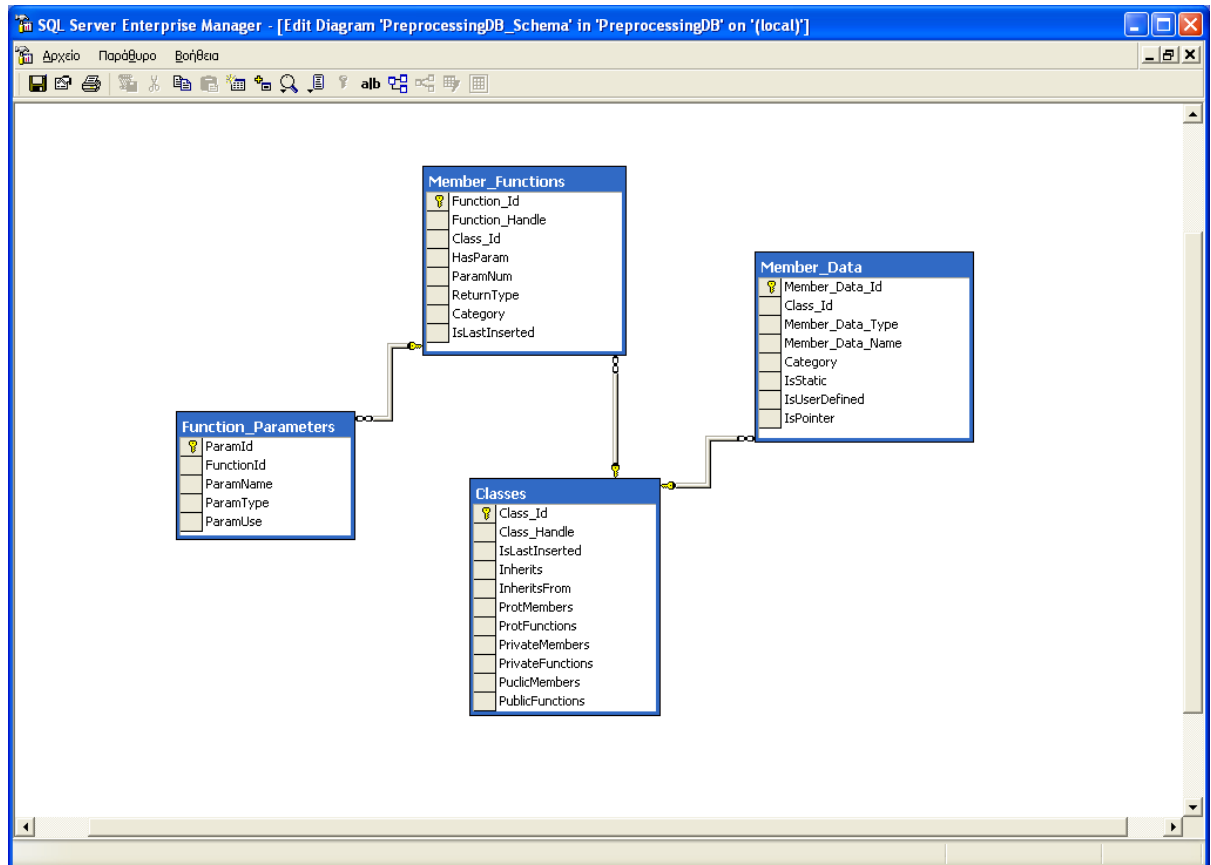


Figure 5-2: PreprocessingDB Database Schema

The above picture depicts that the database contains the following tables:

- 1) **Classes**, which has the following fields:
 - i) **ClassId**: This is the ID of the class, and it is generated automatically from the database.
 - ii) **Class Handle**: It is the name of the class
 - iii) **Inherits**: It defines whether the class inherits from a base class. In other words it defines if the class has a super-class or not.
 - iv) **InheritsFrom**: In case the class inherits this field defines the base class.
 - v) **ProtMembers**: It identifies the number of the Protected data members.

- vi) **ProtFunctions**: It outlines the number of the Protected Member Functions.
 - vii) **PrivateMembers**: It defines the number of the Private data members.
 - viii) **PrivateFunctions**: It describes the number of the Private Member Functions.
 - ix) **PublicMembers**: It defines the number of the Public data members.
 - x) **PublicFunctions**: It stores information concerning the number of the Public Member Functions.
- 2) **Member_Data**, which further consists of the following fields:
- i) **Member_Data_Id**: It is the ID of the Member_Data, and is generated automatically from the database.
 - ii) **Class_Id**: It is a foreign key to this table, as it is the ID of the class that the Member Data belongs to.
 - iii) **Member_Data_Type**: It is the type of the Member_Data, (for example int, char, and float).
 - iv) **Member_Data_Name**: It is the name of the Member Data.
 - v) **Category**: It defines the category that the Data Member belongs (private, protected, and public). It is a qualitative characteristic.
 - vi) **IsStatic**: It defines whether the Data Member is Static or not. It is a qualitative characteristic.
 - vii) **IsUserDefined**: It defines whether the Data Member is User Defined (such as an object).
 - viii) **IsPointer**: It defines whether the Data Member is a Pointer or not.
- 3) **Member_Functions**, that includes the following fields:
- i) **Function_Id**: It is the ID of the function, and is generated automatically from the database.
 - ii) **Function_Handle**: It is the name of the function.
 - iii) **Class_Id**: It is a foreign key to this table, as it defines the ID of the class that the Member Function belongs to.

- iv) **HasParam**: It defines whether the Member Function has Parameters.
 - v) **ParamNum**: It is the number of the Parameters that the Member Function may have.
 - vi) **ReturnType**: It is the type of the returning value of the Member Function.
 - vii) **Category**: It defines the category that the Member Function belongs to (private, protected, and public). It is a qualitative characteristic.
- 4) **Function_Parameters**, that contains the following fields:
- i) **ParamId**: It is the ID of the parameter, and is generated automatically from the database.
 - ii) **FunctionId**: It is a foreign key to this table, as it is the ID of the Member Function where the parameter belongs.
 - iii) **ParamName**: It is the name of the parameter.
 - iv) **ParamType**: It is the type of parameter (such as int, char, float etc.)
 - v) **ParamUse**: It defines the kind of the parameter's use (by value or by reference).

5.3 Implementation of the Preprocessing Application

5.3.1 Front-End (G.U.I.)

The Front-End of the Preprocessing Application uses a multiple document interface (MDI), which is a main window (often called the parent window) containing other windows (often called child windows) in order to manage several open documents that can be processed in parallel. It consists of the following parts:

- 1) The “C++ Source Code Preprocessing Application” window, which is the parent window and has the menu with the functionalities of the Preprocessing Application. According to picture 5-3 the form contains the following components:
 - i) A menu bar that includes the menu of the Preprocessing Application

- ii) A menu called “Preprocess”, which is an element included as a member of the menu bar. Menus in general display a list of member menu items, when clicked. Selecting the menu item usually results in some action performed by the application [Piroumian 1999]. More specifically, in the case of Preprocessing Application, when clicking the “Preprocess” menu, the “File...” menu item appears. When the user selects it, the “File Preprocessing” form appears.

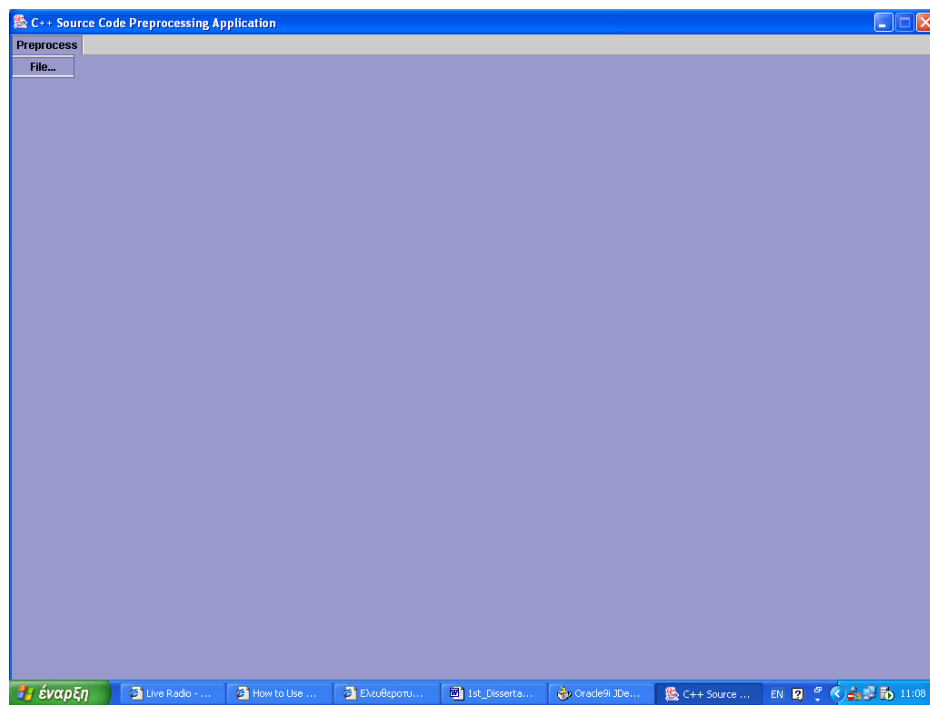


Figure 5-3: “C++ Source Code Preprocessing Application” window

- 2) The “File Preprocessing” form: It is the child window and it is presented in the picture 5-4:

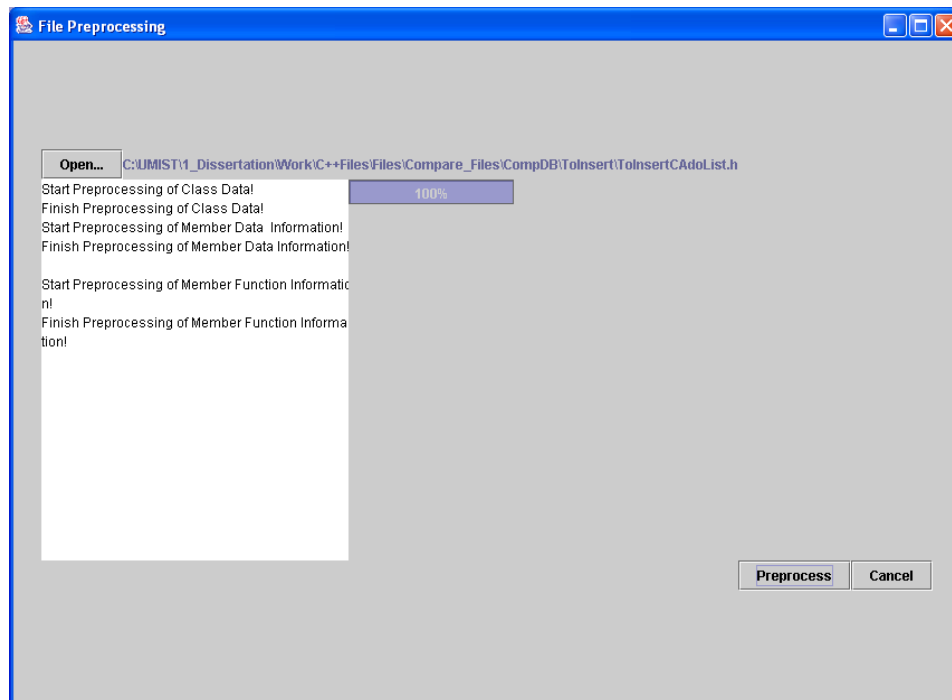


Figure 5-4: “File Preprocessing” form

As illustrated the form contains the following components:

- i) The “Open” command button, which is used in order to open the file that is going to be preprocessed. When clicking it, the user calls a `JFileChooser`, from which he/she can choose the file that is going to be opened in order to be preprocessed.
- ii) A label used in order to provide some information to the user. In particular, as the user opens the file, the label displays the path and the name of it.
- iii) A text area that is provided to the user in order to help him understand which stage of the Preprocessing Algorithm the application performs.
- iv) A progress bar that provides indication to the user concerning the preprocessing of source code that takes place, and the amount of work that has been accomplished.
- v) The “Preprocess” command button, which is used in order to trigger the execution of the Preprocessing Algorithm. When clicking it, the user calls the `ExtractDataFromFile` that will be described in the §5.3.2.

- vi) The “Cancel” command button, which is used in order to cancel the execution of the Preprocessing Algorithm.

5.3.2 Back-End (Algorithm)

The Back-End (Preprocessing Algorithm) of the Preprocessing Application is responsible for the extraction of the data from the source code and its storing in the database. In other words, the Back-End is the one that performs the actual preprocessing and sends feedback to the Front-End.

The execution of the Algorithm starts when the user clicks the “Preprocess” button in the “File Preprocessing” form. Following this action, the function `ExtractDataFromFile`, which is the core of the Algorithm, is invoked. Its purpose is to invoke all functions that preprocess information concerning the class, its member data and its functions with their respective parameters.

It initiates the value of the Progress Bar and calls the `ExtractClassData` in order to retrieve the information that describes the classes. When the execution of this function starts, the message “Start Preprocessing of Class Data!” is displayed in the text area, informing the user that the preprocessing of the information of the class has started. As soon as the information is inserted in the `PreprocessingDB`’s `Classes` table, the message “Finish Preprocessing of Class Data!” is displayed in the text area, the value of the progress bar increases and the value 1000 is returned to the `ExtractDataFromFile`. The particular value indicates that the function completed its execution without any errors. If an error occurs, the value 999 is returned, and the execution of the `ExtractDataFromFile` stops. The following picture (5-5) illustrates the successful preprocessing of the information describing a class in a header file:

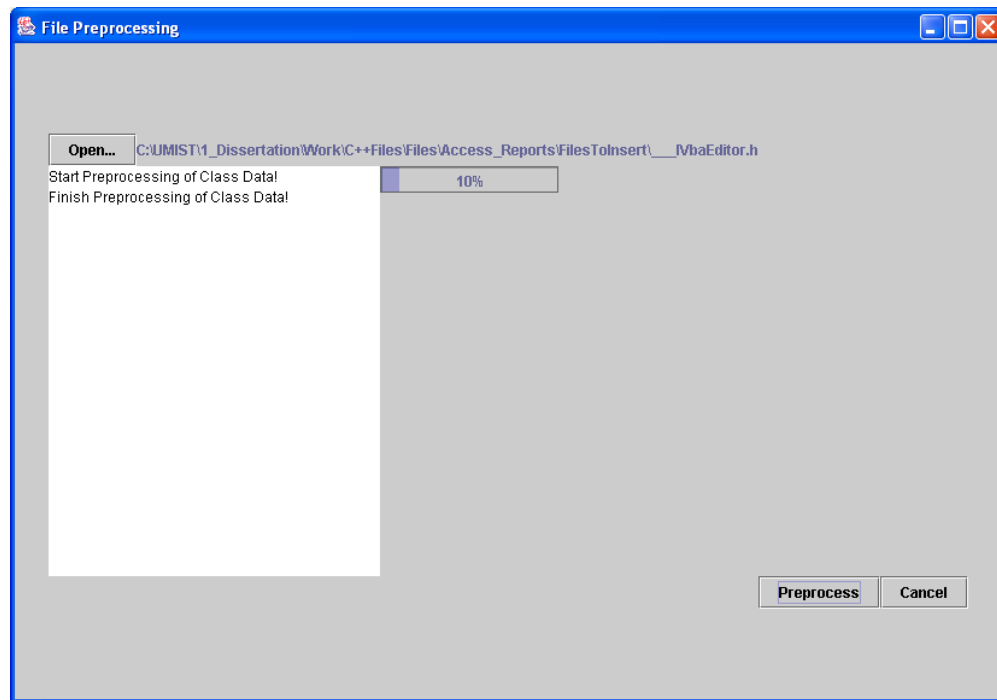


Figure5-5: Successful execution of the `ExtractClassData` function

If the `ExtractClassData` returns 1000, then the `ExtractDataFromFile` continues by invoking the `ExtractMemberData`. When the execution of this function starts, the message “Start Preprocessing of Member Data Information!” is displayed in the text area. Then the function `MemberDataCategory` is called in order to complete the preprocessing of the information describing the member data. As soon as all this information is extracted, it is inserted in the `PreprocessingDB`’s `Member_Data` table. When the execution of the `MemberDataCategory` is completed successfully, the control is returned to the `ExtractMemberData` function, which returns the value 2000 to the `ExtractDataFromFile`. This particular value indicates that the function completed its execution without any errors. If an error occurs, then the value 1999 is returned, and the execution of the `ExtractDataFromFile` stops. The message “Finish Preprocessing of Member Data Information!” is displayed in the text area and the value of the progress bar increases. The following picture (5-6) depicts a successful preprocessing of the information describing the member data of the class:

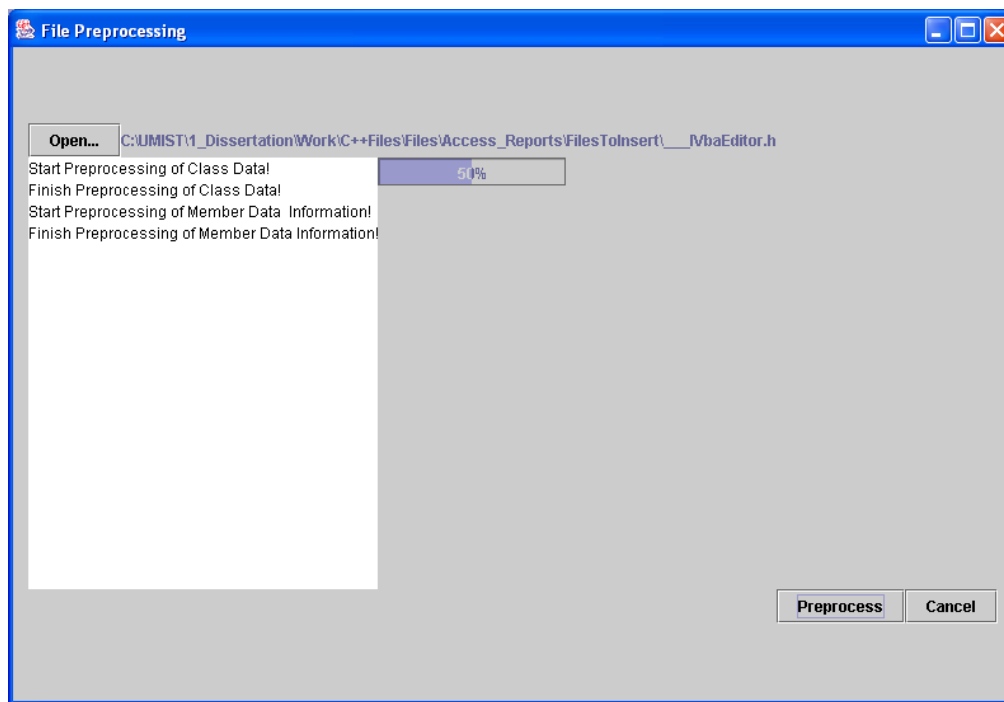


Figure5-6: Successful execution of the functions `ExtractMemberData` and `MemberDataCategory`.

If the `ExtractMemberData` returns 2000, then the `ExtractDataFromFile` continues by invoking the `ExtractMemberFunctionsData`. When the execution of this function starts, the message “Start Preprocessing of Member Function Information” is displayed in the text area. Then the function `MemberFunctionCategory` is called in order to complete the preprocessing of the information describing the member functions. As soon as all the information is completed, it is inserted in the `PreprocessingDB`’s `Member_Functions` table. Next, if the member function has parameters, the `MemberFunctionCategory` continues by preprocessing the information that describes the member function’s parameters. When all this information is extracted, it is inserted in the `PreprocessingDB` (`Parameters` table) and the control is returned to the `ExtractMemberFunctionsData`. On the other hand, if the member function does not have any parameters, then the `MemberFunctionCategory` ends and returns the control to the `ExtractMemberFunctionsData`. The message “Finish Preprocessing of Member Function Information!” is displayed in the text area and the value of the progress bar increases. The following picture (5-7) is an example of the

successful preprocessing of the information describing the member functions of a class.

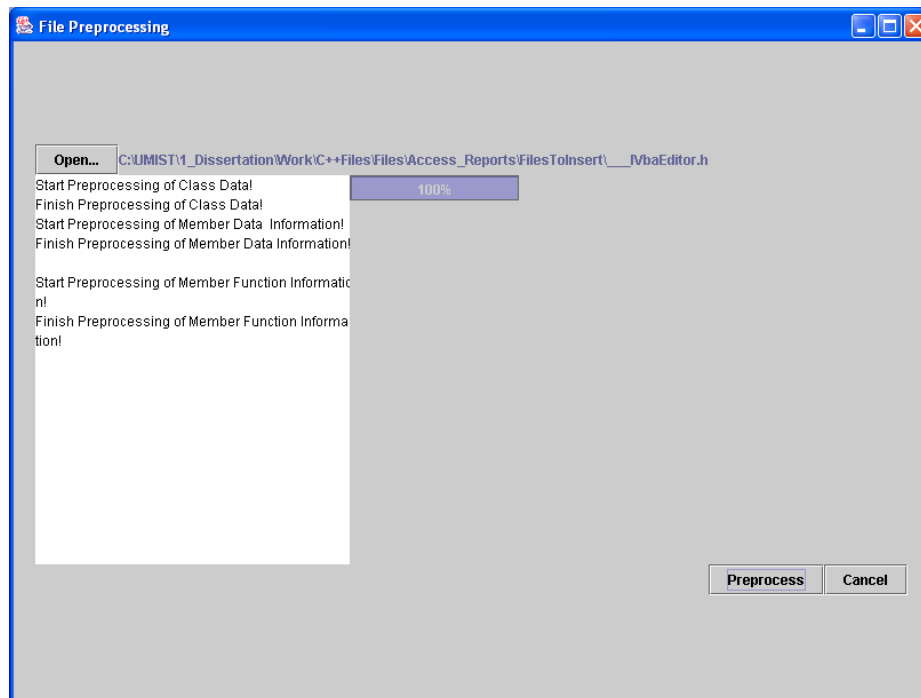


Figure5-7: Successful execution of the functions `ExtractMemberFunctionsData` and `MemberFunctionCategory`.

6. Testing Chapter

6.1 Introduction

This chapter will present the testing of the Preprocessing Application. This testing consists of the preprocessing of a header file and the behaviour of the Front-End (G.U.I.) and the Back-End (Algorithm). Moreover, the size of the input data required (header file), the total time and effort needed to extract the information from the source code in order to insert it in the database, are factors that should also have to be tested.

6.2 Testing of the Preprocessing Application

A testing script is followed in order to examine whether the Preprocessing Application meets the requirements that were imposed during the design phase, and are described in the design chapter of this report. According to this script, a header file has to be opened and preprocessed. The desired result is the successful insertion of the extracted information in the respective tables of the `PreprocessingDB` database and the right feedback from the G.U.I of the Preprocessing Application.

The header file that will be used for the testing of the Preprocessing Application is the “`_Form.h`”. Its size is 8 KB which makes it the biggest file in the `CAccessReport` system. It consists of 239 member functions and has no member data.

