AUTOMATING BUSINESS INTELLIGENCE RECOVERY IN SOFTWARE EVOLUTION

PHD THESIS

JIAN KANG

SOFTWARE TECHNOLOGY RESEARCH LABORATORY
SCHOOL OF TECHNOLOGY
DE MONTFORT UNIVERSITY
LEICESTER, UK
FEBRUARY 2009
Acknowledgement

I would like to express my gratitude to many people who have helped me in different ways with the development of this thesis. Without their continuous support and guidance, the completion of my research leading to this thesis would be impossible.

I wish to express my most sincere thanks to my supervisor Prof. Hongji Yang for his invaluable advice, support and encouragement. I will carry out his guidance though out my life.

Meanwhile, I would like to thank colleagues at the Software Technology Research Laboratory at De Montfort University and for their support and feedback, and providing such a stimulating and friendly working atmosphere. It is fortunate for me to work and study with them.

This thesis would not be possible without the continuous support of my father Chengye, my mother Ying, my wife Jie, and all my friends who gave me the strength and will to success.
Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, UK from October 2004 to February 2009. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied by me based on this work.
Abstract

The theme of this thesis is to pave a path to vertically extract business intelligence (BI) from software code to business intelligence base, which is a tank of BI. Business intelligence is the atomic unit to build a piece of program comprehensibility in business logic point of view. It outstands because it covers all reverse engineering levels from code to specification. It refers to technologies for the localisation, extraction, analysis of business intelligence in software system. Such an approach naturally requires information transformation from software system to business intelligence base, and hence a novel set of automatic business intelligence recovery methods are needed.

After a brief introduction of major issues covered by this thesis, the state of art of the area coined by the author as “business intelligence elicitation from software system”, in particular, the kinds of business intelligence that can be elicited from software system and their corresponding reverse engineering technical solutions are presented.

Several new techniques are invented to pave the way towards realising this approach and make it light-weight. In particular, a programming-style-based method is proposed to partition a source program into business intelligence oriented program modules; concept recovery rules are defined to recover business intelligence concepts from the names embedded in a program module; formal concept analysis is built to model the recovered business intelligence and present business logic. The future research of this task is viewed as “automating business intelligence accumulation in Web” which is defined to bridge work in this thesis to nowadays Web computing trends.

A prototype tool for recovering business intelligence from a Web-based mobile retailing system is then presented, followed by case study giving evaluation on the approach in different aspects.

Finally, conclusions are drawn. Original contributions of this research work to the field of software reverse engineering are made explicit and future opportunities are explored.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2 Computational Complexity of BIR</td>
<td>166</td>
</tr>
<tr>
<td>10.3 Original Contributions</td>
<td>166</td>
</tr>
<tr>
<td>10.4 Conclusions of Auto-BIR in the Thesis</td>
<td>168</td>
</tr>
<tr>
<td>10.5 Future Work</td>
<td>169</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>170</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>183</td>
</tr>
<tr>
<td>Programming Style Based Program Partition</td>
<td>183</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>198</td>
</tr>
<tr>
<td>Business Intelligence Concept Recovery</td>
<td>198</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>207</td>
</tr>
<tr>
<td>List of Publications by Author</td>
<td>207</td>
</tr>
</tbody>
</table>
Table of Figures and Tables