6.2.1 Step 1: Preprocessing the “ `_Form.h`”

This step will test whether the second requirement of the section §3.3.2 and the requirement of the section §3.3.1 are satisfied. More specifically, the ability of the Preprocessing Application processing a reasonable amount of data, responding to each task performed in a satisfying amount of time; and reflecting the “user-friendliness” of it, will be tested.

The expected results are:

- 1) The path and the name of the “`_Form.h`” file should be displayed in the label of the “File Preprocessing” form.

- 2) The information indicating the stage of the “_Form.h”'s preprocessing should be written in the text area of the “File Preprocessing” form. More specifically the following six sentences should be displayed, informing the user that the preprocessing of the header file was successful:
 - i) “Start Preprocessing of Class Data!”
 - ii) “Finish Preprocessing of Class Data!”
 - iii) “Start Preprocessing of Member Data Information!”
 - iv) “Finish Preprocessing of Member Data Information!”
 - v) “Start Preprocessing of Member Function Information!”
 - vi) “Finish Preprocessing of Member Function Information!”
- 3) The progress bar should depict the progress of the extraction and the insertion of the data in the database.
- 4) The time needed for the preprocessing of the “_Form.h” should be less than ten seconds.

The preprocessing of the “_Form.h” file lasted 8 seconds and 79 nsec. The following picture (6-1) indicates that the expected results are satisfied.

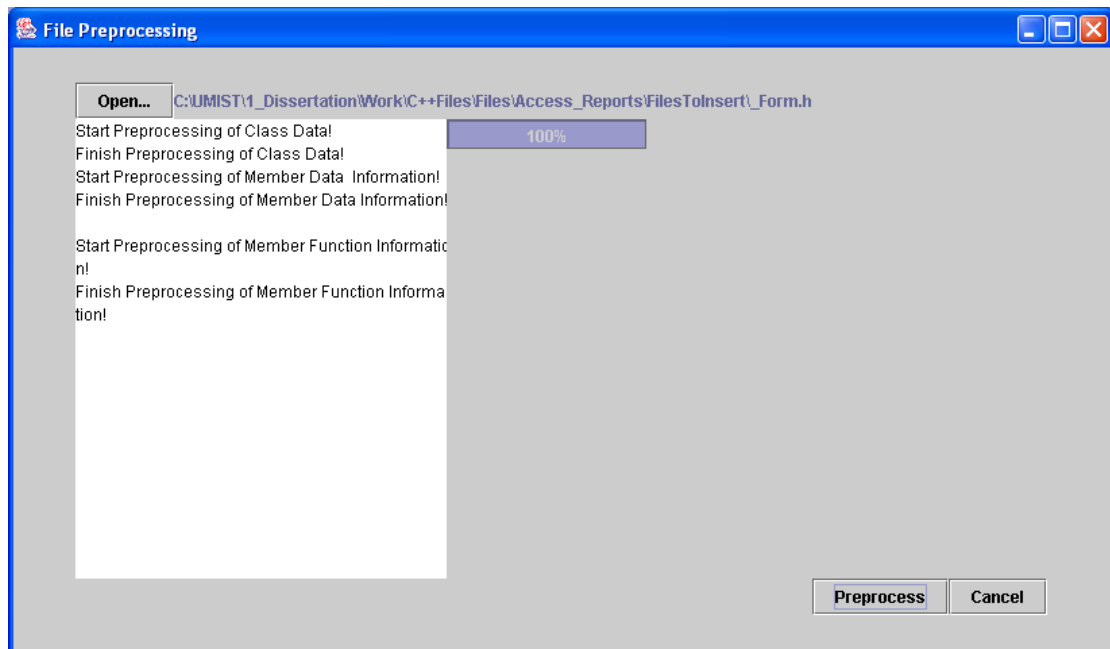


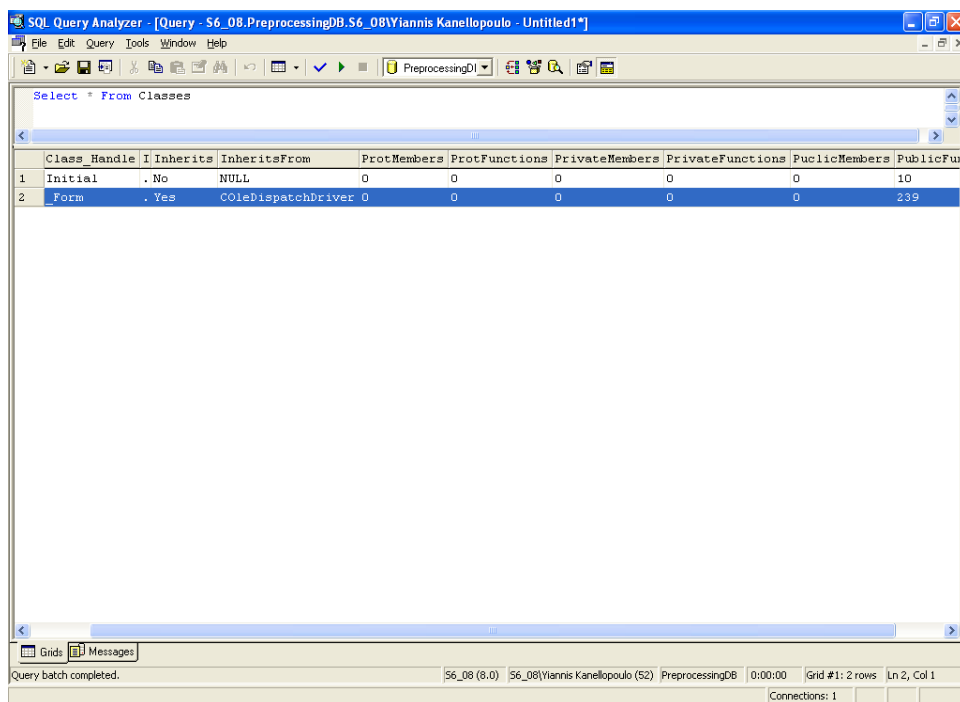
Figure6-1: Feedback provided to the user about the preprocessing of the _Form.h file

6.2.2 Step 2: Insertion of data in Classes table

This step will verify that the data describing the class `_Form`, has been inserted in the `Classes` table of the `PreprocessDB` database. The expected results are:

- 1) The `Classes` table should have a new record.
- 2) The field `Class_Handle` should have the value “`__Form`”.
- 3) The field `Inherits` should have the value “`Yes`”.
- 4) The field `InheritsFrom` should have the value “`COleDispatchDriver`”.
- 5) The field `ProtMembers` should have the value 0.
- 6) The field `ProtFunctions` should have the value 0.
- 7) The field `PrivateMembers` should have the value 0.
- 8) The field `PrivateFunctions` should have the value 0.
- 9) The field `PublicMembers` should have the value 0.
- 10) The field `PublicFunctions` should have the value 239.

The following picture (6-2) illustrates the successful insertion of the data which describes the class `_Form`, in the `Classes` table of the `PreprocessingDB`:



The screenshot shows the SQL Query Analyzer interface with a query result grid. The query is 'Select * From Classes'. The grid contains two rows of data. The first row is 'Initial' with various fields set to 0 or NULL. The second row is 'Form' with 'Inherits' set to 'Yes', 'InheritsFrom' set to 'COleDispatchDriver', and 'PublicFunctions' set to 239. The status bar at the bottom indicates 'Query batch completed.' and 'Grid #1: 2 rows Ln 2, Col 1'.

	Class_Handle	Inherits	InheritsFrom	ProtMembers	ProtFunctions	PrivateMembers	PrivateFunctions	PublicMembers	PublicFunctions
1	Initial	.No	NULL	0	0	0	0	0	10
2	Form	.Yes	COleDispatchDriver	0	0	0	0	0	239

Figure6-2: Insertion of data in the `Classes` table

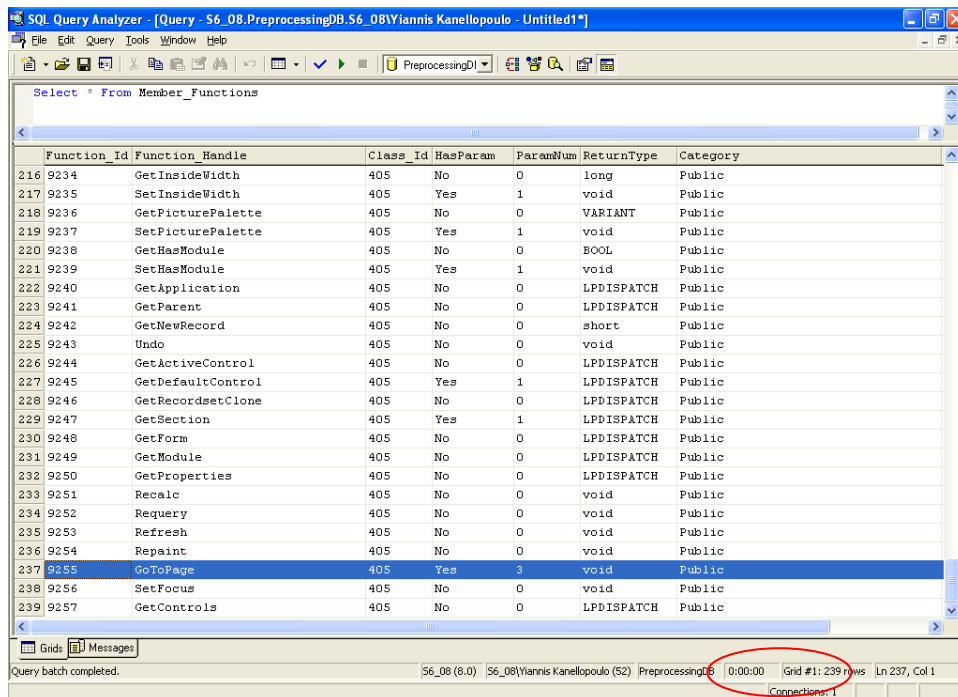
6.2.3 Step 3: Insertion of data in Member Functions table

This step will confirm that the data describing the member functions of the class `_Form`, has been inserted in the `Member_Functions` table of the `PreprocessDB`

database. Due to the fact that the number of the member functions of the class `_Form` is large, the function `GoToPage` will be used as a sample for the examination. The expected results are:

- 1) The `Member_Functions` table should have 239 new records (that is, the total number of the member functions of the class `_Form`).
- 2) The field `Class_Id` should have the value “405”.
- 3) The field `Function_Handle` should have the value “GoToPage”.
- 4) The field `HasParam` should have the value “Yes”.
- 5) The field `ParamNum` should have the number 3.
- 6) The field `ReturnType` should have the value `void`.
- 7) The field `Category` should have the value `Public`.

The following picture (6-3) portrays the successful insertion of the data that describes the member functions of the class `Form`, in the `Member_Functions` table, of the `PreprocessingDB`:



Function_Id	Function_Handle	Class_Id	HasParam	ParamNum	ReturnType	Category
216 9234	GetInsideWidth	405	No	0	long	Public
217 9235	SetInsideWidth	405	Yes	1	void	Public
218 9236	GetPicturePalette	405	No	0	VARIANT	Public
219 9237	SetPicturePalette	405	Yes	1	void	Public
220 9238	GetHasModule	405	No	0	BOOL	Public
221 9239	SetHasModule	405	Yes	1	void	Public
222 9240	GetApplication	405	No	0	LPGDISPATCH	Public
223 9241	GetParent	405	No	0	LPGDISPATCH	Public
224 9242	GetNewRecord	405	No	0	short	Public
225 9243	Undo	405	No	0	void	Public
226 9244	GetActiveControl	405	No	0	LPGDISPATCH	Public
227 9245	GetDefaultControl	405	Yes	1	LPGDISPATCH	Public
228 9246	GetRecordsetClone	405	No	0	LPGDISPATCH	Public
229 9247	GetSection	405	Yes	1	LPGDISPATCH	Public
230 9248	GetForm	405	No	0	LPGDISPATCH	Public
231 9249	GetModule	405	No	0	LPGDISPATCH	Public
232 9250	GetProperties	405	No	0	LPGDISPATCH	Public
233 9251	Recalc	405	No	0	void	Public
234 9252	Requery	405	No	0	void	Public
235 9253	Refresh	405	No	0	void	Public
236 9254	Repaint	405	No	0	void	Public
237 9255	GoToPage	405	Yes	3	void	Public
238 9256	SetFocus	405	No	0	void	Public
239 9257	GetControls	405	No	0	LPGDISPATCH	Public

Figure6-3: Insertion of data in the `Member_Functions` table (in the red circle there is the number of the new records)

6.2.4 Step 4: Insertion of data in Function Parameters table

This step will confirm that the data, describing the parameters of the member functions of the class `_Form`, has been inserted successfully in the `Function_Parameters` table of the `PreprocessDB` database. As happened in

§6.2.3, due to the fact that the number of the parameters of the member functions of the class `_Form` is large, the parameters that belong to the function `GoToPage` are going to be examined thoroughly. The expected results are:

- 1) The `Function_Parameters` table should have 116 new records (that is, the total number of the parameters of the member functions of the class `_Form`).
- 2) The field `Function_Id` should have the value “9255” (it is the Id of the function “GoToPage” on the `Member_Functions` table) in three consecutive records.
- 3) In the same three consecutive records, the field `Param_Name` should have the values “PageNumber”, “Right”, “Down”.
- 4) In the same three consecutive records, the field `ParamType` should have the value “long”.
- 5) In the same three consecutive records, the field `ParamUse` should have the value “By Value”.

The following picture (6-4) illustrates the successful insertion of the data, which describe the parameters of the member functions of the class `Form`, in the `Function_Parameters` table of the `PreprocessingDB`:

SQL Query Analyzer - [Query - S6_08.PreprocessingDB.S6_08Yiannis Kanellopoulo - Untitled1*]

Select * from Function_Parameters

	ParamId	FunctionId	ParamName	ParamType	ParamUse
93	6981	9203	lpszNewValue	LPCTSTR	By Value
94	6982	9205	lpszNewValue	LPCTSTR	By Value
95	6983	9207	nNewValue	long	By Value
96	6984	9209	bNewValue	BOOL	By Value
97	6985	9211	nNewValue	short	By Value
98	6986	9213	nNewValue	short	By Value
99	6987	9215	nNewValue	short	By Value
100	6988	9217	nNewValue	short	By Value
101	6989	9219	nNewValue	short	By Value
102	6990	9221	nNewValue	long	By Value
103	6991	9223	nNewValue	long	By Value
104	6992	9225	nNewValue	long	By Value
105	6993	9227	nNewValue	long	By Value
106	6994	9229	nNewValue	long	By Value
107	6995	9231	nNewValue	const VARIANT	By Reference
108	6996	9233	nNewValue	long	By Value
109	6997	9235	nNewValue	long	By Value
110	6998	9237	nNewValue	const VARIANT	By Reference
111	6999	9239	bNewValue	BOOL	By Value
112	7000	9245	ControlType	long	By Value
113	7001	9247	var	const VARIANT	By Reference
114	7002	9255	PageNumber	long	By Value
115	7003	9255	Right	long	By Value
116	7004	9255	Down	long	By Value

Grid #1: 116 rows | Ln 114, Col 1

Figure6-4: Insertion of data in the `Function_Parameters` table (in the red circle there is the number of the new records)

7. Results – Evaluation chapter

7.1 Introduction

The evaluation is the examination of the of the Preprocessing Application output's accuracy. This output should be:

- Valid, which means that it has to represent the actual model of the system under examination.
- Useful to the potential maintainer of the system.
- Novel, which means that it has to be something new and not known before.

Two applications, the `CAccessReport` and `CompDB`, are used as samples. Their actual structure will be compared to the outcome of the analysis of their respective input models. At this point, it has to be clarified that the term analysis indicates the application of clustering in the input model (data that is stored in the `PreprocessingDB` database). IBM's Intelligent Miner, was used for this purpose.

Another point that has to be emphasised is that both applications, `CAccessReports` and `CompDB`, are created with the help of MFC (Microsoft Foundation Classes). This means that they can automatically generate most of their own windows, handle their own messages and do their own drawings. MFC is a system of C++ classes designed to simplify Windows programming. It consists of a multi-layered class hierarchy that defines approximately 200 classes, which allow programmers to construct a Windows application using object-oriented principles. In other words, MFC provides the programmers with a framework upon which they can build Windows applications [Schild 1998].

7.2 Evaluation of the Preprocessing Application

7.2.1 Description and General Characteristics of the CAccessReport system

CAccessReport is an application, which was created in order to help users, who:

- Use the MSACCESS database engine
- Use the MSACCESS database engine from a Visual C++ application

It uses the Access automation objects in order to open a specified database, run a report (within Access), print it and save the database in HTML format. Such process is carried out for the database to be viewed in an application that is written in Visual C++ [http://www.codeguru.com/mfc_database/access_reports_class.shtml].

CAccessReports, is developed by Tom Archer who is an author and a Visual C++/C# consultant [<http://www.codeproject.com/interview/tomarcher3jun2001.asp>]. The programming language that was used is Visual C++ 6.

It is a medium-size application and includes 53 public classes and 2812 functions that have 1614 parameters. It is not a data-driven application as it has only five data members. It is an automation client as the vast majority of the classes (52 out of 53) are a sub class of COleDispatchDriver, with the latter to implement the client side of OLE automation. An automation client is an application that can manipulate exposed objects belonging to another application. It manipulates them by accessing those objects' properties and functions [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_Automation_Clients.asp].

7.2.1.1 CAccessReport: Classes Analysis

The classes of CAccessReport application have many similarities as:

- Almost all of them inherit from COleDispatchDriver.
- Almost all of them (52 out of 53) have only public member functions.

Therefore, during the clustering process only the fields describing the number of the public functions and the handle of the class (as a supplementary field) are the only factors that can play an important role to the creation of the clusters. After the clustering process the following three clusters were derived:

1. As presented in picture 7-1, the first cluster, which represents the 54.72% of the population, consists of small classes that include a range of 3 to 27 public member functions.

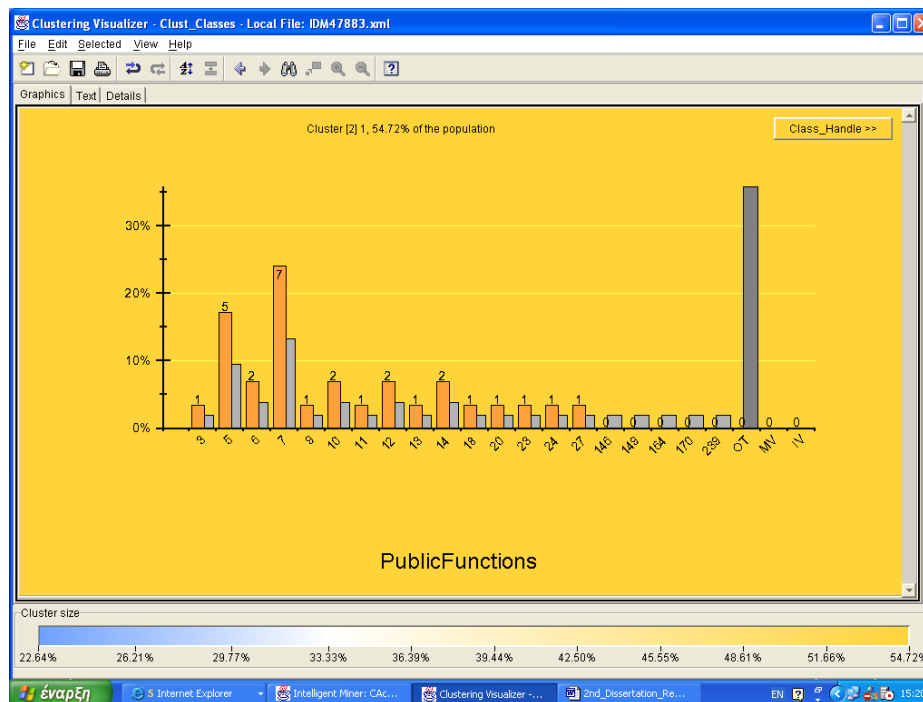


Figure7-1: CAccessReport: First cluster of the `CClasses` table (small size classes).

2. The second cluster, which represents the 22.64% of the population, consists of medium size classes that include a range from 35 to 87 public member functions. This is illustrated in picture 7-2.

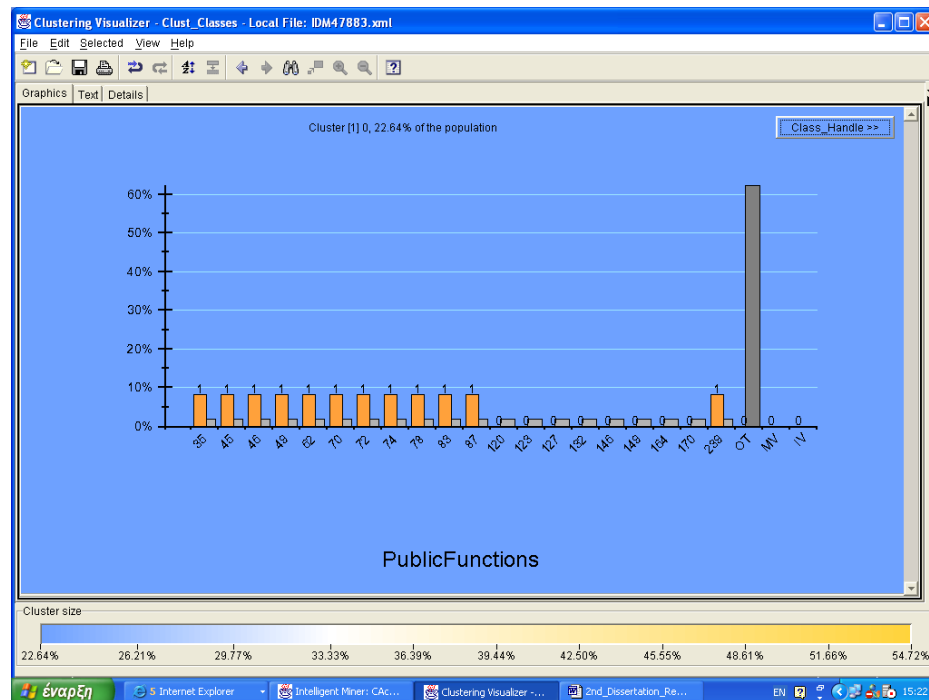


Figure7-2: CAccessReport: Second cluster of the `Classes` table (medium size classes).

3. The third cluster (picture 7-3), which also represents the remaining 22.64 of the population, consists of medium to large size classes, including a range from 100 to 170 public member functions.

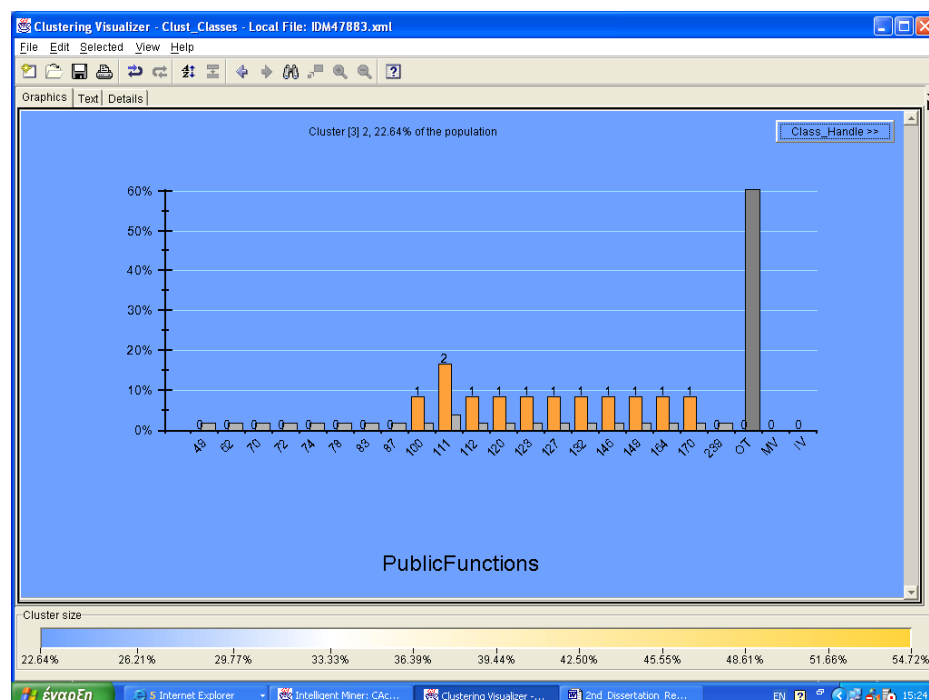


Figure7-3: CAccessReport: Third cluster of the `Classes` table (large size classes).

7.2.1.2 CAccessReport: Member Functions Analysis

There are two significant characteristics of the member functions of CAccessReport; the first is that all of them are public and the second is that almost half of them contain parameters (48.06% of the population). They can be divided in the following three clusters:

1. As illustrated in pictures 7-4 and 7-5, the first cluster, which represents the 45.82% of the population, consists of public functions that have parameters, and that they have no return type or they return void. Therefore, it can be concluded that this cluster includes the constructors of the system's classes and functions that usually set values to the classes.

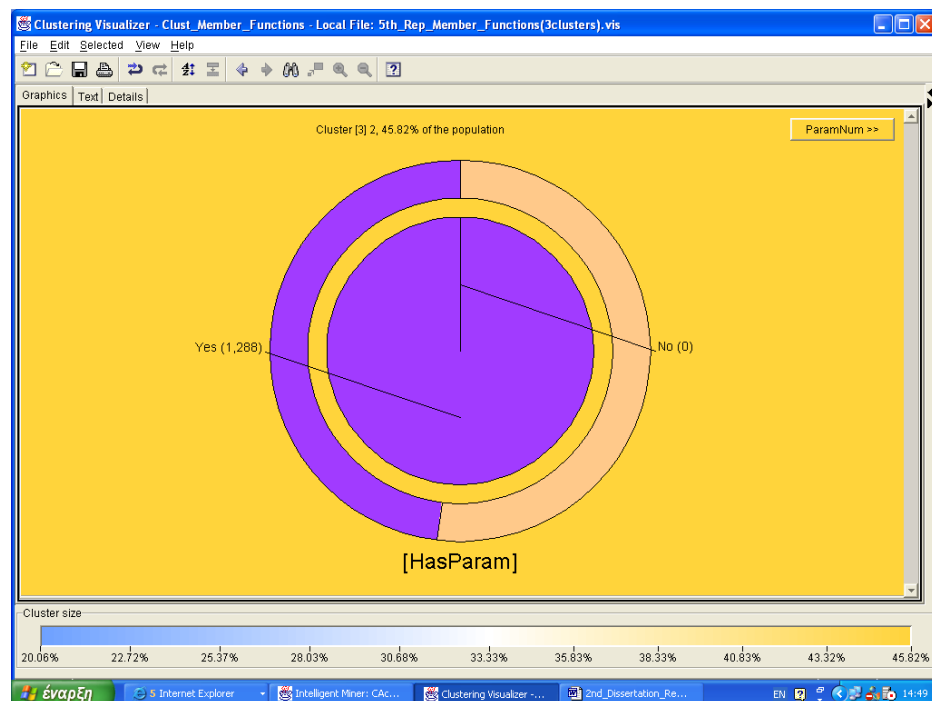


Figure7-4: CAccessReport: First cluster of the Member_Functions table. Public functions with parameters

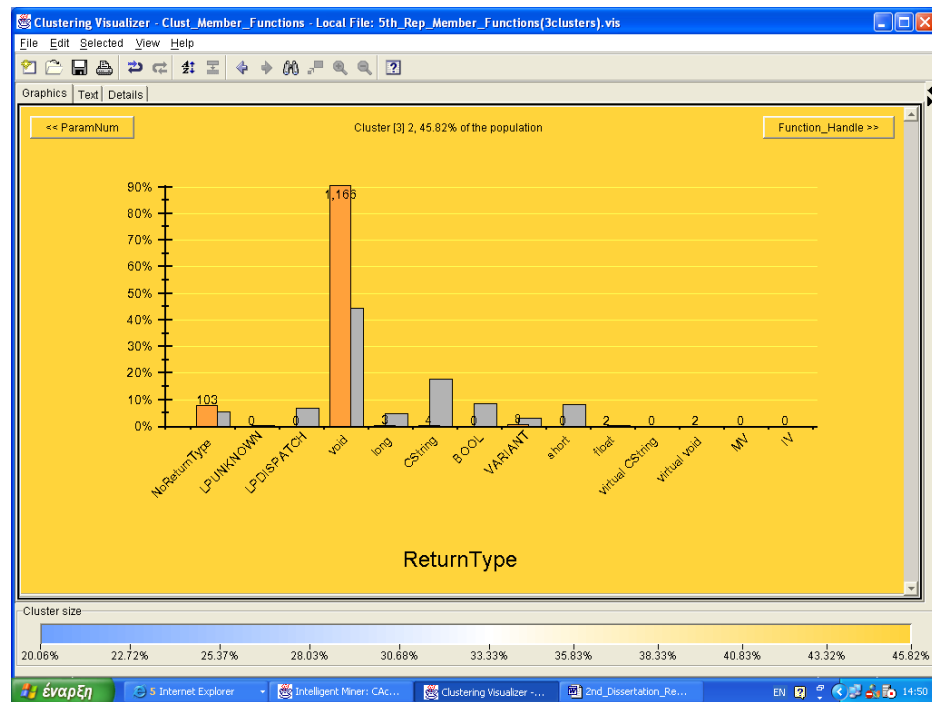


Figure7-5: CAccessReport: First cluster of the Member_Functions table. Public functions that return void or null

2. The second cluster as shown in pictures 7-6 and 7-7 represents the 34.12% of the population and consists of public functions that have no parameters. Half of this cluster's functions return the type `CString`, which encapsulates a character string. It is a class that provides basic string operations such as concatenation, comparison and assignments [Shepherd and Wingo 1996].

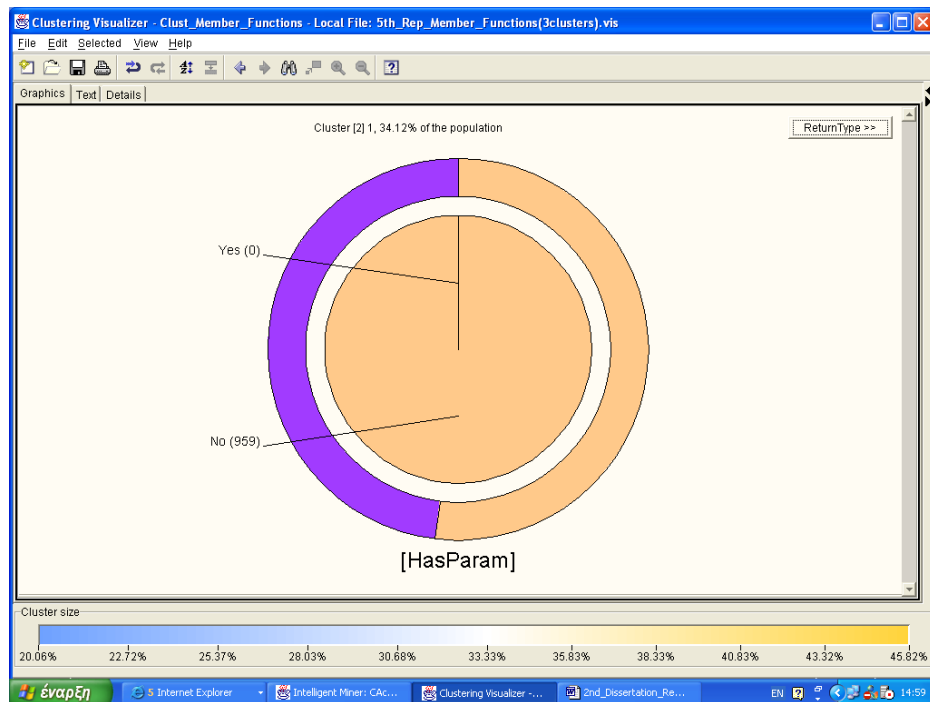


Figure7-6: CAccessReport: Second cluster of the `Member_Functions` table. Public functions with no parameters

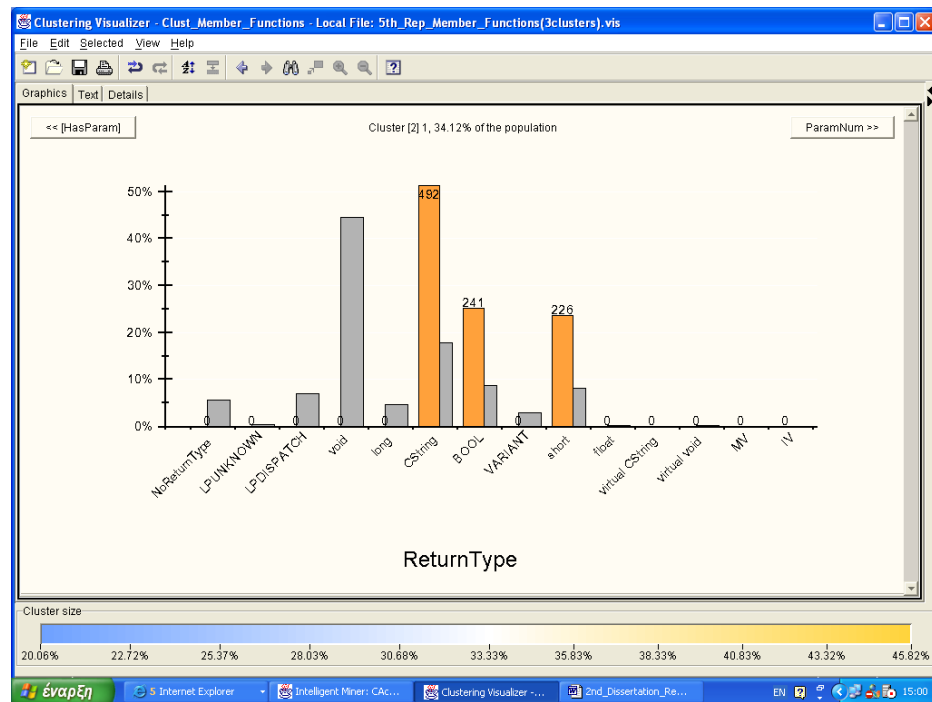


Figure7-7: CAccessReport: Second cluster of the `Member_Functions` table. Public functions that return `CString`

3. As demonstrated in pictures 7-8 and 7-9, the third cluster that represents the 20.06% of the population, consists of public functions in which a rate of 11.17% have no parameters at all, when the remaining 88.83% have. Almost half of this cluster's functions return the following types:

- i) **VARIANT**, which is a self-described data type. The first field of the **VARIANT** structure is an unsigned short representing the type of data which is contained in the structure. By using this data type, passing of data becomes fairly simple since all the parameters are homogenous [Shepherd and Wingo 1996].
- ii) **LPDISPATCH**, which accesses the underlying **IDISPATCH** pointer of the **coledispatchDriver** object [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/vclrfcoledispatchdriveroperatorlpdispatch.asp].

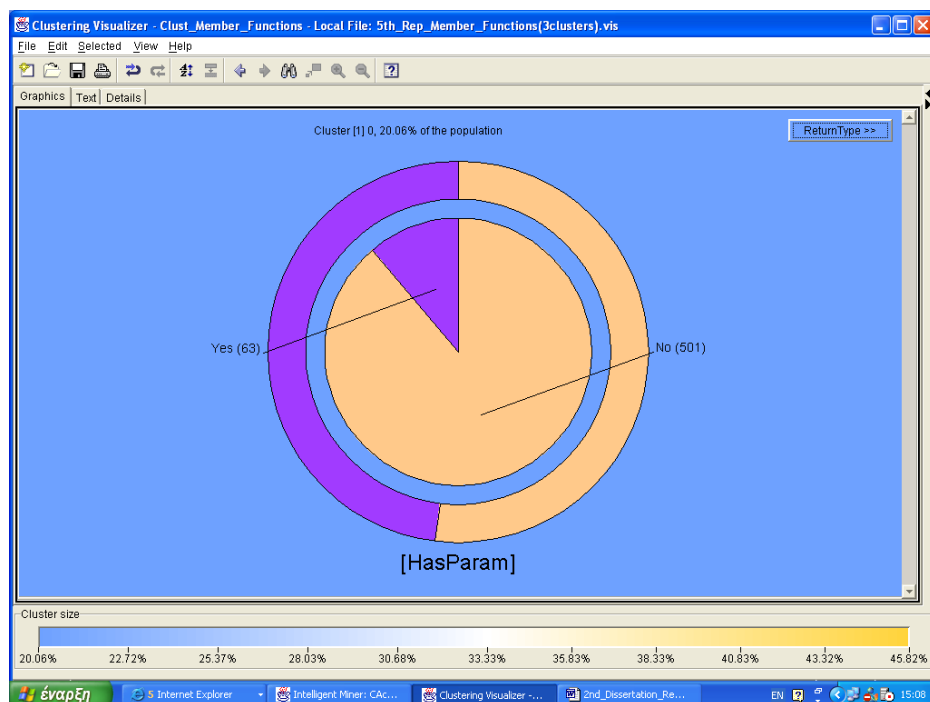


Figure7-8: CAccessReport: Third cluster of the **Member_Functions** table. Public functions with parameters and no parameters

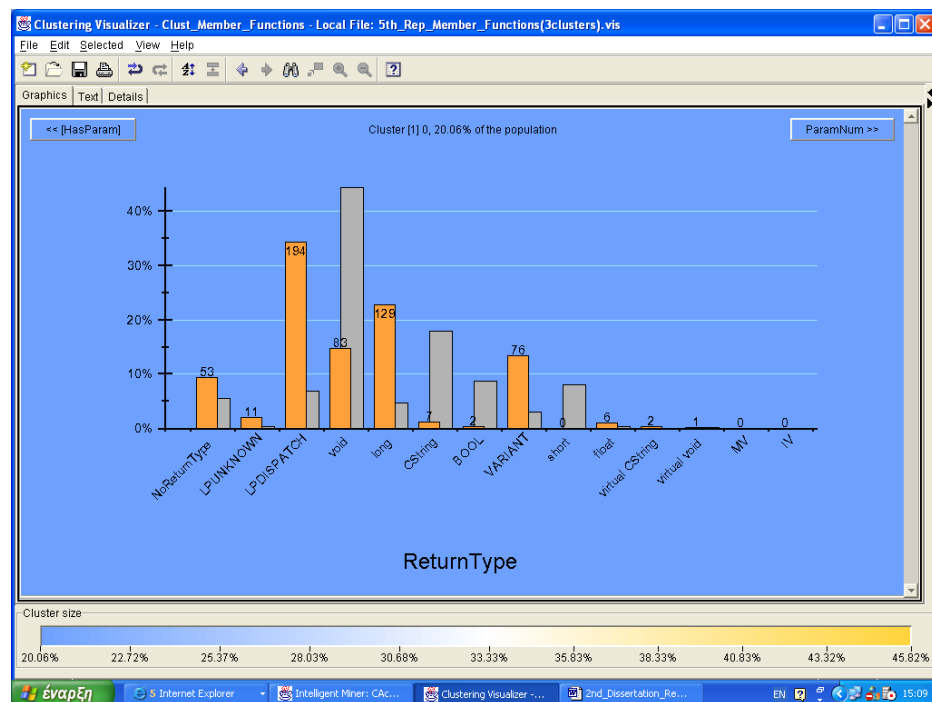


Figure7-9: CAccessReport: Third cluster of the Member_Functions table. Public functions that return VARIANT and LPDISPATCH

7.2.1.3 CAccessReport: Analysis of the parameters of member functions

Most of the member functions' parameters in CAccessReport are passed by value (1256 out of 1614). They can be divided into three clusters:

1. According to the pictures 7-10 and 7-11, the first cluster (42.44% of the population) consists of parameters that are passed by value and originate from the following types:
 - i) LPCSTR, which is a Windows data type. It is a constant pointer to a string [Schild 1998].
 - ii) LPDISPATCH.
 - iii) pointers of type VARIANT.

From the parameters' types mentioned above, it can be concluded that the functions making use of these parameters can manipulate strings and access ActiveX clients.

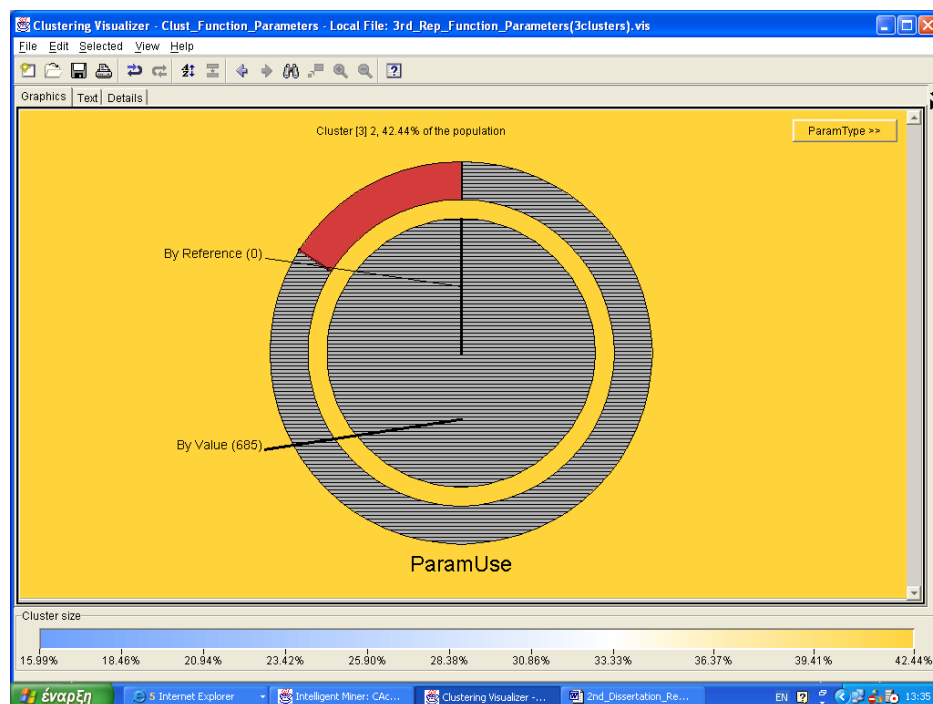


Figure7-10: CAccessReport: First cluster of the Function_Parameters table. Parameters by value

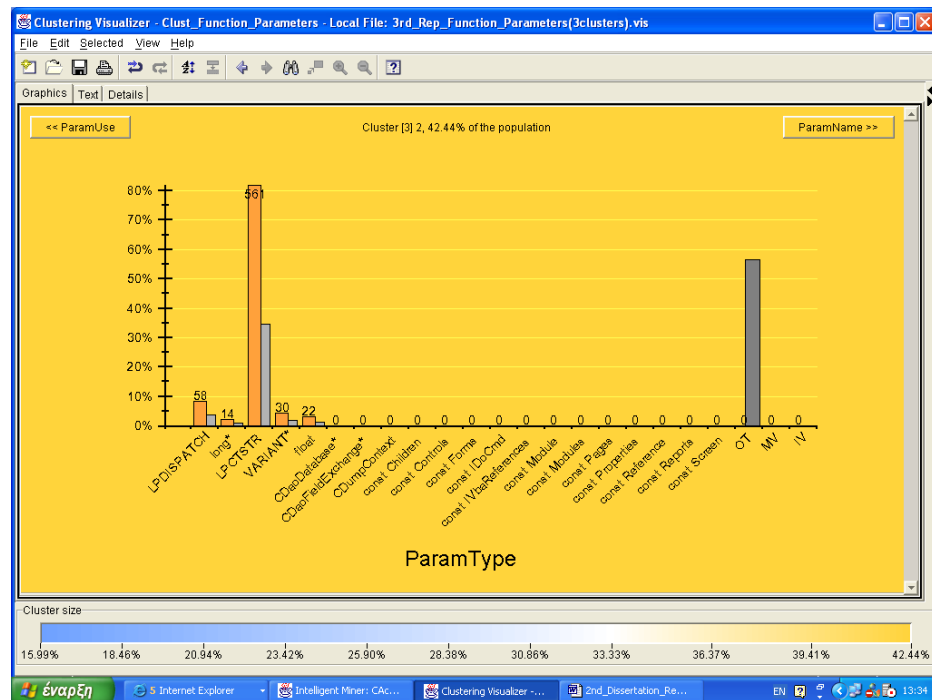


Figure7-11: CAccessReport: First cluster of the `Function_Parameters` table. Parameters of type LPCSTR, LPDISPATCH and VARIANT

2. The second cluster (41.57%) also consists of parameters that are passed by value, most of which originate from the following types (pictures 7-12, 7-13):
 - a. BOOL
 - b. short
 - c. long

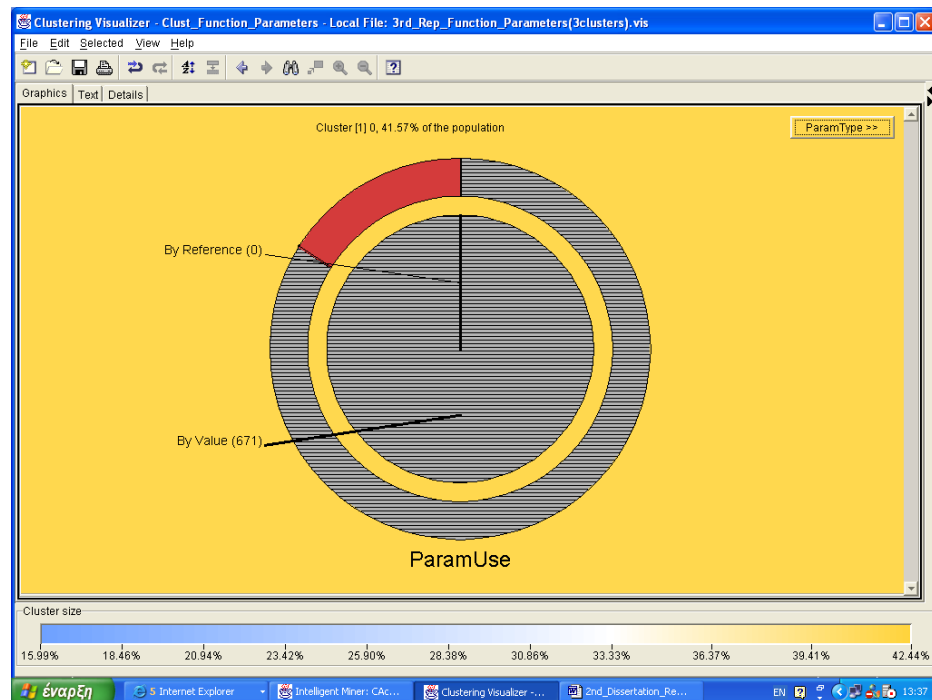


Figure7-12: CAccessReport: Second cluster of the Function_Parameters table. Parameters by value

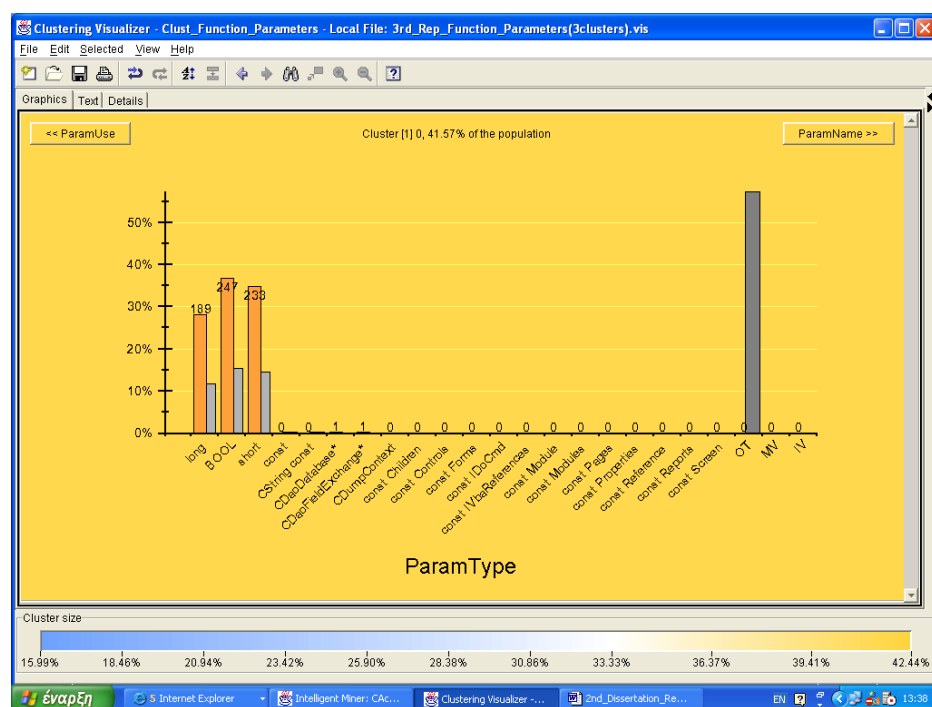


Figure7-13: CAccessReport: Second cluster of the Function_Parameters table. Parameters of type BOOL, short and long

3. As seen in pictures 7-14 and 7-15, the third cluster (15.99%) consists of parameters that are passed by reference and they are constants of type VARIANT (77.91%). This means that this cluster's parameters are based on the operator `IDispatch::Invoke`, which provides a way of accessing and exposing objects within a particular application. They are used by functions which aim to specify data that cannot be passed by reference in any other way.