Figure 4.1 Comparisons between Forward Engineering and Reverse Engineering ................................................. 44
Figure 4.2: Proposed Research Steps of Business Intelligence Recovery .......................................................... 47
Table 4.1: Survey of Techniques for Business Intelligence Recovery .............................................................. 48
Figure 4.3: Proposed Business Intelligence Recovery Techniques ....................................................................... 49
Figure 4.4: Business Intelligence Recovery and Reverse Engineering ............................................................. 50
Figure 4.5: A Five-Tier Perspective on Business Intelligence Recovery ............................................................ 50
Figure 4.6: Research Methods on Business Intelligence Recovery ................................................................. 52
Figure 4.7: Proposed Techniques of Business Intelligence Recovery ............................................................... 54
Table 4.2: Business Intelligence Slices in Program Modules ............................................................................... 56
Figure 5.1: Simplification of a Complex Business Intelligence Base ............................................................... 65
Figure 5.2: Simplification of a Business Intelligence Base of an Online Mobile Retailing System ...................... 65
Table 5.1: Examples of Relationships between Nodes in a Semantic Network ................................................... 66
Table 6.1: Programming Styles ......................................................................................................................... 77
Table 7.1: Business Intelligence Concept Recovery .......................................................................................... 94
Table 7.2: Belief of Dominate (A,B) in a Tree ....................................................................................................... 98
Table 7.3: Belief of Dominate (A,B) in an Acyclic Graph ..................................................................................... 100
Table 7.4: Belief of Dominate (B,A) in an Acyclic Graph ..................................................................................... 100
Table 7.5: Belief of Dominate (A,B) in a Directed Acyclic Graph ..................................................................... 100
Table 7.6: Belief of Dominate (B,A) in a Directed Acyclic Graph ..................................................................... 101
Table 7.7: Belief of Before (B,A) in a Tree ........................................................................................................ 104
Table 7.8: Belief of Before (A,B) in a Tree ........................................................................................................ 104
Table 7.9: Belief of Before (B,A) in an Acyclic Graph ......................................................................................... 104
Table 7.10: Belief of Before (A,B) in an Acyclic Graph ....................................................................................... 105
Table 7.11: Belief of Input (T,P) in a Directed Tree ........................................................................................... 107
Table 7.12: Belief of Output (P,T) in a Directed Acyclic Graph ........................................................................ 107
Table 7.11: Belief of Input (T,P) in a Directed Acyclic Graph ........................................................................ 108
Table 7.12: Belief of Output (P,T) in a Directed Acyclic Graph ........................................................................ 108
Table 7.15: Raw Relationship Table for Recovered Business Intelligence Concepts (1) ..................................... 110
Table 7.16: Raw Relationship Table for Recovered Business Intelligence Concepts (2) ..................................... 110
Table 7.17: Belief of Dominate (N1,N2) in a Directed Acyclic Graph ................................................................. 112
Table 7.18: Belief of Dominate (N2,N1) in a Directed Acyclic Graph ................................................................. 113
Table 8.1: A Small Example of Mobile Context Table ....................................................................................... 118
Figure 8.1: A Small Example of Mobile Concept Lattice ................................................................................... 119
Table 8.2: Filtering Source Code Entities ......................................................................................................... 122
Table 8.3: Functions of Filtering Business Intelligence Concepts ...................................................................... 123
Table 8.4: An Example of Generated the Tabular Concept Context ............................................................... 125
Figure 8.2: An Example of Building a Concept Lattice .................................................................................... 127
Figure 9.1: Business Model of an Online Mobile Retailing System ................................................................. 130
Table 9.1: Programming Constructs for C and Database .................................................................................. 131
Figure 9.2: Graphical Version of Business Intelligence for Mobile Retailing .................................................. 134
Table 9.2: Tabular Context of “PlanRolling” .................................................................................................... 153
Figure 9.3: Concept Lattice of “PlanRolling” ..................................................................................................... 154
Figure 9.4: Recovered Business Intelligence of an Online-Mobile Retailing System ........................................ 155
Figure 9.5: A Business Intelligence Base of Sale Fact ..................................................................................... 156
Figure 9.6: BIS Modelling Demo 1 .................................................................................................................. 157
Figure 9.7: BIB Modelling Demo 2 .................................................................................................................. 158
Figure 9.8: BIB Modelling Demo 3 .................................................................................................................. 159
Figure 9.9: Web Toolkit for Auto-BIR Techniques Evaluation ............................................................................. 160
The Auto-BIR Web toolkit platform is developed under Eclipse Open IDE environment. It is enabled with JRE to invoke Auto-BIR user interface. The demo of Auto-BIR process contains four presentation parts: the source package, recovered BI concepts and data, tabular context, and BI lattice as shown in Figure 9.9. ...................................................... 160
Figure 9.10: Concept Recovery in Program Partitioning Process ...................................................................... 161
The concept recovery process as discussed in Section 9.2.1 is shown in Figure 9.10, the confidence of the initial state, the first round and the second round of business intelligence concept recovery are the computation results. The business intelligence slices are further generated within the process of BI oriented program partition. There are 9 BI slices as shown in the figure.

Figure 9.11: Lattice Modelling the Recovered Business Intelligence

In Figure 9.11, the result of formal business intelligence concept lattice is generated. This is to demo the result of program partition and business intelligence concept recovery results as discussed in Section 9.2.3.
Chapter 1

Introduction

The overall research objective of thesis is to develop an integrated and practical approach that handles business intelligence recovery tasks. Such an approach should permit a real-time business intelligence localisation, rapid business intelligence extraction, and further business intelligence analysis. The automatic reverse engineering method means to be compositional and ensure consistency between the various stages in business intelligence recovery and should provide a reliable linkage of refinement.

Prior to this research, formal methods in [129, 74, 92, 134] are dominating techniques that were used for software reverse engineering. However, formal method techniques have limitations. Let us denote software reverse engineering as a series of activities to perform program partition and program abstraction, $A_1 \xrightarrow{t_1} A_2 \xrightarrow{t_2} \cdots \xrightarrow{t_n} A_n$, where $A_1$ is the source program, $A_n$ is the final result, and $A_i \xrightarrow{t_i}$, $1 \leq i \leq n$, is a series of program partition or program abstraction rules [94]. In other words, $A_n$ is the output from a rule-based reverse engineering system in which a mixture of partition and abstraction rules are applied to the $A_1$, which is the input of the rule-based reverse engineering system. Obviously, the comprehensibility, which is quoted as manual effort to conduct program partition and abstraction cannot be fully alternate into computing. Nonetheless, as the information set in $A_1$ is decreasing in reverse engineering process, the available data to comprehensibility is losing as well. It points out that lacking comprehensibility is the bottleneck for the rule-based reverse engineering system.

On the other hand, as with any other existing solid computing functionality in code, business intelligence which is essential to business computing is the product of software evolution. This must be demonstrated via a kind of reverse engineering that we can see some business idea operates now in the code and trace it back to the process of its design decision that precedes it in real business computing. But the crucial situation is, when the software product is implemented on hardware platform in a company, the understanding from business to software is stop. Instead, companies are going to adapt
their business style to software priority in years; while, in very seldom cases, the software re-engineering could performed after that, since the system is lacking of program understanding in business logics. Business intelligence is not rich enough to build new software component to meet business requirement and then the company fails in the market competition.

However, enriching comprehensibility is not a non-achievable goal. According to the Mainstream Science on Intelligence 1994 [2], intelligence is referred as the atomic set of comprehensibility. The accumulation and calculation of intelligence makes comprehensibility obtainable. It evolves many sciences and effort from different research communities, e.g., Software Engineering, Artificial Intelligence, Computer Science, and different software implement fields, such as Business Data Administration, Financial Economics, and Biology Informatics, etc.

In the research, Business Intelligence Recovery (BIR) aims at obtaining business intelligence to enrich program comprehensibility to software systems.

Business intelligence recovery is a bottom-up reverse engineering process. It elicits information sets from the source program to build a piece of business intelligence oriented program comprehensibility, e.g. a piece of comprehensibility will be “this piece of program is made to count mobile selling; this part of code is made to monitor monthly usage of free minutes; this part of code does plan analysing and it indicates BlackBerry Bold with unlimited email data plan makes the biggest profit in this season” etc., because, despite of the Web-based infrastructure, it contains calculation algorithms, variable/parameter/method names of “plan”, “call”, and “email”, etc.

Nonetheless, the most important idea of business intelligence is that business intelligence as an entity in software system is a non-functional, conceptual, and flexible to the system, but it is very crucial to system evolution, e.g. if a new service of mobile TV broadcasting involves, it does change as much the mobile plan architecture as the code. To do this, the business intelligence about data usage need to be elicited, then the new plan and data service coding will be approved.

The curiosity to the work is our drive. In the research, business intelligence recovery process is partitioned into four sections: partitioning business intelligence base into business intelligence slices, partitioning program source into business intelligence oriented program modules, eliciting business intelligence from a program module, and building formal concept lattice to model the recovered results to present business logic.
Chapter 1 Introduction

1.1 Proposed Research and Overview of Problem

The indication of computing advent is that the software has been widely adopted in business processing to store business data, cope with business questions, and seal business deals, like eBay. But the dynamic business environment urges software system to be frequently adapted in order to cope with new questions, e.g., bridging service to service, migrating business service, and transforming business data, etc. As a matter of fact, the requirement is not merely about understanding code; instead, it is about understanding code from a business point of view. This is the reason why an automatic business intelligence recovery approach is important for the software system in software evolution and business evolution. Actually, software applications of business intelligence have already been developed and applied to support business data extraction, analysis and integration in/from data warehouse. And business intelligence in the data warehouse is daily updating. The circle, however, is not complete, since the business intelligence which is embedded in the legacy system has nearly not been touched. Ideally, an automated business intelligence recovery system with business intelligence base will be the guidebook both for the software system and for the business of industry. Here, our effort is made to answer the question “Can business intelligence be automatically recovered and how?”