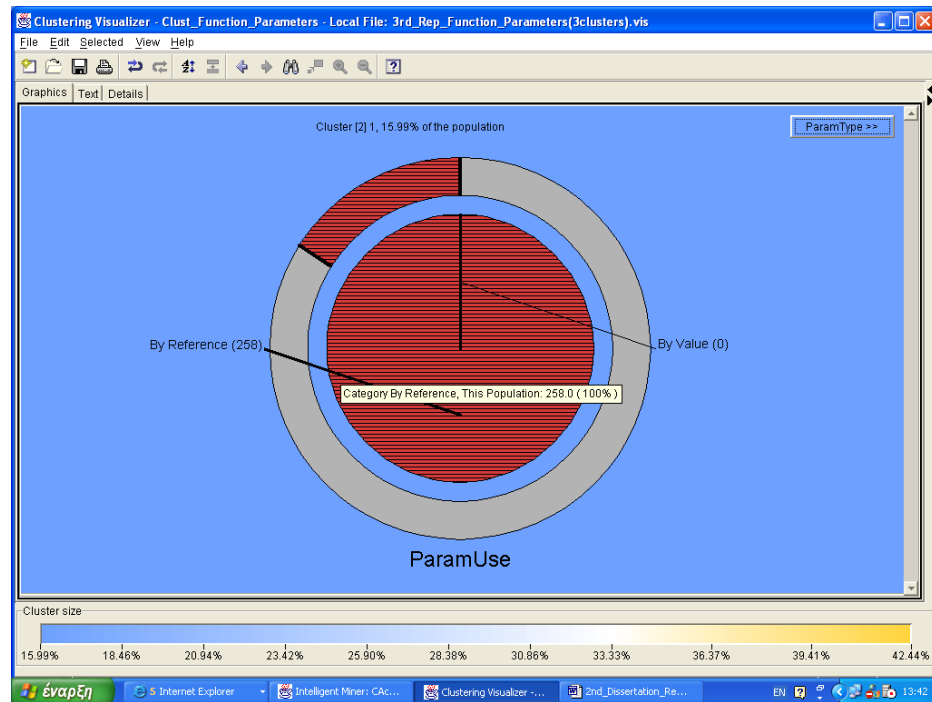


Figure7-14: CAccessReport: Third cluster of the `Function_Parameters` table. Parameters by reference

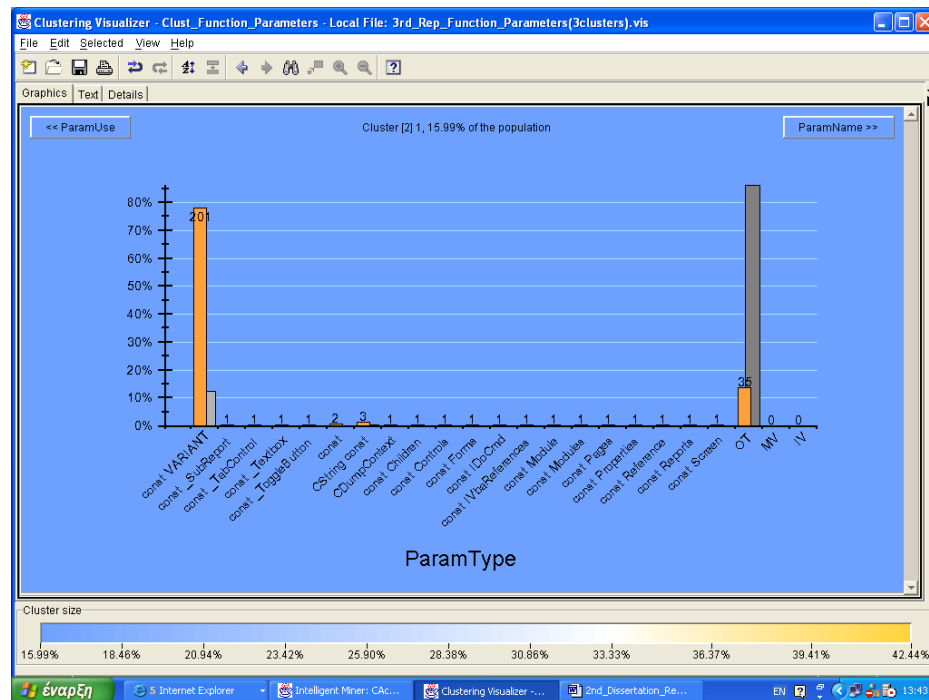


Figure7-15: CAccessReport: Second cluster of the `Function_Parameters` table. Parameters of type `VARIANT`

7.2.1.4 CAccessReport: Conclusions

`CAccessReport` does not show important differences between its components. The majority of its classes (52 out of 53) inherits from the same superclass (`COleDispatchDriver`), and has no member data. All of their member functions also are public. The differences between the `CAccessReport`'s components are mainly quantitative, like for example the number of public functions or parameters. The return types of the member functions and their parameters' types are the only differences that are not quantitative but qualitative are.

A valid point is that every parameter passed by reference is a constant of the type `VARIANT`. Thus, it can be derived that the functions that use these parameters are binding during run-time and the data is not known ahead of time.

7.2.2 Description and General Characteristics of the CompDB system

CompDB application is used in order to compare recordsets in a database. A common problem that programmers face, is the customisation of a new version of an application in order to process and change data that already resides in a complex table structure of a database. The main problem is the comparison and the verification of the results of this new version with the respective results of the old version. CompDB application helps the user to define a set of queries having parameters, and export the results in a file. Next, the user can compare the two files (the one from the old version and the other from the new one), by using the program “winDiff” [http://www.codeguru.com/mfc_database/CompDB.html].

CompDB system is a small-size application. It consists of 18 public classes, 64 member data and 256 functions that have 235 parameters. It is based on the document/view architecture, which is one of the cornerstones of MFC. This architecture provides a single, consistent way of coordinating application data and rendering of that data. The document handles data management and the views handle the user-interface management. In effect, an application’s data is centralised in one place, and the user-interface code is packaged separately [Shepherd and Wingo 1996].

This separation between the application data (documents) and representations of that data (views) is the main advantage of the document/view architecture. This architecture consists of the following components:

- Document, which is the core of the document/view architecture. It is not implying data in the form of things like word processors or spreadsheets. It is a place to keep data, a kind of data source [Shepherd and Wingo 1996].
- View, which is responsible for rendering a document’s data. It also provides a user interface for manipulating the data [Shepherd and Wingo 1996].
- Document/View Frame, which provides the ability to apply a different user interface (that is, a different set of menus and controls) to each separate view. Single Document Interface (SDI) applications use a class derived from `CFrameWnd` as the frame is housing one view. On the other hand Multiple Document Interface (MDI) applications derive a class from `CMDIChildWnd` as the frame is housing multiple views of the document [Shepherd and Wingo 1996].

- Document Template, which ties the three previous components together. More specifically the ideas of a document, its renderings, and its user interface are treated together as a whole. These three components are managed by a class called `CDocTemplate` [Shepherd and Wingo 1996].

7.2.2.1 CompDB: Classes Analysis

After the clustering process of the CompDB application's Input Model, the following clusters were derived:

1. The first cluster, as depicted in picture 7-16, represents the 38.89% of the population, and consists of classes that all of them inherit. Their respective superclasses are:
 - i) `CStatic`, which encapsulates the static control. It is considered to be a classic control as it has been in Windows since the beginning [Shepherd and Wingo 1996].
 - ii) `CView`, which is the class that a `view` class is derived from. A view, in general, is a physical representation of the data. A view class that is derived from `CView`, controls the display of the document. The view window overlays the frame window and relies on the frame window for the basic window functions, such as resizing and minimising [Schild 1998].
 - iii) `CMDIFramewnd`, which provides a main, frame window for Multiple Document Interface (MDI) applications [Shepherd and Wingo 1996].
 - iv) `CMDIChildwnd`, which provides child windows for an MDI application [Shepherd and Wingo 1996].

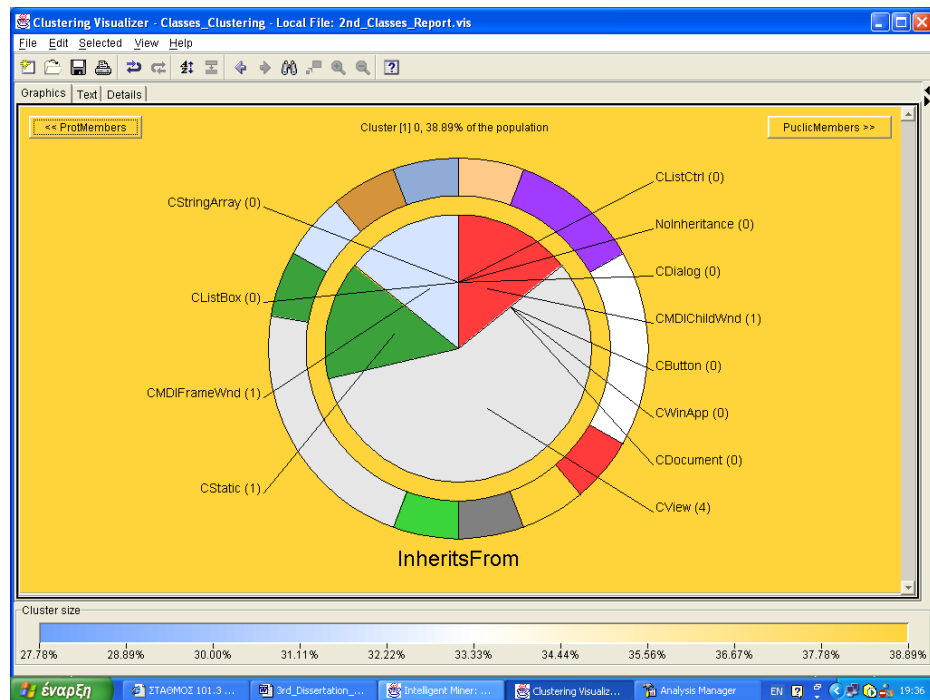


Figure 7-16: CompDB: First Cluster **Classes** table. Information about the super classes

As presented above the classes belonging to this cluster, are related logically, as they represent components of the document/view architecture. There are four classes (CQueryView, CPropView, CParaView, CCompDBView) that derive from the CView class and two classes that provide the respective frames and which derive from CMDIFrameWnd (CMainFrame) and CMDIChildWnd (CChildFrame).

In addition what characterises the classes that are members of this cluster is that the values representing the amount of public functions is distributed between 5 and 10, but the ones representing the amount of the protected functions are either too small or too big. This can be seen in pictures A-1, A-2 of §Appendix.

2. According to the picture 7-17, the second cluster, represents the 33.33% of the population, and consists of classes, amongst which, two of them do not inherit and four of them do. The respective superclasses of those who inherit are:
 - i) **CStringArray**, which is an array of the **String** type. Its main task is to handle the memory allocation [Shepherd and Wingo 1996].

- ii) `CListBox`, which encapsulates the list box control. It is a classic control since it has been in Windows from the beginning [Shepherd and Wingo 1996]. It is similar to the `CStatic` class.
- iii) `CDocument`, which is the class where the document of an MFC application (like `CompDB`) derives from [Shepherd and Wingo 1996].
- iv) `CListCtrl`, which displays a graphical list (Explorer-like) of list items [Shepherd and Wingo 1996].

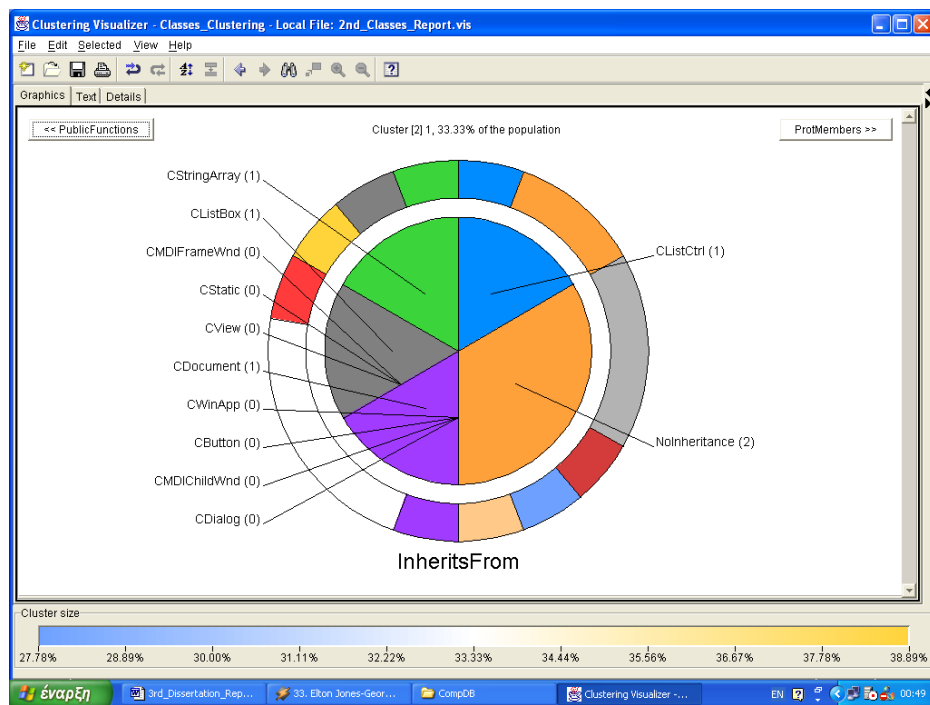


Figure7-17: CompDB: Second cluster of `Classes` Table. Information about the super classes

As portrayed in the above picture, the classes belonging to this cluster do not have the same logical correlation. There is only one class representing a component included in the document/view architecture (`CCompDBDoc`), two others represent control classes (`CPropertyListCtrl`, `CCrack`) and another one that represents a shape of the MFC collection (`CSortStringArray`).

In addition what characterises the classes that are members of this cluster is that the values that represent the number of public and the protected functions are either too small or too big. This is portrayed in pictures A-3, A-4 of §Appendix.

3. According to the picture 7-18 the third cluster represents the 27.78% of the population, and consists of classes that all of them inherit. Their respective superclasses are:

- i) `CDialog`, which implements Windows dialogs. It supports both modal and modeless dialog creation and operations. Modal dialogs “freeze” the rest of the application and force the user to perform some operation before it will go away. On the other hand, modeless dialogs let the user continue using the application while it is displayed [Shepherd and Wingo 1996].
- ii) `CButton`, which is a standard Windows pushbutton [Shepherd and Wingo 1996].
- iii) `CWinApp`, which represents the standard Windows Application. It has overrideable functions that can be used in the initialisation and the termination cleanup of the application [Shepherd and Wingo 1996].

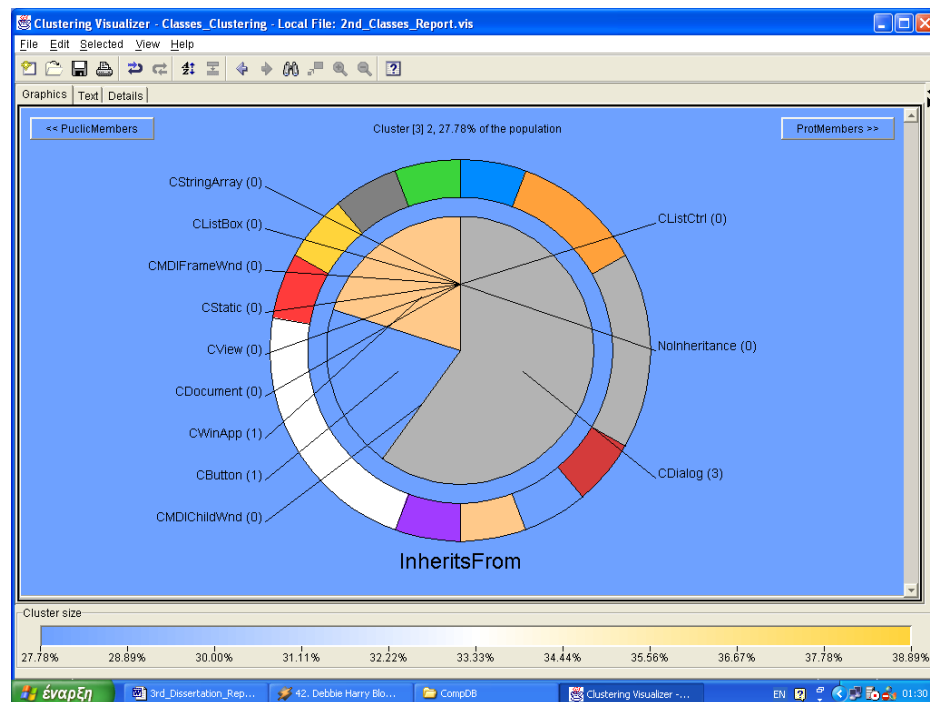


Figure7-18: CompDB: Third cluster of `Classes` Table. Information that describes the super classes

As outlined above, not all classes belonging to this cluster have the same logical relation. There are three classes representing the dialog controls (`CArgDlg`,

CCompDlg, CQueryDlg), one representing the Windows Application (CCompDBApp) and another one that represents a control (CComboButton).

An additional characteristic of this cluster is that the values representing the public functions' number are distributed between 1 and 5 and the values that represent the number of the protected functions are between 2 and 7. As illustrated in pictures A-5, A-6 of §Appendix there are no significant differences.

7.2.2.2 CompDB: Member Data Analysis

The member data of the CompDB application classes are either public or protected. The clusters drawn after the clustering process are:

1. The first cluster, as portrayed in pictures A-7, A-8 of §Appendix, represents the 59.38% of the population and consists of protected members, where none of which is a pointer. The member data of this cluster belongs to the majority of the CompDB application classes (picture 7-19). However, almost half of them belongs to classes CCompDBView (with ID = 427) and CAdoList (with ID = 420). The types of the member data vary. The more predominant, as showing in picture 7-20, are:

- i) `int`
- ii) `CString`, which encapsulates a character string [Shepherd and Wingo 1996].
- iii) `CFont`, which wraps the Windows font object (an `HFONT`) and the API functions for creating and managing fonts [Shepherd and Wingo 1996].
- iv) `CGridCtrl`, which is a control derived from the `CWnd` class.