Reverse engineering [101] is the main research area in this thesis. It is the traditional path to achieve program comprehension and business rule extraction. The definition of reverse engineering is “it takes the existing software system as substance to extract system abstractions and design information” [45]. Theoretically, a reverse engineering toolkit partitions program into parts via partition rules, abstracts program into abstractions via abstraction rules, and represents program into representations via representation rules, etc. Technically speaking, a reverse engineering toolkit is a rule-based expert system.

However, the very challenging question has been proposed, and the question seems have been talking for ages: “Can reverse engineering be fully automated?” The answer is clearly stated: “It needs to enrich the comprehensibility.” Because lack of comprehensibility is the bottleneck of any rule-based expert system. The issue is not limited to reverse engineering systems. In the real world, taking a software engineer for instance, he/she may be one of the most complex and professional expert systems in the world to deal with program understanding. But being in front of a large piece of
program code with a huge amount of data, or in front of incomplete piece of program code and parts of data, he/she is limited by comprehensibility. Thus, without any question, obtaining comprehensibility makes big sense.

The process of creating comprehensibility is accumulating information from software system, transforming it and representing it in well formed way. In other words only well formed information makes comprehensibility. Business intelligence (BI) is one kind of comprehensibility more oriented at business. Ideally, it would be fully represented in a business intelligence base. The content covers from business intelligence data, business intelligence concepts, BI mathematics model, BI physics model, and business logics. Business intelligence is not limited to any kind of presentation since various requirements of localisation, extraction, analysis could produce new results. It is because business intelligence has different abstraction levels. In this research, the business intelligence has been categorised into five different levels: BI data and concepts, BI algorithms, BI mathematics model, BI physic model, and Business logic.

It is very interesting to discover those business intelligence levels in business intelligence recovery process have sorts of connection with the levels in traditional reverse engineering process. Actually, they are coordinated and complemented to each other. The phenomenon is observed in two points. Firstly, reverse engineering lacks comprehensibility; and it needs business intelligence recovery to obtain business oriented program comprehensibility. Secondly, those two process both start from code and data, and end themselves at high-levels of system abstraction, such as program module, architecture, specifications and business rules logics. This part of discussion will be further presented in Section 1.4 and Section 4.2.

Business intelligence recovery research is not analysing business intelligence directly; it is about obtaining business intelligence from program code and modelling it while performing software reverse engineering. The further analysis of the recovered business intelligence could present business logic, but it will not be quoted in this thesis.

Like the most of other research topic, it is not going to acclaim that the whole work is accomplished. On the contrary, the thesis is trying to specialise the research effort in a specific domain, which also explores whether the research result of business intelligence recovery could contribute to the entire software engineering research community.
1.2 Thesis Objective

The thesis is developed to explore a path to recover business intelligence automatically. It takes reverse engineering as main research area and benefits the mature reverse engineering techniques.

The research on program comprehension and business rule extraction was pioneered by several researchers [33, 74, 91, 106, 108, 116, 122] who gave theoretical and empirical analysis on how programmers understand programs and how business rule could be identified and extracted. Later, researcher explored the possibility of building practical program understanding tools and business rule extraction tools [20, 26, 27, 68, 110], and many prototype systems evolved. Boosted by the emergence of Web, Web-Service, Web-Commerce, Web-Finance, Web-Telecommunication etc. all appear to shift the traditional life style of human life to Web. An undeniable fact is that the software running on network/hardware business platform which is designed to perform business activities to save the labour work, money and time turns to be quantitatively more and more evolved in real business and life. The situation cannot be acclaimed perfect unless the computation environment about business and life could be further understood and engineered, specifically, the advanced program needs to be further designed to sift through trillions of trillions bit data about business and life in the system.