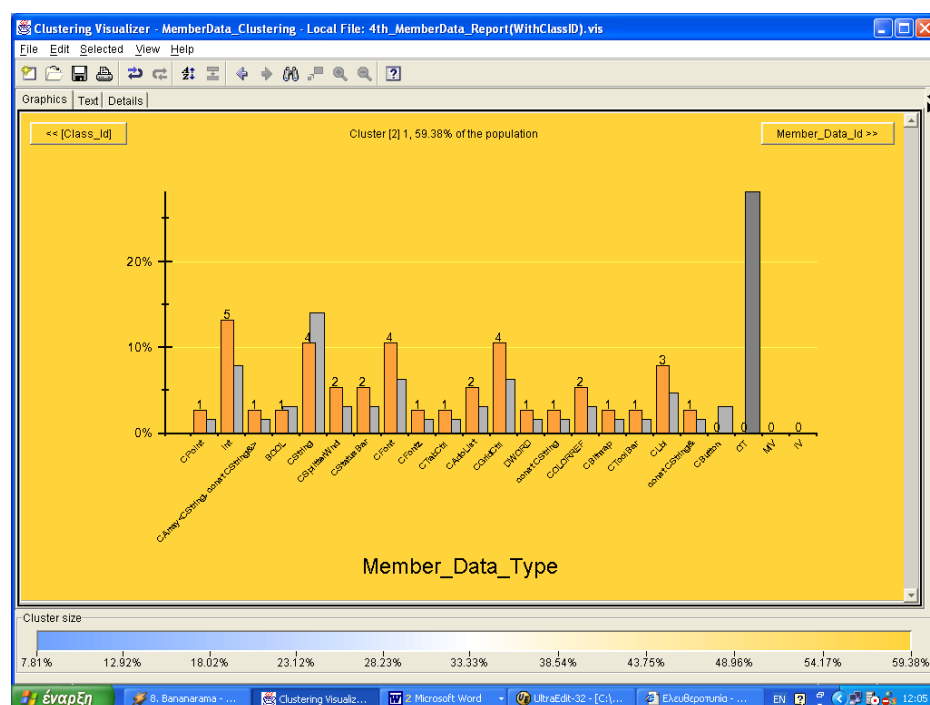
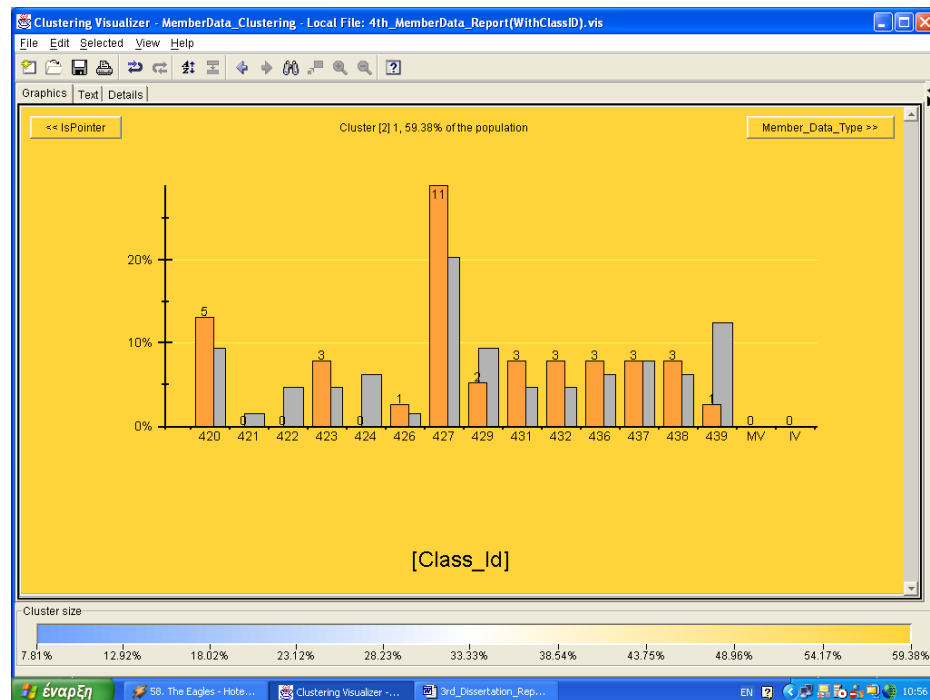


Figure7-20: CompDB: First cluster of Membe_Data Table. Types of Member Data

2. The second cluster, as depicted in pictures A-9, A-10 of §Appendix, represents the 32.81% of the population and consists of public members that none of them is a pointer. Member data of this cluster belong to half of

CompDB application classes (picture 7-21). In particular, they belong to classes CQueryDlg (with ID = 439), CCompDlg (with ID = 429) and CArgDlg (with ID = 422). It is clear that there is a logical connection between the member data of this cluster as the majority of it belonging to classes that are derived from the CDiallog class.

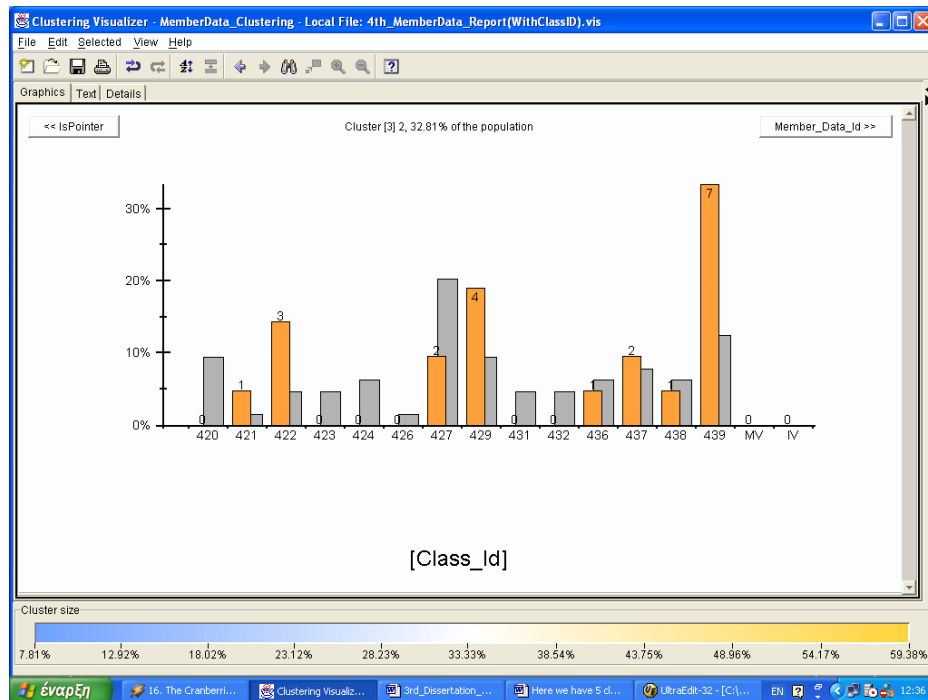


Figure7-21: Distribution of the member data of the second cluster among the classes of the CompDB application

The types of the member data vary. The more predominant, as depicted in the picture 7-22, are:

- i) enum
- ii) CString, which encapsulates a character string [Shepherd and Wingo 1996].
- iii) CButton, which wraps a standard Windows pushbutton [Shepherd and Wingo 1996].

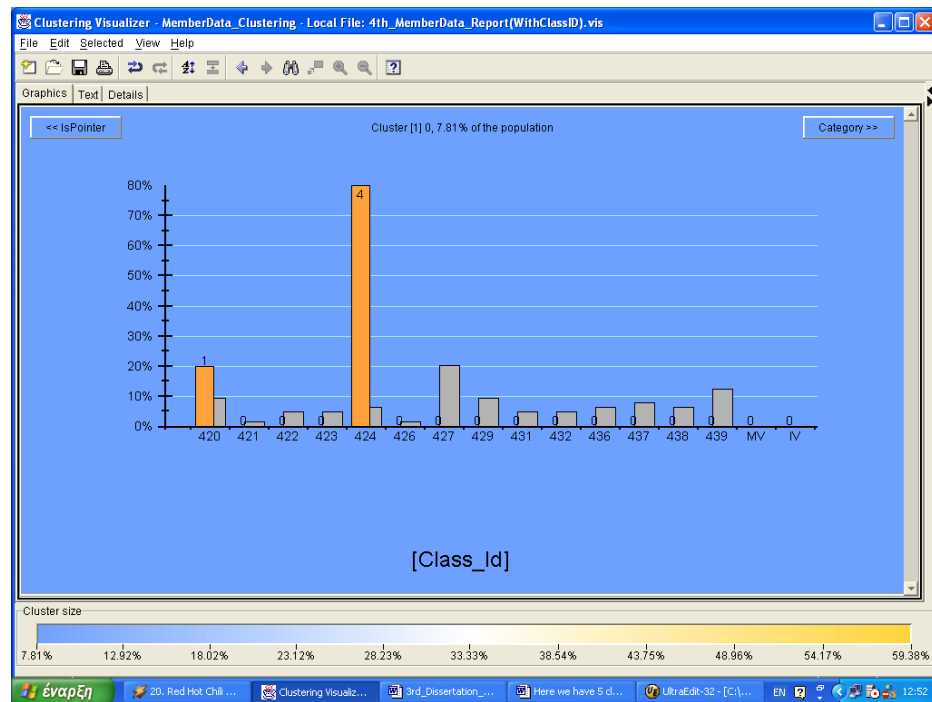


Figure7-23: CompDB: Third cluster of **Member_Data**. Distribution of the member data among the classes of the CompDB application

The types of the member are different. The more predominant, as shown in picture 7-24, are:

- i) **CPen**, which wraps the Windows pen object (an **HPEN**) and includes the API functions for creating pens as member functions [Shepherd and Wingo 1996].
- ii) **CBrush**, which wraps the Windows brush object (an **HBRUSH**) and the API functions for creating brushes [Shepherd and Wingo 1996].

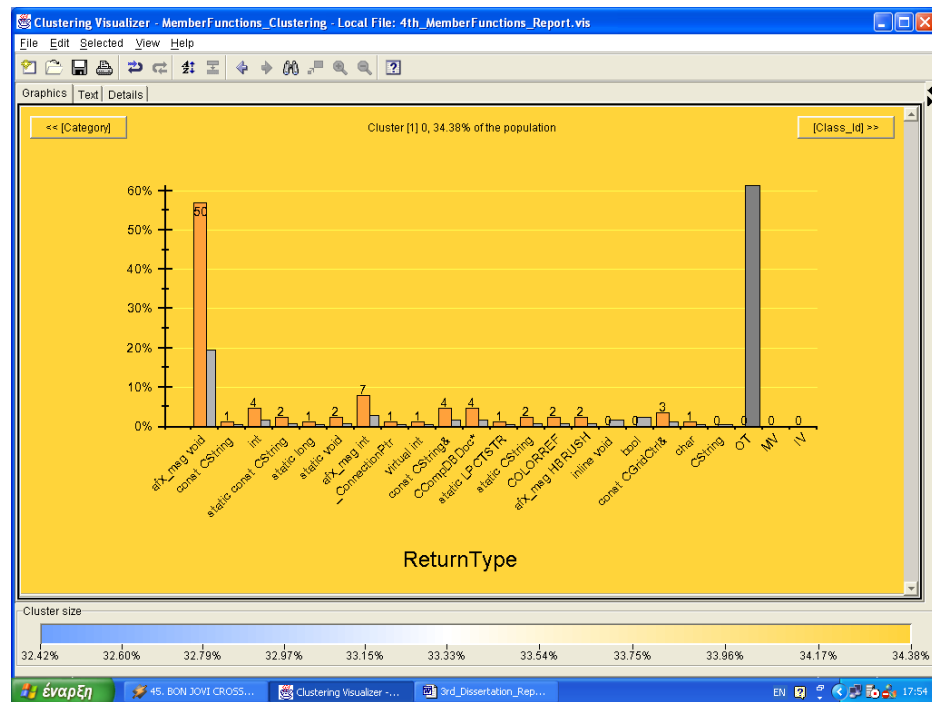


Figure7-25: CompDB: First cluster of Member_Functions table. Return Types of the member functions

An additional characteristic of the member functions of this cluster is that they belong to the majority of the **CompDB** application classes (picture 7-26). However, most of them belong to classes **CChildFrame** (with ID = 423), **CPropertyListCtrl** (with ID = 434), **CParamView** (with ID = 436), **CPropView**, (with ID = 437) and **CQueryView** (with ID = 438).

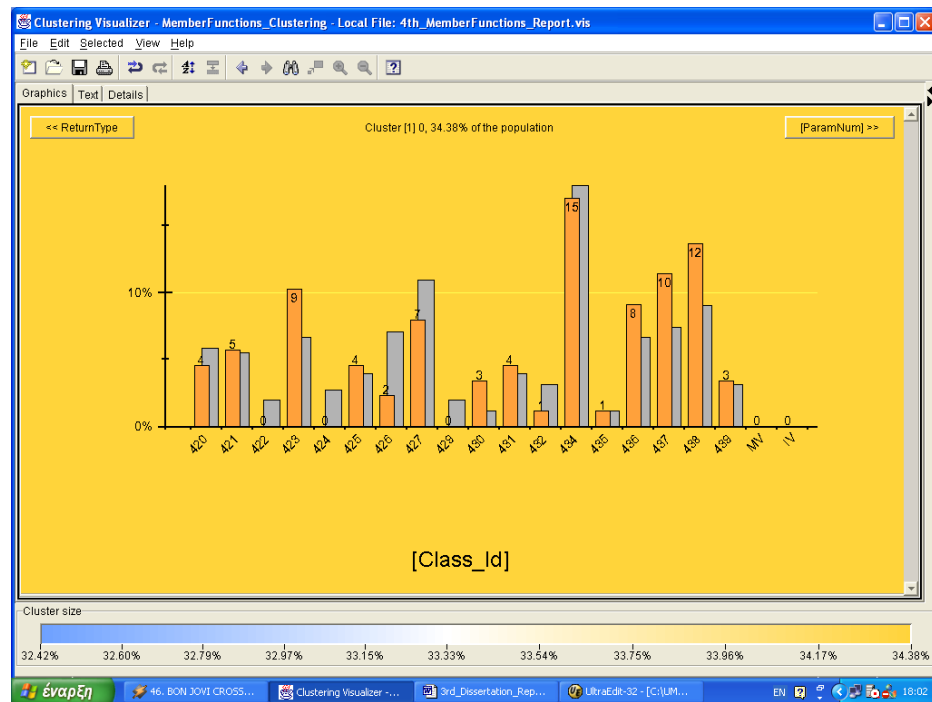


Figure7-26: CompDB: First cluster of `Member_Functions` table. Distribution of the member functions among the classes of the CompDB application

The final common characteristic of the member functions of this cluster is that the majority of them have zero or one parameters. This is depicted in picture A-14 of §Appendix.

2. The second cluster, as presented in picture A-15 of §Appendix, represents the 33.20% of the population and includes all the categories of the member functions (public, protected and private). The return types of the member functions of the `CompDB` application classes vary, with the most predominant (according to picture 7-27) being:
 - i) `void`
 - ii) `BOOL`

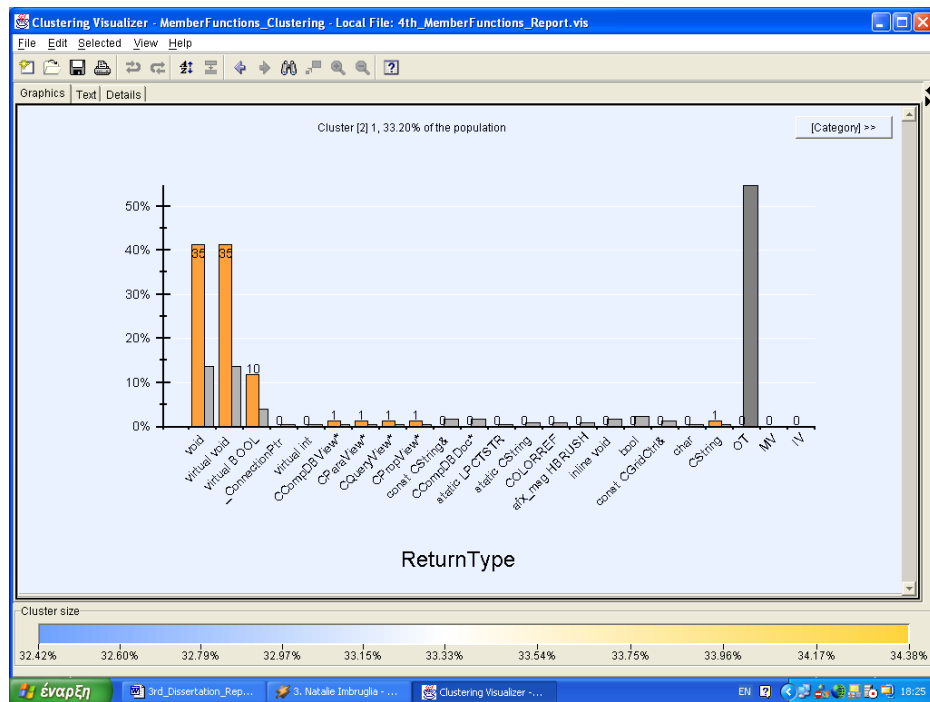


Figure7-27: CompDB: Second cluster of Member_Functions table. Return Types of the member functions

In addition the most significant member functions feature in this cluster is that they belong to the majority of CompDB application classes (picture 7-28). However, most of them belong to classes CCompDBDoc (with ID = 426), CCompDBView (with ID = 427), CPropertyListCtrl (with ID = 434) and CParaview (with ID = 436).

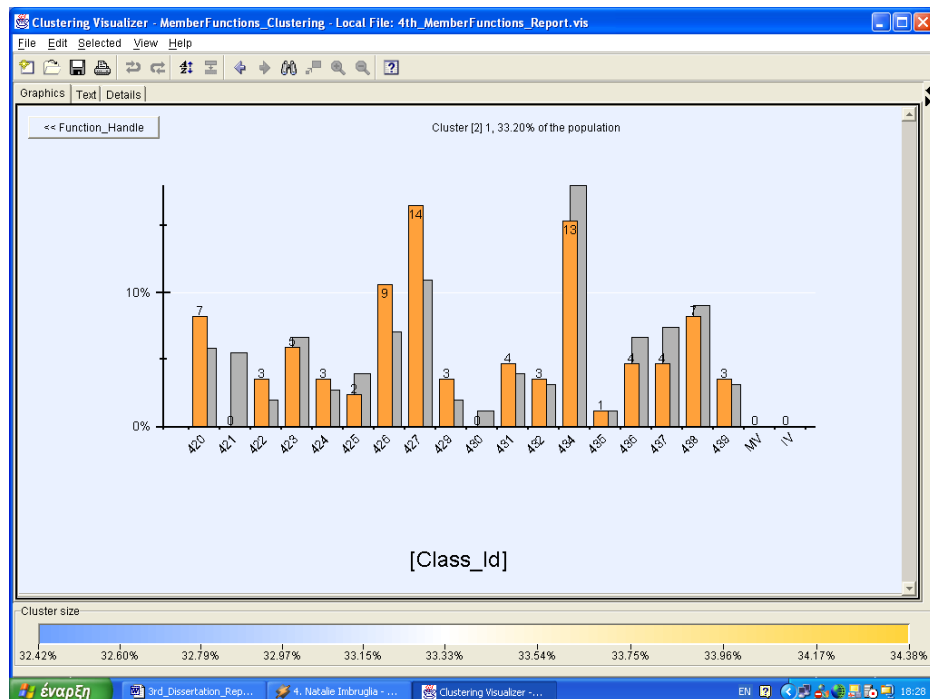


Figure7-28: Distribution of the member functions of the first cluster among their classes

The final common characteristic of the member functions of this cluster is that the majority of them have zero or one parameters. This is depicted in the picture A-16 of §Appendix.

3. The third cluster, as demonstrated in picture A-17 of §Appendix, represents the 32.42% of the population and consists of all the categories of the member functions (public, protected and private). Almost half of the functions of this cluster do not have a return type. This indicates that they are either the constructors or the destructors of the classes where they belong. Regarding member functions that have a return type, the most predominant is the type **BOOL**. This can be depicted in the picture 7-29:

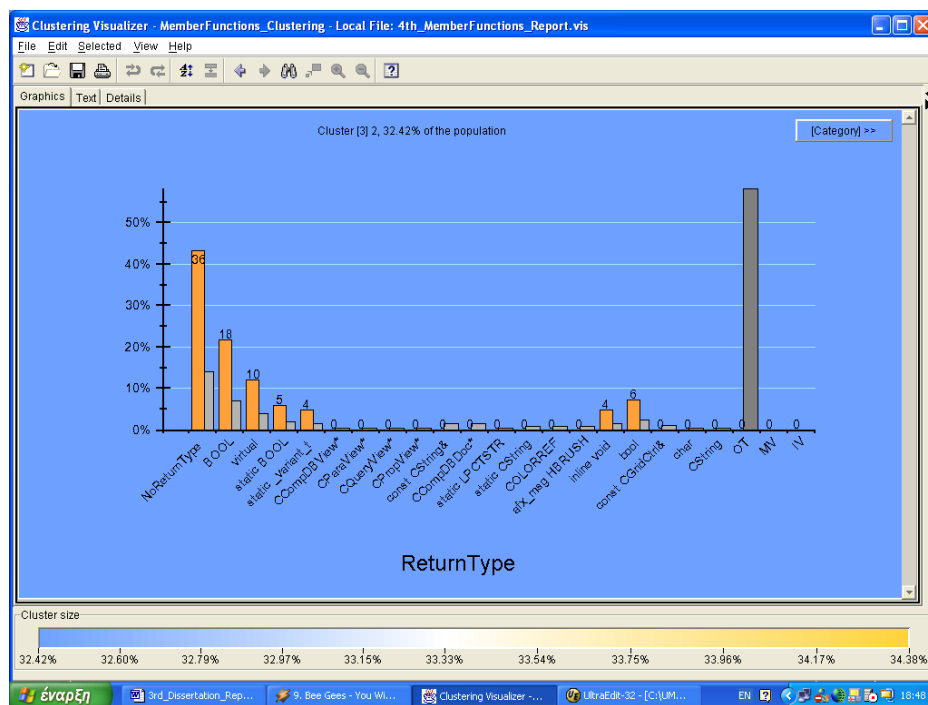


Figure7-29: CompDB: Third cluster of `Member_Functions` table. Return Types of the member functions

An additional feature of the member functions of this cluster is that they belong to the majority of the classes of the `CompDB` application (picture 7-30). However, most of them belong to the classes `CCompDBDoc` (with ID = 426), `CCompDBView` (with ID = 427), `CPropertyListCtrl` (with ID = 434) and `CAdouti` (with ID = 421).

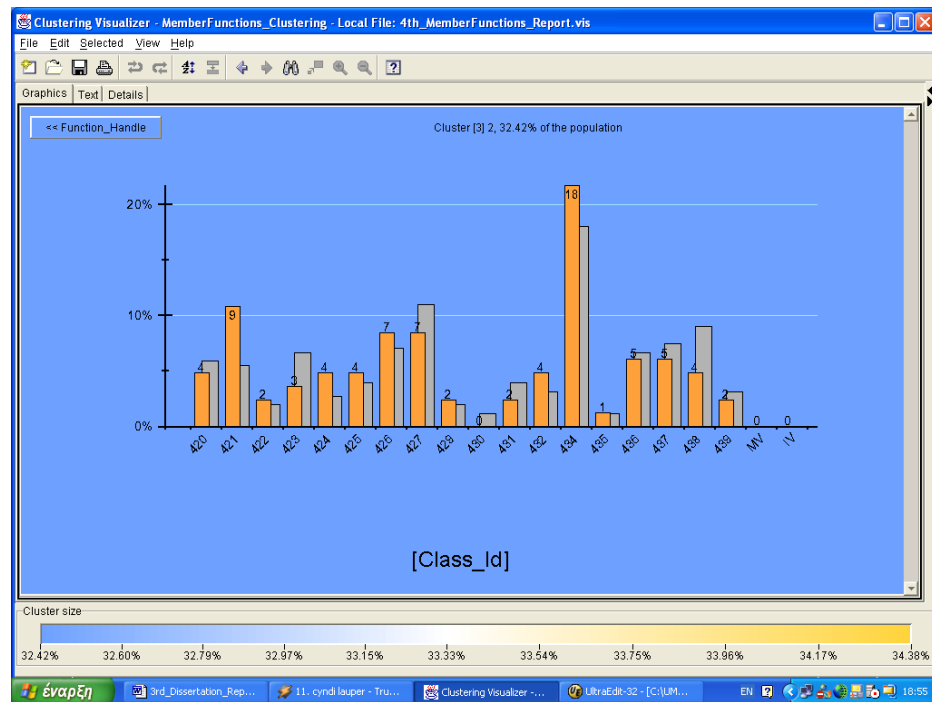


Figure7-30 CompDB: Third cluster of `Member_Functions` table. Distribution of the member functions among their classes

The final common characteristic of the member functions of this cluster is that the majority of them have zero or one parameters. This is depicted in picture (A-18) of §Appendix.