Not surprisingly, most of the fundamental techniques used to support program understanding and business rule extraction can be found in traditional software reverse engineering practices. However, the existing results demonstrate there is blank of extracting business rules in/from an entire software system, because the existing business rule extraction research works are more focusing on program structural and algorithmic level. One another, the reverse engineering methodology is not bringing code information to a certain high level at which code understanding is neck to neck with the needs of business data analysis. Therefore, the continuous emergence of new reverse engineering techniques and a smart combination of fundamental techniques of two fields will keep this area expending.

In this thesis, a new research methodology is introduced. The existing reverse engineering system is viewed as rule-based expert systems. And the bottleneck of these systems is lacking comprehensibility. The research methodology is built on base of reverse engineering study. Business intelligence is developed to be the atomic set of business oriented program comprehensibility. The obtaining and engineering business
intelligence is the target. Five levels of business intelligence are identified in software reverse engineering that can be elicited from software code. A collection of fundamental reverse engineering techniques can support automated business intelligence elicitation and presentation. The problems are identified from the state of the art and the practice of this area and a suite of new solution to these problems are proposed.

1.3 Contributions

Lack of comprehensibility is the barrier to rule-based reverse engineering system. Moreover, it is very hard to have comprehensibility by automated machinery means. In this thesis, the situation has been improved. A method of business intelligence recovery is composed of reverse engineering techniques such as program partition, concept recovery, and formal concept analysis.

As known, traditional program partitioning methods are non-linear; in most cases, the computational effort needed for partitioning a source program will increase exponentially as the size of the source program grows. This constructs a computational barrier for program analysis, since the partitioning of source program must be conducted on-line and the efficiency of any program partition method that partition source program into program modules. This proposed method uses a special kind of heuristic knowledge of programming styles which can partition a source program into business intelligence oriented program modules at linear time-scale.

Business intelligence concepts in context are classified as objects and measures. An object is usually corresponding to a variable style name in a program module; while a measure can find its counterpart as a value in the source code. Although there are also program constructs in source code, such as program comments, where measure and object can occur, these are not reliable sources from which to perform business intelligence concept recovery, since the program comments may not be updated and therefore contain incorrect information about the program. Also program comments are highly unstructured information sources which make it difficult to recover information from them. In the thesis, only procedure names and variable style names are considered to be the candidates from which to recover business intelligence concepts. The first step at this stage is to collect all the procedure names and variable style names from the program module. Once this has been done, business intelligence concept recovery rules can then be applied to each of these names to recover business intelligence concepts.
The recovered business intelligence concepts need to be further analysed. Formal concept analysis leads us to an understanding of an important technique of graphical business intelligence representation namely the line diagrams of concept lattices. Besides, a physics model of substance is more flexible and evolvable. Formal concept analysis is chosen as the representation method because it is featured with two parts. Firstly, it categorised objects with attributes, which enrich the styles of different business intelligence by connections at the same level. Secondly, the relationships among business intelligence at different levels are built in a concept lattice, which represent business logic among business intelligence and connections at different levels.

1.4 Research Method

The research method of business intelligence recovery is built on traditional software reverse engineering study. Firstly, those two processes are compared in steps and at different abstraction levels. It is our conclusion that these two processes are not isolated but literally coordinated. Reverse engineering takes existing software program as substance to extract useful program information to create system abstractions and recover the design model or business rules. Business intelligence recovery takes the program code as substance as well to locate business intelligence information, elicit it with reverse engineering techniques, accumulate it and model the recovered result. Generally, both reverse engineering and business intelligence recovery are bottom-up methods passing through five different abstraction levels: Reverse engineering starts with program code, program structure, and ends at component model, architecture, and requirement; Business intelligence recovery starts at program code and algorithms, and ends at mathematics models, physics models and logics.

The observation draws forth an idea of putting those two processes into a 2-D coordinate system. Thus, the conclusion is that reverse engineering is interpreting program code to have system abstractions, while business intelligence recovery is interpreting the code and data to have business abstraction as well. The issue “it is difficult to have a satisfied result when taking each path individually” matches the observation “it is also difficult to perform software reverse engineering without enriching comprehensibility”. It is easy to understand it, because the research route is localised in a 2-D coordinate system between 0-X and 0-Y, if you are taking path along with X axis or Y axis solo, the route will never arrive at any place between them. That
is the reason why abstraction is so important for performing software reverse engineering.