7.2.2.4 CompDB: Analysis of thr Parameters of Member Functions

The parameters of the member functions of the CompDB application classes can be divided into three clusters:

1. The first cluster, as illustrated in picture A-19 of §Appendix, represents the 42.98% of the population and consists of parameters that are passed by value. The return types of the member functions of the classes of the CompDB application vary. According to picture 7-31 the most predominant are:
 - i) pointers of type `char`
 - ii) `int`
 - iii) pointers of type `CDC`, which is a class that encapsulates device-context support. A device context defines a display environment

(such as a window on the screen). In addition, it provides functions that allow applications to draw various objects, like lines and circles [Schild 1998].

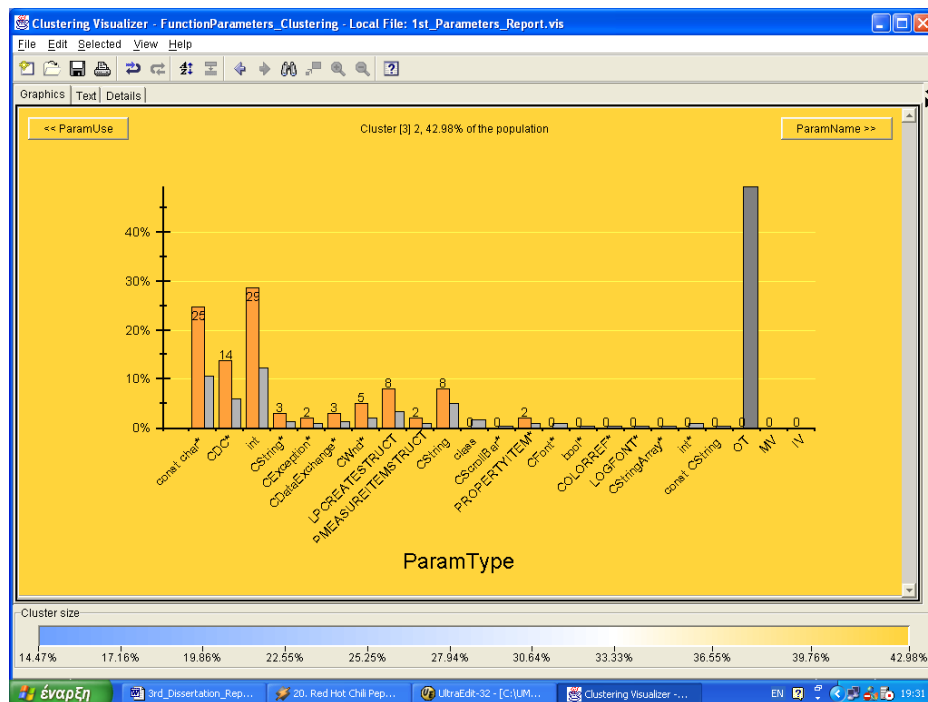


Figure7-31: CompDB: First cluster of `Function_Parameters` table Types of parameters.

2. The second cluster, as portrayed in picture A-20 §Appendix, represents the 42.55% of the population and consists of parameters that are passed by value. The return types of the member functions of the classes of the CompDB application vary. According to picture 7-32 the more predominant are:
 - i) `char`
 - ii) `UINT`, which is an unsigned 32-bit integer. It is not a standard C/C++ data type but a Window data type that MFC uses [Schild 1998].
 - iii) `COLOREF`, which is a 32-bit integer that holds an RGB color [Schild 1998].

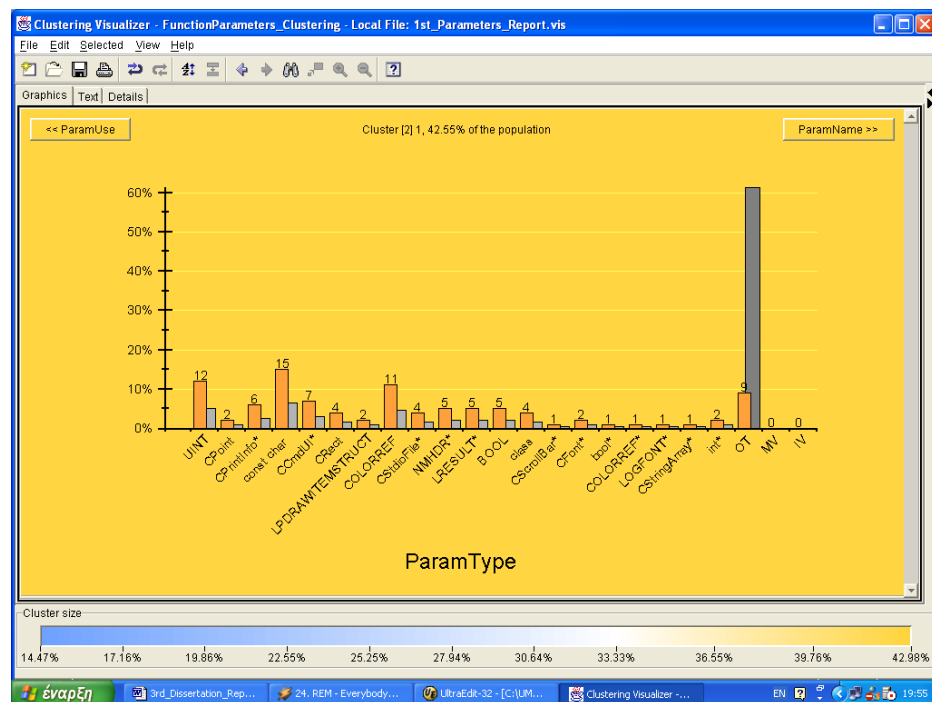


Figure7-32: CompDB: Second cluster of `Function_Parameters` table Types of parameters

3. The third cluster, as it is depicted in the picture A-21 of §Appendix, represents the 14.47% of the population and consists of parameters that are passed by reference. The return types of the member functions of the classes of the CompDB application vary, with the most predominant being (according to picture 7-33):

- i) `CDUMPCONTEXT`, which is a class that its objects provide several diagnostic messages.
- ii) `_CONNECTIONPTR`, which is a class that its objects are pointers to a Connection Interface [<http://www.devarticles.com/art/1/230/4>].

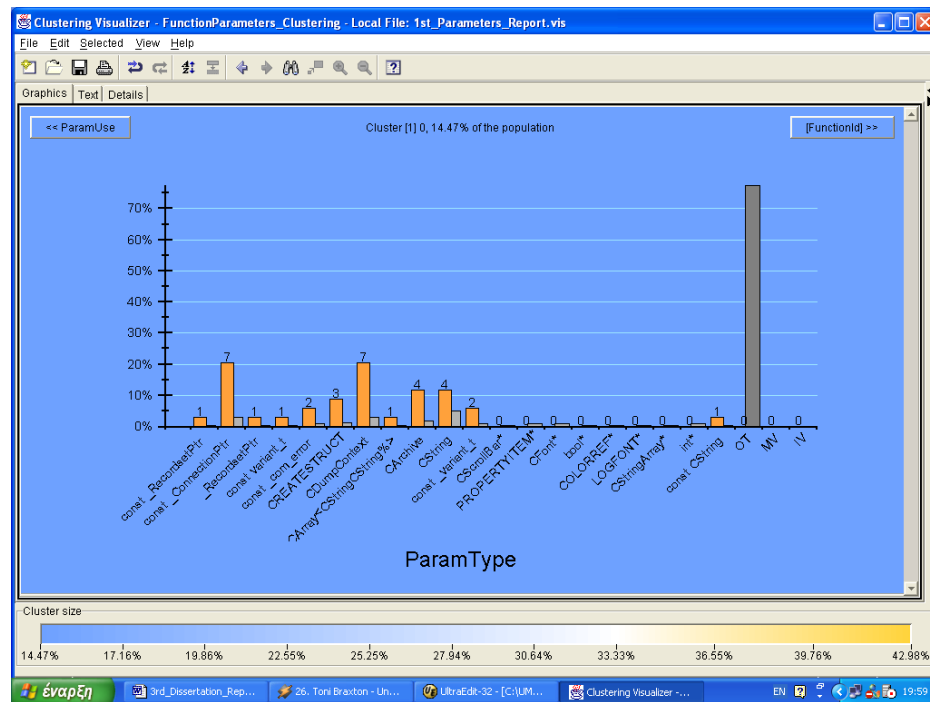


Figure7-33 CompDB: Third cluster of `Function_Parameters` table. Types of parameters

7.2.2.5 CompDB: Conclusions

Logical correlations between the classes of the `CompDB` can be found: In the first place, there are the classes deriving from the `CView` (`CQueryView`, `CPropView`, `CParaView`, `CCompDBView`). These classes apart from logical correlation, they have also structural (member data) and behavioural (member functions) similarities. They have the same size as there are no significant differences between the number of their member data and functions. Their member data has also common data types. On the other hand, their member functions have common return types and similar parameter numbers.

Another category of logically related classes are (CMainFrame, CChildFrame) classes that derived from CWnd. They also have structural similarities, since their member data have common data types. On the other hand their member functions seem to have similarities but a lower rate than the member data.

The classes `CArgDlg`, `CCompDlg`, `CQueryDlg` that derive from the class `CDialog` are also logically related and present structural and behavioural similarities. Their member data and functions have lots of similarities, such as common data types (for the member data), number of parameters and return types (for the member functions).

Nine out of the eighteen classes of the **CompDB** application (50% of its population) are logically, structurally and behaviourally correlated, which is a useful fact for the application's maintenance. For example, if the maintainer needs to change the data type of a member data in the **CParaView** class, then he/she may have to change **CQueryView**, **CPropView** and **CCompDBView**. Hence, the conclusion is that by finding logical correlations between the classes of a system, it is very likely to find also structural and behavioural correlations. This can be useful for the easier understanding and maintenance of **CompDB**. Their ability to inherit is a vital factor of the logical correlation of a system's classes. If some classes have the same superclass, then it is more likely to have similar structure and behaviour.

At this point it has to be underlined that finding logical correlations between the classes of a system is not the only way for a maintainer to understand the system. There are classes that do not have any logical correlation, but they can have either structural or behavioural similarities. Examples of this are the **CAdoList** and **CComboButton** classes, which have member data of the same type (structural similarity). They are pointers of type **CPen** and **CBrush**. Therefore, if the maintainer wants to make a change in the type of the member data of **CAdoList**, is very likely that he will do the same change to **CComboButton** as well.

8. Conclusions – Future Work Chapter

8.1 Introduction

This chapter presents the conclusions derived from the investigation in terms of program understanding and software maintenance, with the help of data mining techniques such as clustering. It outlines the challenges that were met, the solutions implemented and their outcome. It also suggests ways on how to further develop this approach in terms of program understanding of the C++ Source Code.

8.2 Overview

The methodology developed in order to carry out this investigation, consists of the following steps:

1. Contacting background research in order to gain appropriate knowledge and form the theoretical basis for the development of the whole methodology. This includes an examination of program understanding strategies (such as bottom-up, top-down, opportunistic), categories of software maintenance changes (such as adaptive, protective, preventive and perfective), and data mining techniques.
2. Gathering the requirements for the C++ source code preprocessing system. The requirements are concerned with the Model of Input Data, the Preprocessing Application's Front-End (G.U.I.) and Back-End (Algorithm). Investigation of the state of the art was the main source for these requirements.
3. Designing the system based on concepts derived from the previous steps. An example of this is the choice of the top-down program understanding strategy, which forms the basis of the Preprocessing Algorithm (Back-End) for the Application. UML is used in order to depict main concepts, such as the Model of the Input Data, during the design process.
4. Implementing the system based on the outcome of the previous step. The system consists of the Preprocessing Application that parses the C++ source code, a database management system where data is stored in a format capable in order to perform cluster analysis, and a data mining tool

that carries out the particular analysis. Java is used for the implementation of the Preprocessing Application and SQL Server 2000 is used as the database management system for the `PreprocessingDB` database. IBM's Intelligent Miner is used as the data mining tool for performing clustering analysis on `PreprocessingDB`'s data.

5. Testing the implemented system to verify that the Preprocessing Application is producing the expected results. A testing script is used for this reason, according to which a header file is opened and preprocessed. The expected outcome is the successful insertion of the extracted information in the respective tables of the `PreprocessingDB` database, and the right feedback by the G.U.I of the Preprocessing Application.
6. Evaluation of the system's results. The intention here is to confirm that the entire system is working according to the stated requirements, and that the overall approach and methodology are valid. For this reason, the clustering analysis results of the two different software systems (`CAccessReports`, `CompDB`) are examined, and useful conclusions about the structure of them are drawn.

8.3 Lessons learned

The carried out research in the domain of program understanding and software maintenance with data mining, and the scientific approach that was conducted in order to provide an appropriate solution, resulted in useful lessons derived from the development of the above methodology. Scientific literature was studied and previous solutions were reviewed. This helped in the creation of a strong scientific background concerning the solution provided, and contributed in gaining a structured and systematic approach in problem solving. An example of this (as mentioned in §8.2) was the choice of the top-down program understanding strategy, which formed the basis for the Preprocessing Algorithm (Back-End) of the Preprocessing Application.

Moreover, a valuable lesson was the improvement of software engineering and programming skills for the development of the Preprocessing Application. The waterfall model was chosen in order to develop the particular application and monitor its progress. Java was used in order to build the G.U.I. of the application and implement the Preprocessing Algorithm. SQL Server 2000 was used for the

management of the Preprocessing DB and IBM's Intelligent Miner for the clustering analysis in relation to its data.

8.4 Conclusions

8.4.1 Conclusions for the methodology

The following conclusions can be derived from the comparison of the methodology described in §8.2 with the solutions described in §2.7.1, §2.7.2, §2.7.3, §2.7.4.

This project focuses on program understanding, unlike the method presented in §2.7.1, which focuses on software reliability assessment. The choice of the clustering data mining technique and the appropriate Model of Input Data indicate the principal area that the project is concerned with.

Then, it is designed specifically for C++ source code and is tested using larger datasets than the solution of §2.7.2. Two systems were used in order to test this solution. `CAccessReport` with 53 files of 224KB in total, and `CompDB` with 18 files of 70KB in total.

The third conclusion is that this solution, unlike the one at §2.7.3, is not fully automated. As soon as the Preprocessing Application finishes the C++ source code preprocessing, the user has to use a separate data mining tool. This particular tool is the IBM's Intelligent Miner. The solution and the methodology developed do not make use of any custom clustering algorithms.

Finally, and unlike the solution in §2.7.4, this work uses only cluster analysis in order to facilitate the understanding of a software system. What is more, according to the level of detail that the user requires; relations of high level (such as classes) and of medium level (such as functions) among components can be found.

8.4.2 Conclusions for the Preprocessing Application

First of all, the Preprocessing Application meets the criteria that were selected during the design phase. Its Front-End (G.U.I.) is user-friendly because:

- It is understandable. There are not many controls on the "Preprocessing Form" that the user can interact with, just buttons ("Open", "Preprocess")

and “Cancel”) that have discrete roles and their functionalities do not overlap.

- It is tolerant to mistakes. Exception handling mechanisms are used in the Preprocessing Application in order to prevent the application from crashing after a user error, or during the preprocessing of a header file. Error messages are also used in order to ensure that the user will be able to observe the progress of the preprocessing process and restart it from an early stage in case that something goes wrong.
- It provides feedback to the user regarding the actions undertaken. A progress bar is used in order to depict the progress of the extraction and the insertion of the data in the database. A text area is also provided to the user, in order to help him/her understand the stage of the Preprocessing Algorithm that the application performs.
- It is task-suitable as it does not provide too much functionality that can confuse the user. The Preprocessing Application consists only of two forms: The “C++ Source Code Preprocessing Application” window, which is the parent window and has the menu with the functionalities of the Preprocessing Application and the “File Preprocessing” form, which is the child window.

In addition, the Back-End (Algorithm) meets its respective requirements. The data that is retrieved from the source code (header file) is stored in a way that is easy to apply cluster analysis. Suitable attributes were selected in order to avoid an unnecessarily increase of the records’ similarity for the Model of Input Data entities. For instance, the `Category` attribute of the Member Data and Classes entities was used in order to define the category of the data members and the classes. Originally the design used to have three binary fields (`IsPublic`, `IsPrivate`, `IsProtected`) instead of this attribute.

The Back-End (Preprocessing Algorithm) is also able to process a large number of files in a reasonable amount of time. For example it took less than fifteen minutes to preprocess (that is parse the files, extract information and insert it in the `PreprocessingDB` database) the 53 separate header files of the `CAccessReport` application of a total size of 224KB.

Finally, the clustering analysis of the output of the Preprocessing Application indicates that:

- The output is valid. The information reflects the actual structure of both the CAccessReport and CompDB systems.
- The output is useful. There are patterns discovered that a potential maintainer would have found very useful, as they could help him/her understand the system under maintenance. An example of this is the following pattern: by finding logical correlations between the classes of a system it is very likely to find also structural and behavioural correlations.
- The patterns discovered were not entirely novel. Most of them have been theoretically described before, but nonetheless were verified by the clustering analysis. A maintainer could have expected results like the previous example. A probable novel pattern would be the fact that either structural or behavioural similarities can be found in classes that are not logically correlated, or in other words they do not inherit from the same class.

8.5 Future Work

The evaluation of the investigation work and the conclusions that were presented in §8.4.1 and §8.4.2 can lead to further work suggestions that can improve this project.

In the first place, the Preprocessing Application is not database – independent, as it functions only with the SQL Server 2000 DBMS. Thus, there is a need for administrative functionalities for the maintainer in order to be able to change the database he/she works with.

Secondly, the Preprocessing Application has to be flexible in order to support several editions of C++. There is the standard C++, Microsoft's Visual C++ and Visual C++ .NET, MFC (Microsoft Foundation Classes), and Borland C++ that can be used for the development of an application. Such editions are different in terms of their data types and their syntax. Hence, the application should be designed in order to be flexible to adjust with these editions of C++.

On the other hand, the Model of the Input Data is designed to facilitate the clustering analysis. Hence, the input model has to change in order to be able to use

other data mining techniques such as association rules. New attributes (fields) can be added to the description of the system entities (database tables).

An added suggestion is to process some elements of the *.cpp files of the analysed system. These elements can be:

- The header files which are included in each file. This is another way to discover logical correlations between the classes of a system, other than inheritance. As it is previously mentioned, logical correlations among the classes of a system are likely to indicate structural and behavioural correlations.
- The definitions of the constants that are used in the *.cpp file. This can help to find more structural similarities between the classes of a system, even in case they do not have any logical correlations.

In order to do that, the Model of the Input Data should change by adding new attributes to the existing entities.

In conclusion, the proposed solution can be fully automated. This means that a module that will implement several data mining algorithms can be integrated in the existing Preprocessing Application, in order to start the data analysis automatically, as soon as the data preprocessing is completed.

References

Papers

1. [Chen et al. 2002]. Kai Chen, Christos Tjortjis, Paul Layzell. A Method for Legacy Systems Maintenance By Mining Data Extracted From Source Code. In the case studies of IEEE 6th European Conference Software Maintenance and Reengineering (CSMR 2002).
2. [Clements et al. 1996]. Paul Clements, Robert Krut, Ed Morris, Kurt Wallnau. The Gadfly: An approach to Architectural – Level System Comprehension. Fourth IEEE Workshop on Program Comprehension, Berlin March 1996.
3. [IEEE Software Maintenance Standards 1998]
4. [Lientz et al. 1978]. B.P.Lientz, E.B. Swanson, and G.E. Tompkins, Characteristics of Software Application Maintenance, Communications of the ACM, Volume 21, Number 6, pp. 1-3, June 1978.
5. [Mancoridis et al. 1998], S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, E.R. Gansner. Clustering to Produce High-Level System Organisations of Source Code. In the IEEE Proceedings of the 1998 International Workshop on Program Understanding.
6. [Padula 1993]. Alan Padula, Use of a Program Understanding Taxonomy at Hewlett-Packard, Second IEEE Workshop on Program Understanding, July 1993.
7. [Sartipi 2001], Karman Sartipi, A Software Evaluation Model Using Component Association Views. In the IEEE Proceedings of the 2001 International Workshop on Program Understanding.
8. [Tjortjis and Layzell 2001]. Christos Tjortjis, Paul Layzell. Using Data Mining to Assess Software Reliability. In the proceedings of the 12th IEEE International Symposium Software Reliability Engineering, (ISSRE2001).
9. [Von Mayrhauser, Vans 1995]. Program comprehension during software maintenance and evolution, Von Mayrhauser Anneliese, Vans Anne Marie, Computer, Volume: 28 Issue: 8 , Aug 1995, Page(s): 44 -55, IEEE Journal

Books

10. [Deitel & Deitel 1999], H. M. Deitel, P. J. Deitel, Java How to Program (Covers Java 2 Introducing Swing), Prentice Hall 1999.
11. [Fayyad et al. 1996]. Usama M. Fayyad, Gregory Piatetsky-Sapiro, Padhraic Smith, and Ramasamy Uthurusamy. Advances in Knowledge Discovery and Data Mining. The MIT press 1996.
12. [Han, Kamber 2001], Jiawei Han, Micheline Kamber, Data Mining Concepts and Techniques, Morgan Kauffman Publishers 2001.
13. [Lafore 1999]. Robert Lafore, Object-Oriented Programming in C++, Third Edition, SAMS Publishing 1999.
14. [Piroumian 1999], Vartan Piroumian, Java GUI Development, SAMS 1999.
15. [Schild 1998], Herbert Schildt, MFC Programming from the Ground Up, Osborne/McGraw-Hill 1998.
16. [Shepherd and Wingo 1996] George Shepherd and Scot Wingo, MFC Internals Inside the Microsoft Foundation Class Architecture, Addison-Wesley Developers Press 1996
17. [Takang, Grupp 1996].Armstrong A. Takang and Penny A. Grubb. Software Maintenance Concepts and Practice. International Thompson Computer Press 1996.

URLs

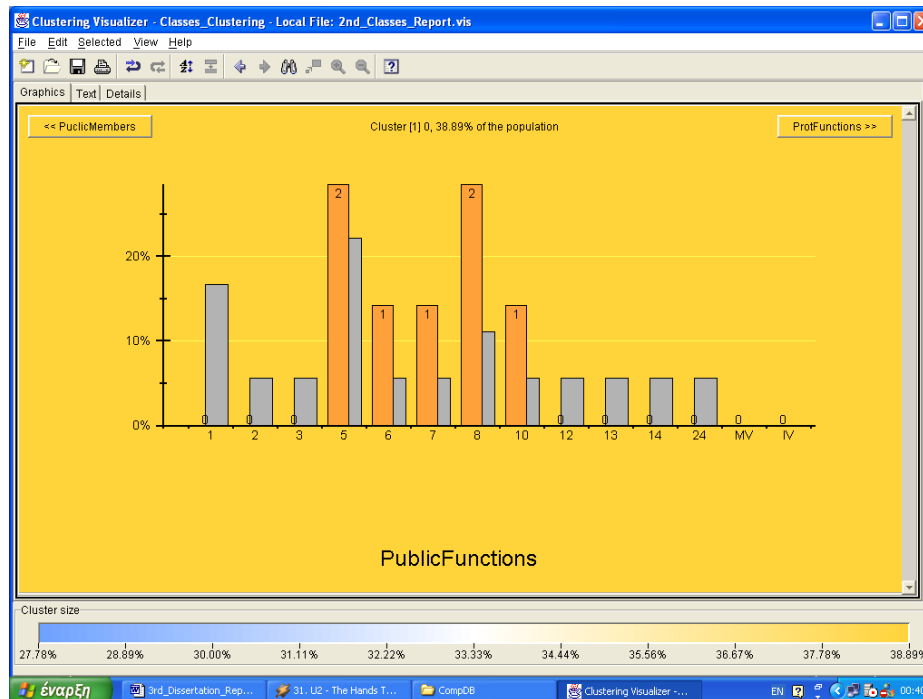
18. <http://developer.kde.org/documentation/standards/kde/style/basics/usage.html>
19. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_Automation_Clients.asp
20. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/vclrfcoledispatchdriveroperatorlpdispatch.asp>
21. http://www.codeguru.com/mfc_database/access_reports_class.shtml
22. http://www.codeguru.com/mfc_database/CompDB.html
23. <http://www.codeproject.com/interview/tomarcher3jun2001.asp>
24. <http://www.devarticles.com/art/1/230/4>

25. <http://www-ksl-svc.stanford.edu:5915/doc/frame-editor/what-is-a-class.html>

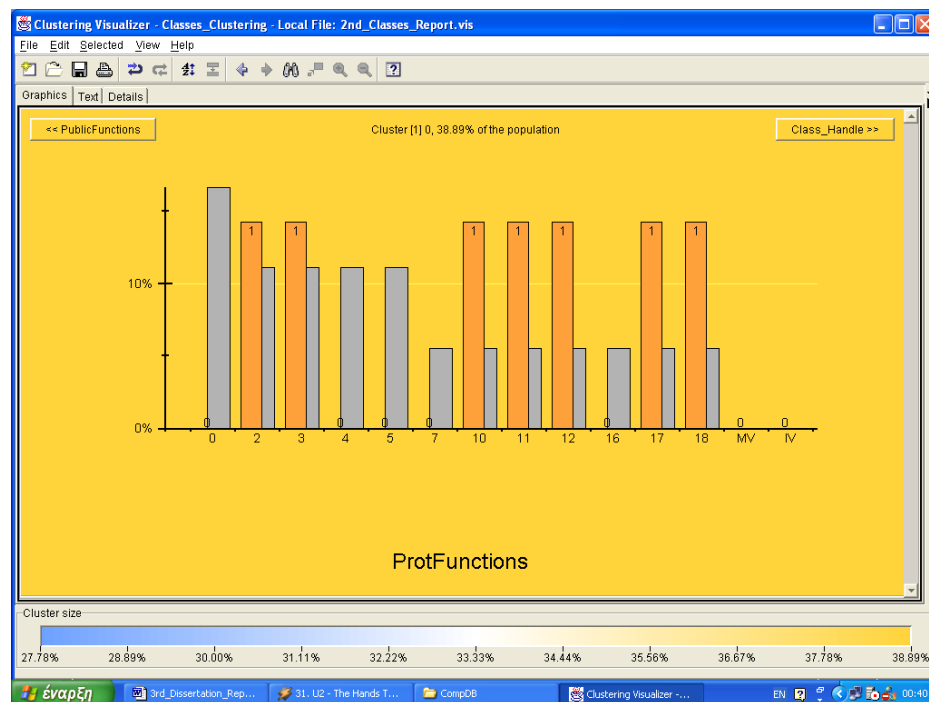
Appendix

Screenshots from the Analysis of CompDB Application

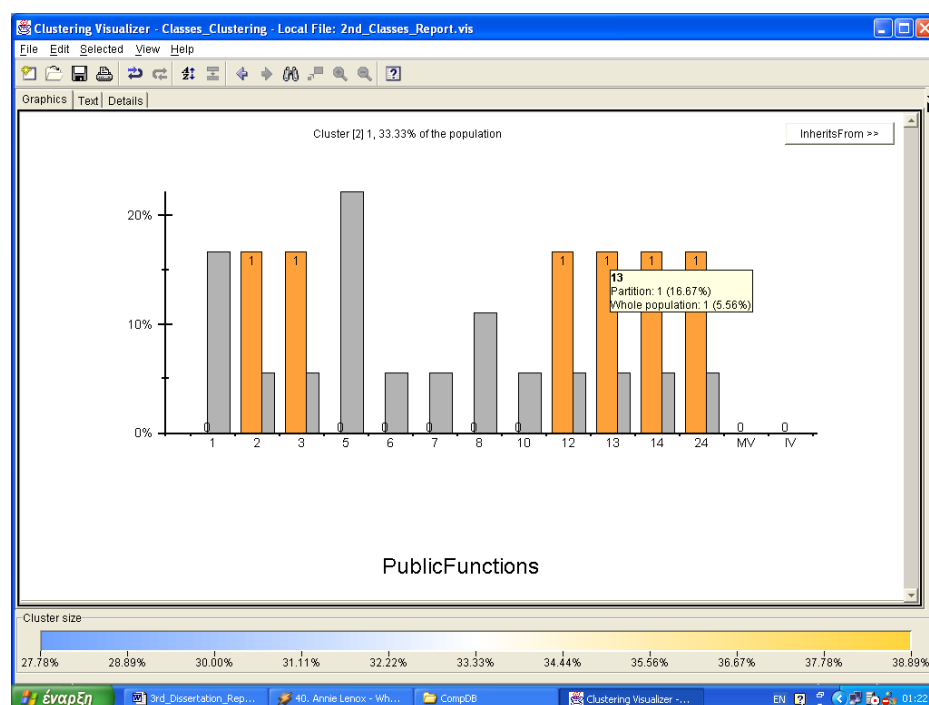
CompDB: Screenshots from Classes Analysis



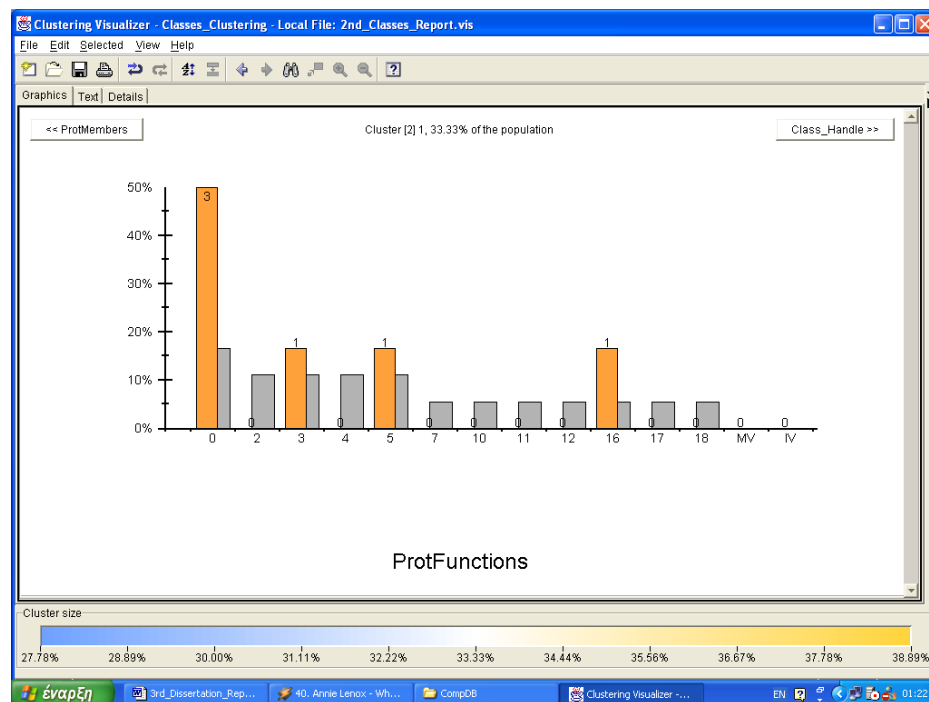
FigureA-1: CompDB: First cluster **Classes** Table. Distribution of the values that represent the number of the Public Functions



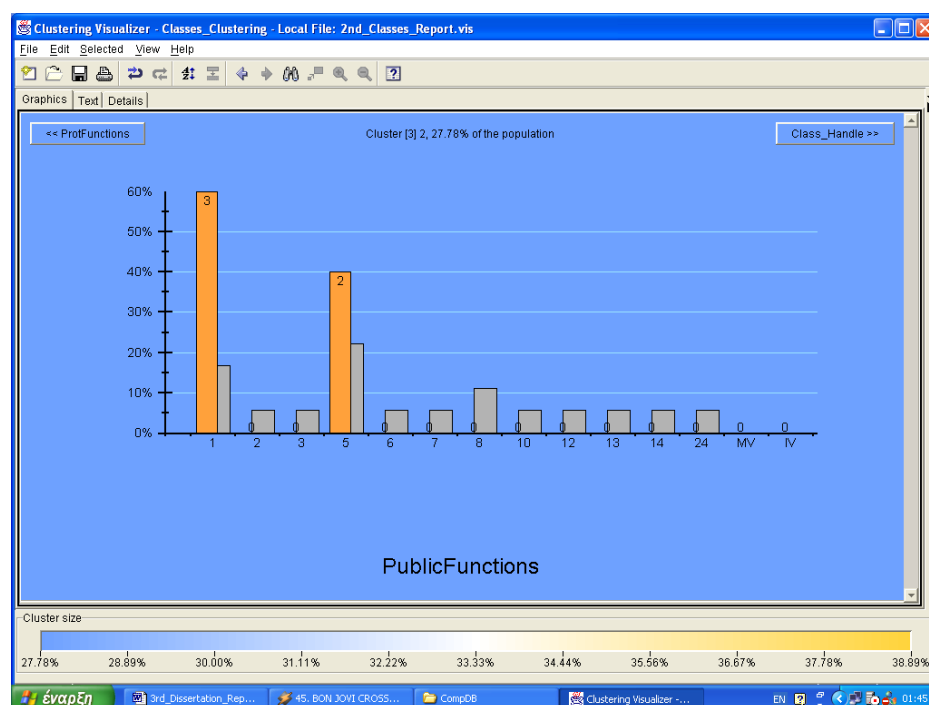
FigureA-2: CompDB: First cluster **C**lasses Table. Distribution of the values that represent the number of the Protected Functions



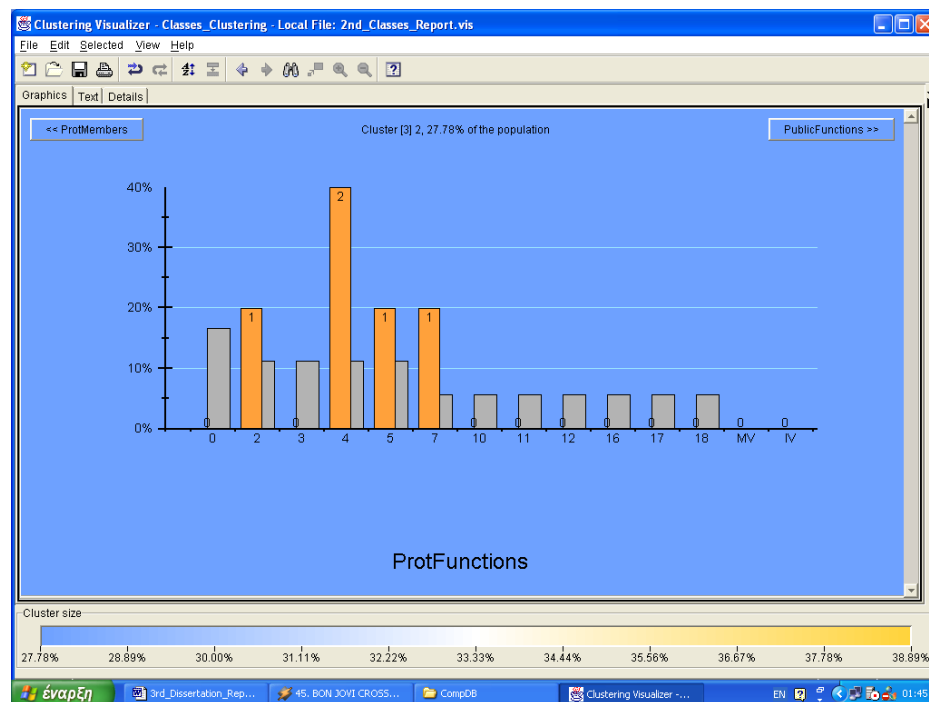
FigureA-3: CompDB: Second cluster of **C**lasses Table. Distribution of the values that represent the number of the Public Functions



FigureA-4: CompDB: Second cluster of **C**lasses Table. Distribution of the values that represent the number of the Protected Functions

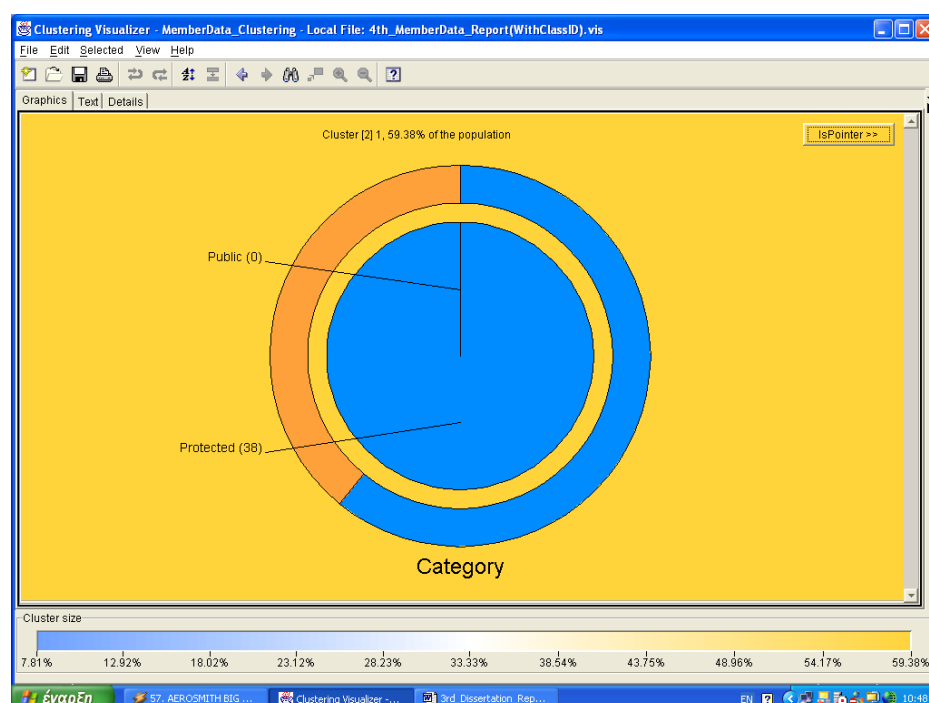


FigureA-5: CompDB: Third cluster of **C**lasses Table. Distribution of the values that represent the number of the Public Functions

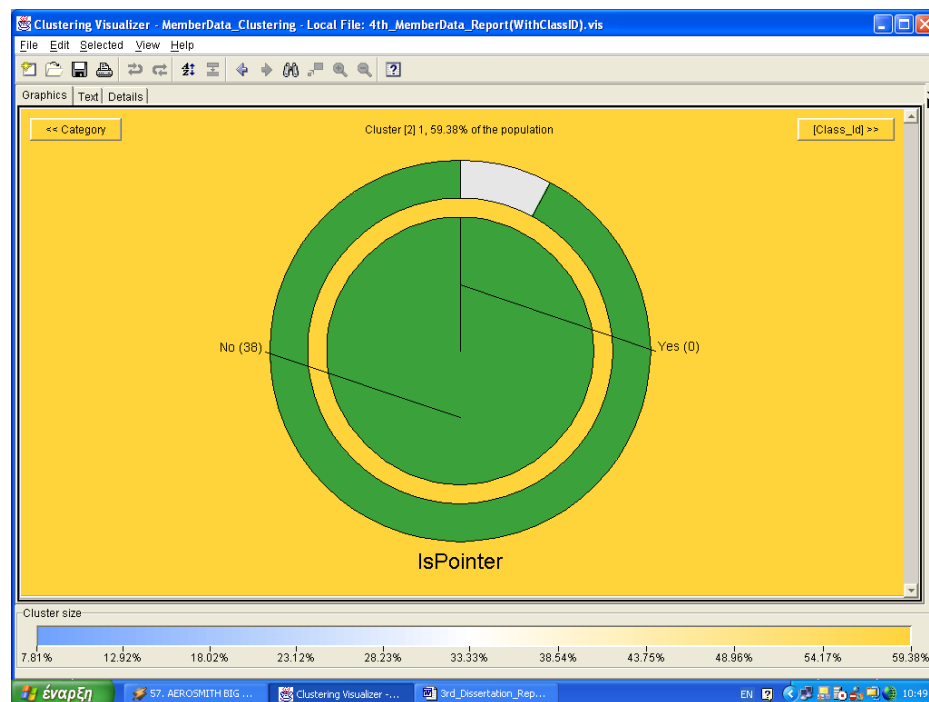


FigureA-6: CompDB: Third cluster of **Classes** Table. Distribution of the values that represent the number of the Protected Functions

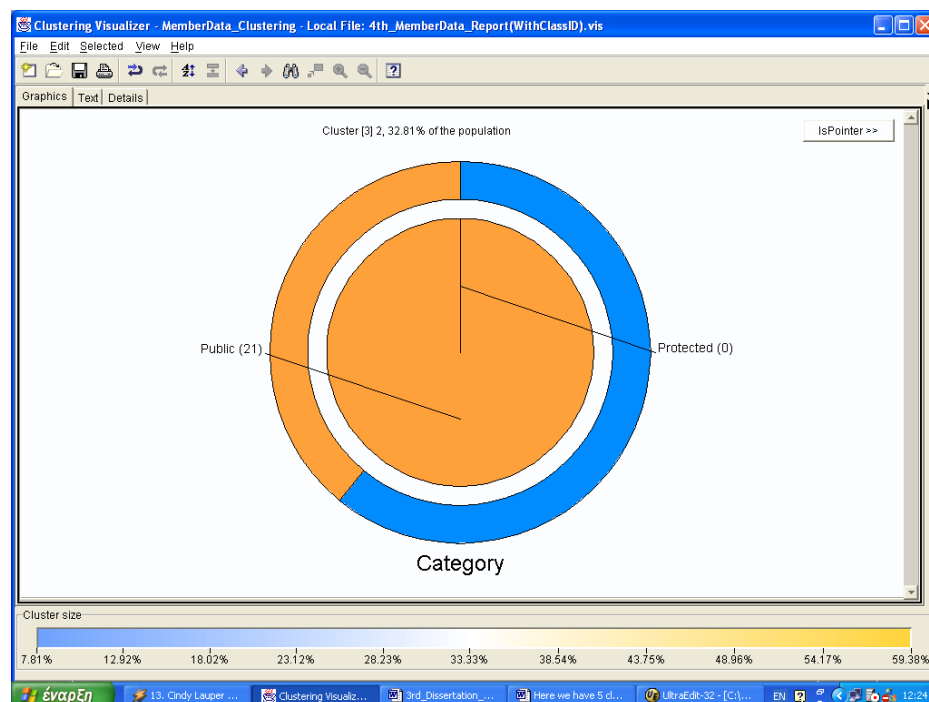
CompDB: Screenshots from Member Data Analysis



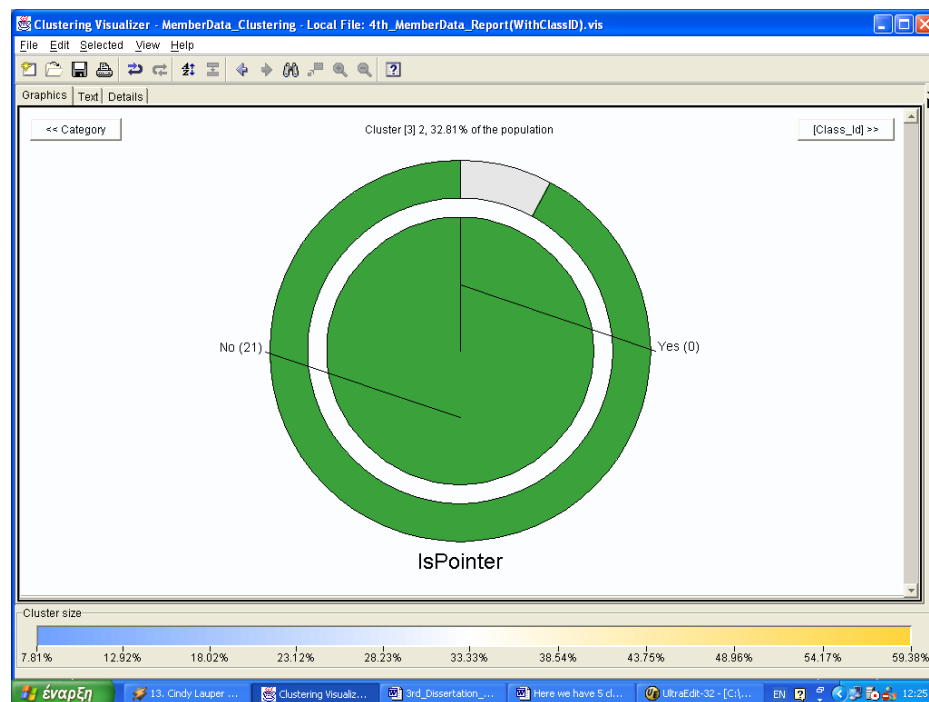
FigureA-7: CompDB: First cluster of **Member_Data** Table. Category of member data



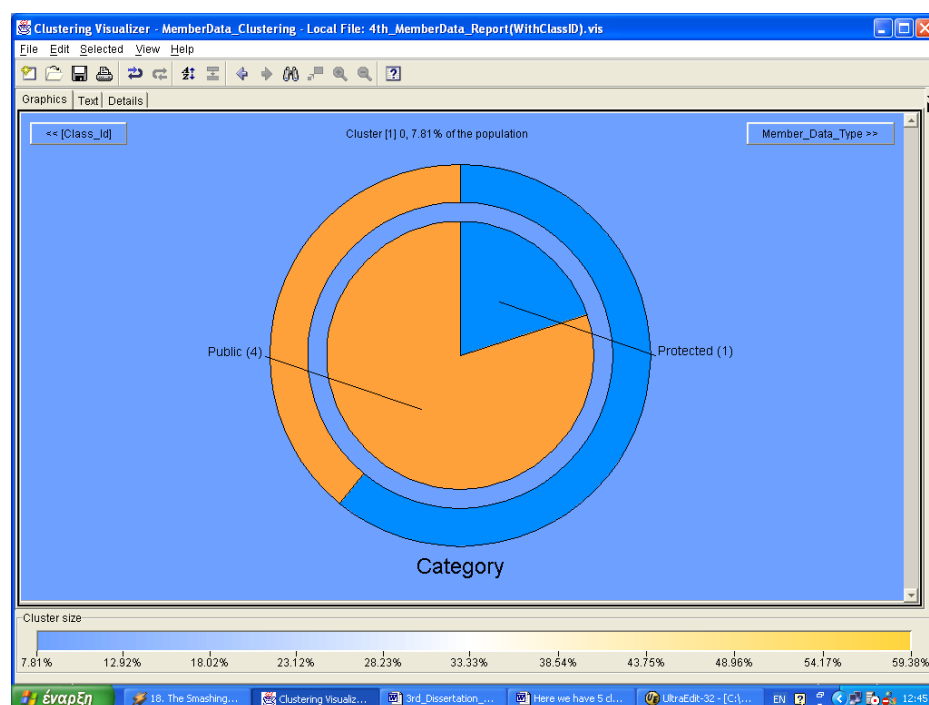
FigureA-8: CompDB: First cluster of Member_Data Table. Information that describes if the member data are pointers or not