In fact, research of business intelligence recovery is to find out the crossing points in the coordinate system. And linkage of those points makes the route. It is a complex and starting from scratch work in program understanding. The best way in dealing with a complex task from scratch is a bottom-up approach: decomposing the task, partitioning program source code, recovering concepts, and modelling the results. Therefore, in the thesis, the reverse engineering theory is developed to make it compatible in our research theme: the business intelligence are decomposed into business intelligence slices, and the match between a business intelligence slice and a program module might light-weight the heavy computational effort while performing business intelligence recovery; the program code is partitioned via programming style based program partition rules, because in a common software development project, different programming styles are applied to develop different business functioned program modules; the business intelligence concepts are recovered from a partitioned program module via concept recovery rules, because the naming of variables and value are named after business intelligence concepts and measures; physics models of business intelligence are built by formal concept lattice, because it is mathematically accurate, but flexible and evolvable, and it represents the relationships between different business intelligence and their connections.

1.5 Research Questions

The following criteria are given to judge the success of the research described in this thesis:

- Does the complexity of the program code is considered, e.g. several pieces are integrated into each other?
- How much confidence can Auto-BIR promise in the process?
- Is the recovered business intelligence concept confident to perform software reengineering?
- How much business intelligence in percentage can be recovered from a software system?
• Crossing the business intelligence base, can confidence of business intelligence recovery be high enough to match with new business intelligence in daily business practice?

• Is this approach feasible for realisation? For example, is it possible to build a practical tool based on the approach?

1.6 Thesis Organisation

The thesis is organised as follows:

Chapter 1 gives the introductions of proposed research and overview of problem, thesis objective, contribution, research method, criteria for success.

Chapter 2 provides an overview of research background in software evolution, reverse engineering, program comprehension, and business rule extraction.

Chapter 3 investigates the existing related work, data mining, domain knowledge, and data warehouse.

Chapter 4 firstly discusses the proposed research method to deal with the research questions. Secondly, it introduces the prefer techniques the thesis is going to develop or use in business intelligence recovery.

Chapter 5 explores the method of partitioning business intelligence which can later be used in conjunction with program partition, concept recovery, and formal concept analysis to establish the linkage between source code and business intelligence.

Chapter 6 introduces programming style based program partition method. Partition rules to conduct program modules decomposition from program source are developed.

Chapter 7 introduces business intelligence concept recovery method based on name recovery rules.

Chapter 8 introduces formal concept analysis to further represent recovered business intelligence into graphic presentations.

Chapter 9 describes realisation of the proposed approach to deal with case studies.
Chapter 10 discusses the approach and the supporting tool according to a set of criteria. Conclusion is drawn based on this discussion, and prospective further work is also discussed.

Appendix A is the code of case study about program partition.

Appendix B is the code of case study about concept recovery.

Appendix C is the list of publications by author.
Chapter 2

Background

The background of business intelligence recovery research is concluded as software evolution, software reverse engineering, program comprehension, and business rule extraction.

2.1 Software Evolution

The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity. The term software evolution lacks a standard definition, but some researchers and practitioners use it as a preferable substitute for maintenance.

Pioneering work was undertaken by Lehman, who carried out empirical experiments on OS360 using a sequence of releases [90]. This has set a good precedent for the field: very small programs do not have maintenance problems, and research results must scale up to industrial applications for them to be useful [100], so that research in maintenance needs to be undertaken in collaboration with industry. Some of the practical consequences of approaches that work in the realities of an industrial context are reported by Sneed [119].

Research in software maintenance has been undertaken in seven broad areas:

- System dynamics, to model the software as it changes over time, in order better to understand the underlying mechanisms.

- Maintenance processes; defining, measuring, improving, risk analysis, quality assurance.

- Studies of software change; impact analysis [15], change propagation.
- Products, linking software attributes to maintainability (from architecture to identifier naming conventions); higher levels of software abstraction.

- Program comprehension methods and tools, to link attributes to better cognitive understanding.

- High level management, business and people issues; business models such as outsourcing and applications management.

- Legacy and reverse engineering, to recover a software asset that has become very hard (expensive) to maintain.

- Validation, ensuring that the software changes work as required, and the unchanged parts have not become less dependable.