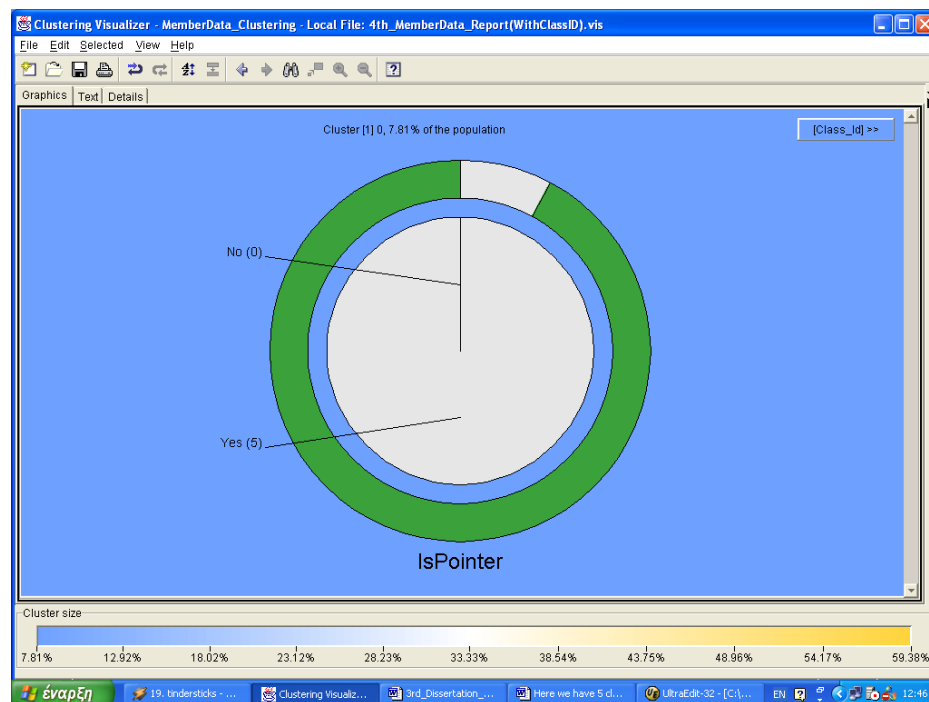
FigureA-9: CompDB: Second cluster of Member_Data Table. Category of member data



FigureA-10: CompDB: Second cluster of Member_Data Table. Information that describes if the member data are pointers or not

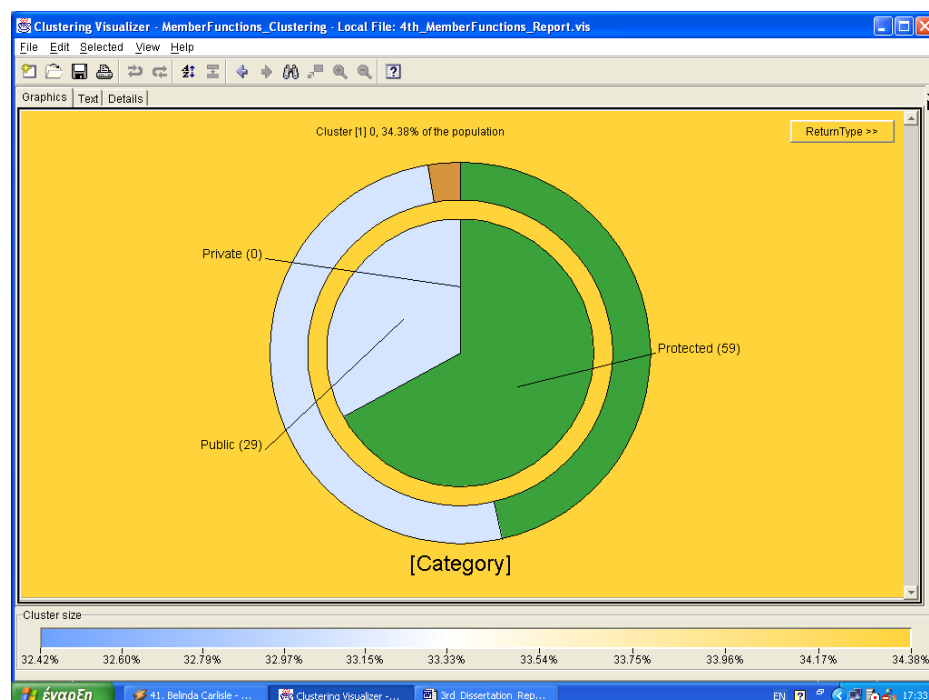


FigureA-11: CompDB: Third cluster of Member_Data Table Categories of member data

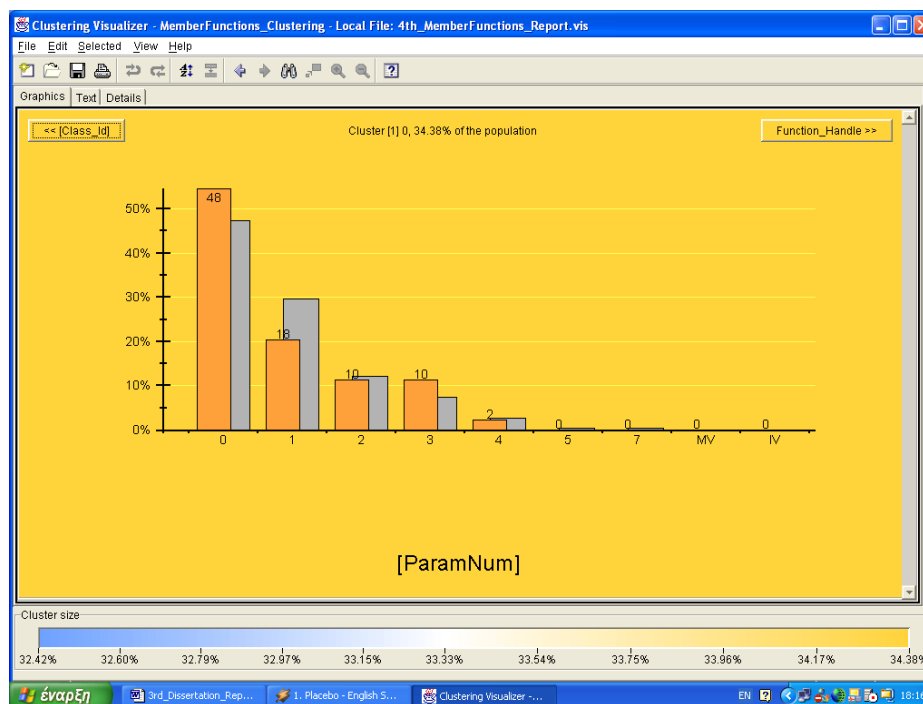


FigureA-12: CompDB: Third cluster of Member_Data Table. Information that describes if the member data are pointers or not

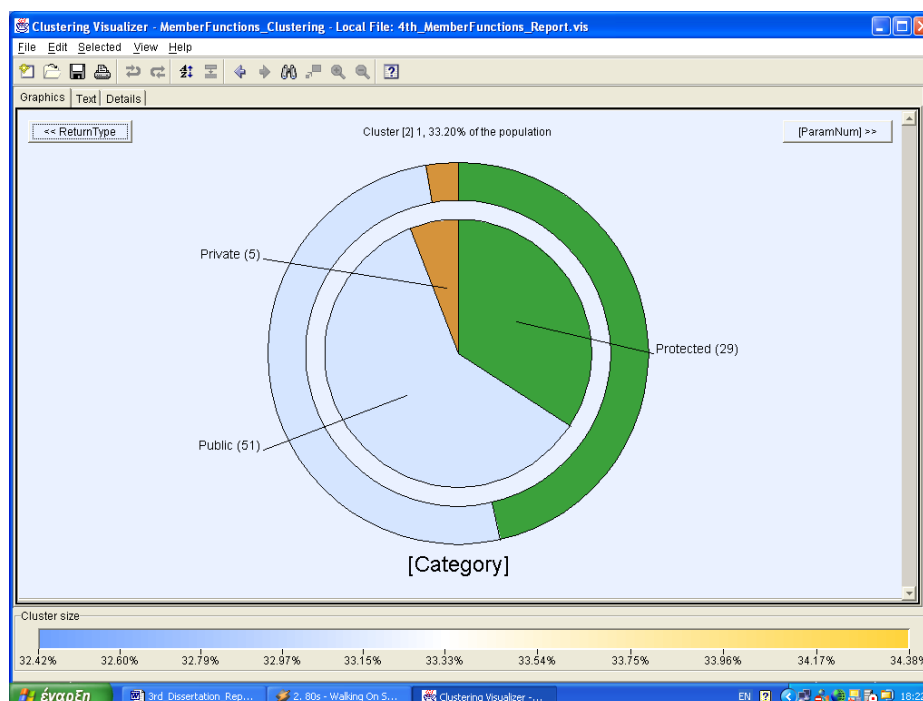
CompDB: Screenshots from Member Functions Analysis



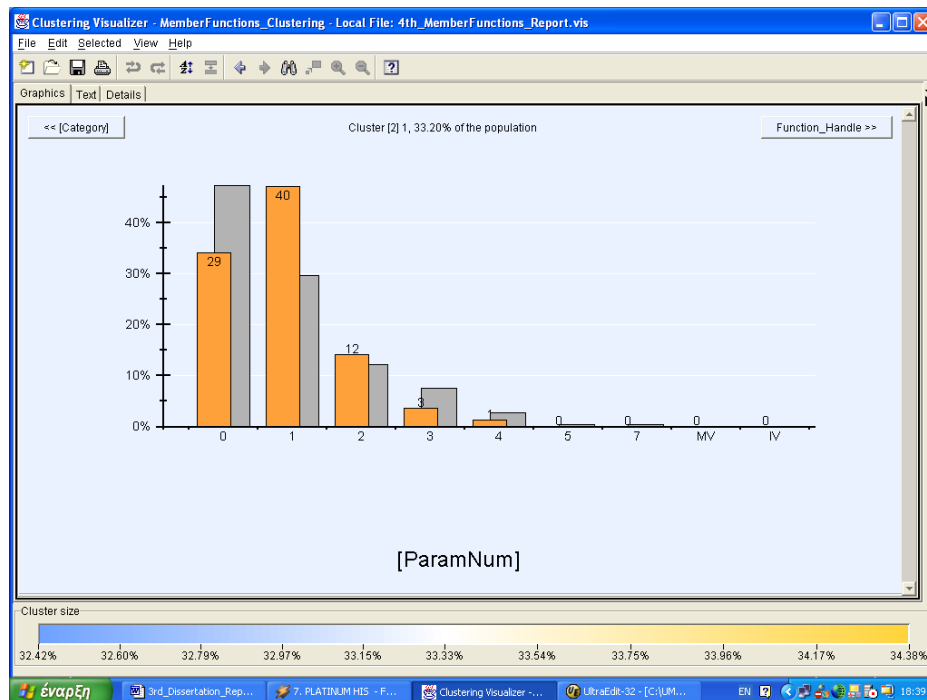
FigureA-13: CompDB: First cluster of Member_Functions table. Categories of the member functions



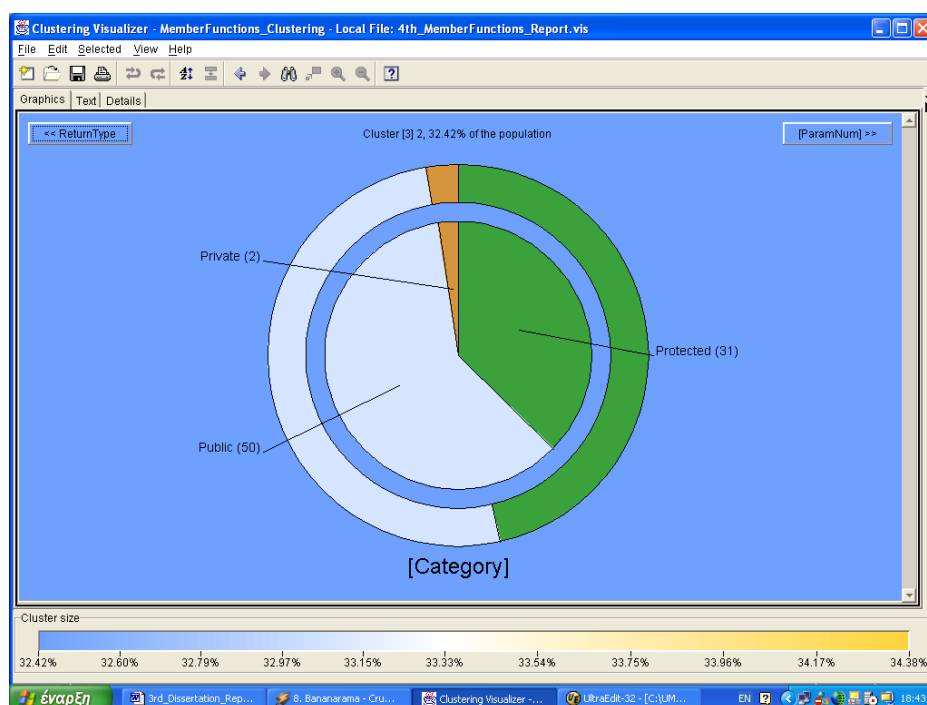
FigureA-14: CompDB: First cluster of Member_Functions table. Number of parameters of the member functions



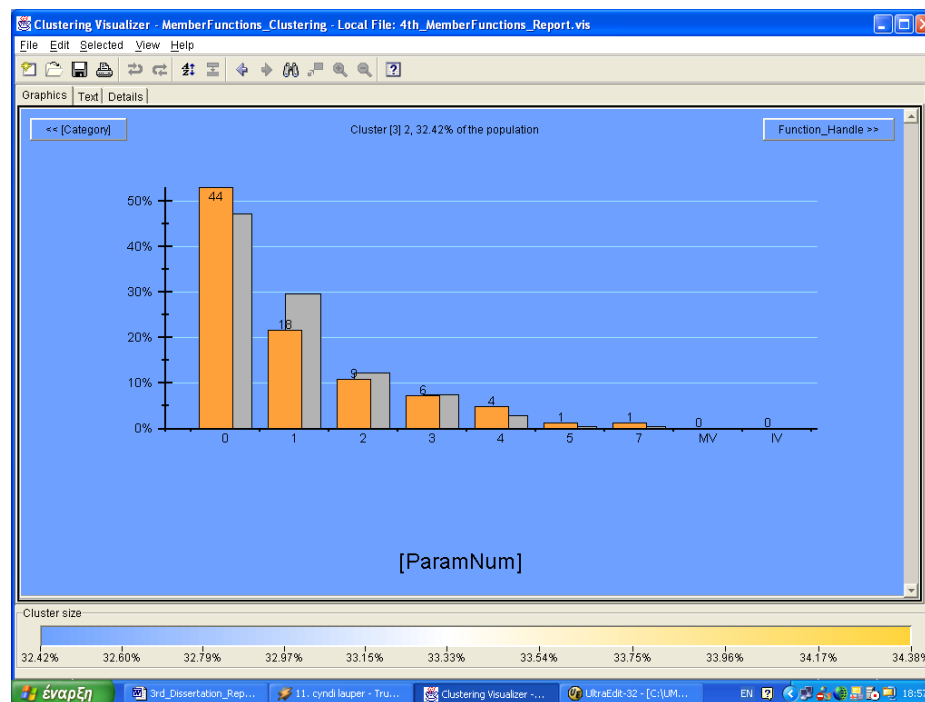
FigureA-15: CompDB: Second cluster of Member_Functions table. Categories of the member functions



FigureA-16: CompDB: Second cluster of Member_Functions table. Number of parameters of the member functions

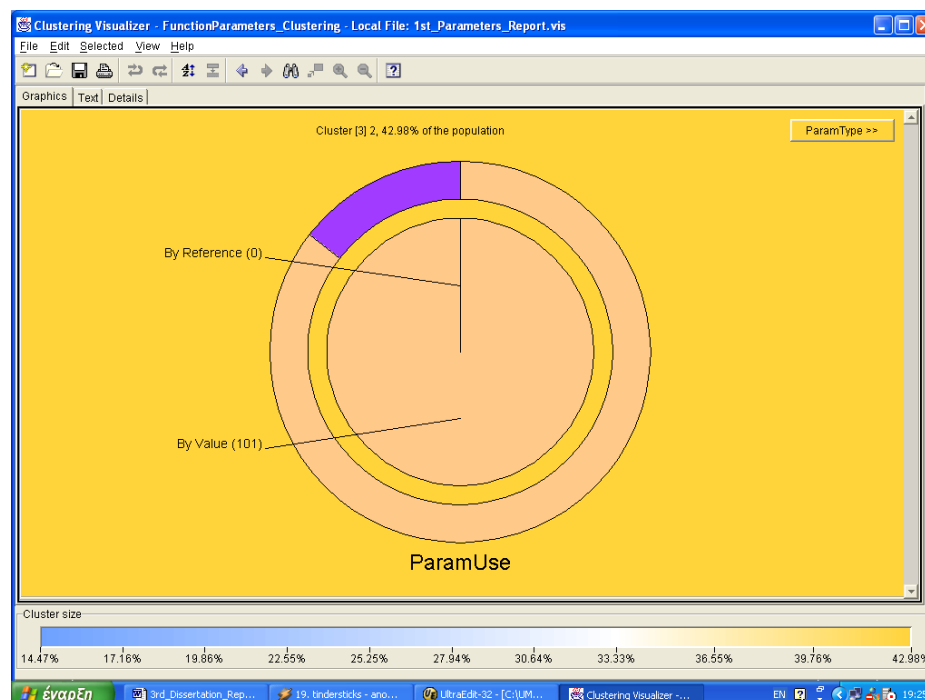


FigureA-17: CompDB: Third cluster of Member_Functions table. Categories of the member functions

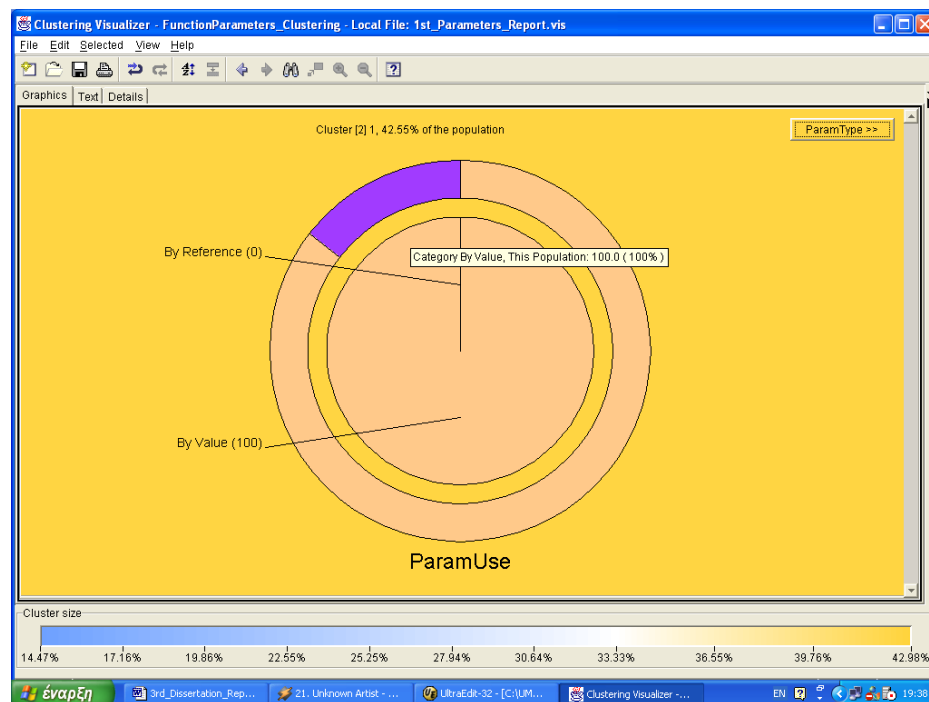


FigureA-18 CompDB: Third cluster of Member_Functions table. Number of parameters of the member functions

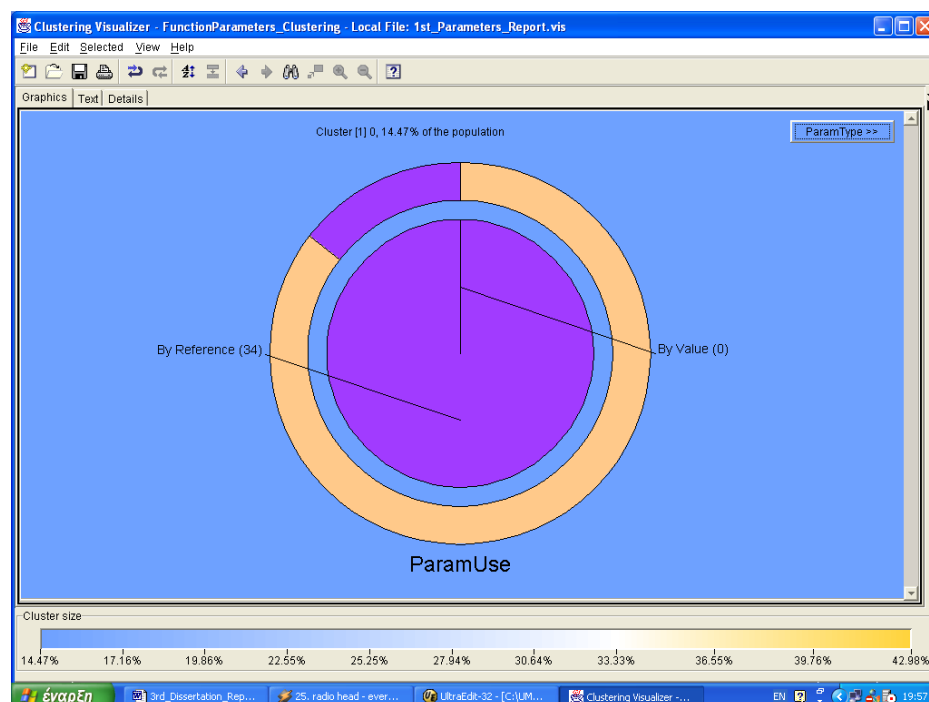
CompDB: Screenshots from Parameters of Member Functions Analysis



FigureA-19: CompDB: First cluster of Function_Parameters table. Use of Parameters



FigureA-20 CompDB: Second cluster of **Function_Parameters** table Use of Parameters



FigureA-21 CompDB: Third cluster of **Function_Parameters** table Use of Parameters