### Importance of Maintenance

A very widely cited survey study by Lientz and Swanson in the late 1970s [93], and repeated by others in different domains, exposed the very high fraction of life-cycle costs that were being expended on maintenance. Lientz and Swanson categorised maintenance activities into four classes:

- Adaptive – changes in the software environment
- Perfective – new user requirements
- Corrective – fixing errors
- Preventive – prevent problems in the future

Of these, the survey showed that around 75% of the maintenance effort was on the first two styles, and error correction consumed about 21%. Many subsequent studies suggest a similar magnitude of the problem. These studies show that the incorporation of new user requirements is the core problem for software evolution and maintenance [16].

If changes can be anticipated at design time, they can be built in by some form of parameterisation. The fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even conceive of. So software maintenance is important because (i) it consumes a large part of the overall lifecycle costs (ii) the inability to change software quickly and reliably means that business opportunities are lost. These are enduring problems, so that the profile of maintenance research is likely to increase over the next ten years.
What is software maintenance?

Despite the large expenditure, little is known about the empirical nature of software maintenance, in terms of its effect on the artifacts, on the process and on the software engineers and users. The first vista in the research landscape is therefore:

- To gain more empirical information about the nature of software maintenance, in terms of its effect on the software itself, on processes, on organisations and people. What actually happens from release to release? For example, it was reported that a feature set during each iteration may change by 30% or more, as a direct result of the team learning process during the iteration. It is described that variations in the frequency of the changes appear in a long lived system. Is it possible to describe the changes in terms of a set of basic operations? More empirical work is crucial to inform progress on better maintenance processes.

- To express such understanding in terms of predictive models which can be validated through experiment. The models may inform both the technical and business facets of maintenance (e.g. risk models). For example, Lehman is using feedback control models in the FEAST project [90].

- To explore and formalise the relationships between technology and business models (for example, the implications of outsourcing and applications management, or the technical and business views of legacy software management).

- To understand how such models may be exploited in an industrial context (for example, in cost estimation). This work should lead to better metrics.

- To establish accepted evaluation procedures for assessing new developments and processes, in terms of the implications for maintenance, especially in an industrial context on large scale applications.

The final point can be generalised: often, new technologies are proposed and introduced without consideration of what happens when the software has to be changed. If such innovations are to be exploited successfully, the full lifecycle needs to be addressed, not just the initial development. For example, object oriented technology was considered to be ‘the solution to software maintenance; empirical evidence is now showing that OO is creating its own new maintenance problems, and has to be used with care (e.g. by keeping inheritance under control) to ensure that maintenance is not even more difficult than for traditional systems. Recent technologies such as agents, components, graphical
user interfaces, and modern ideas of logical, constraint, real-time and concurrent programming and so on need to be explored from a maintenance perspective [8].

Better understanding via such models should help researchers devise a much better definition of maintainability; currently this is a very poorly defined term of very limited use in industry.

A major challenge for the research community is to develop a good theoretical understanding and underpinning for maintenance and evolution [25], which scales to industrial applications. Most computer science research has been of benefit to the initial development of software. Style theory and configuration management have in different ways made major contributions to maintenance. Many others claim to do so, but reliable empirical evidence is lacking.

2.2 Reverse Engineering

The notion of computers automatically finding useful information is an exciting and promising aspect of just about any application intended to be of practical use [49]. A decade ago, following up on the successes of the early CASE tools, Chikofsky and Cross introduced taxonomy for reverse engineering and design recovery [45]. They defined reverse engineering to be “analysing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information.”

Over the past ten years, researchers have produced a number of capabilities to explore, manipulate, analyse, summarise, hyperlink, synthesise, componentise, and visualise software artifacts. These capabilities include documentation in many forms and intermediate representations for code, data, and architecture. Many reverse engineering tools focus on extracting the structure of a legacy system with the goal of transferring this information into the minds of the software engineers trying to reengineer or reuse it. In corporate settings, reverse engineering tools still have a long way to go before becoming an effective and integral part of the standard toolset that a typical software engineer uses day-to-day.
Chapter 2 Background

**Code Reverse Engineering**

In current research and practice, the focus of both forward and reverse engineering is at the code level [11]. Forward engineering processes are geared toward producing quality code. The importance of the code level is underscored in legacy systems where important business rules are actually buried in the code [74]. During the evolution of software, change is applied to the source code, to add function, fix defects, and enhance quality. In systems with poor documentation, the code is the only reliable source of information about the system. As a result, the process of reverse engineering has focused on understanding the code.

Over the past ten years, reverse engineering research has produced a number of capabilities for analysing code, including subsystem decomposition [19, 72], concept synthesis [9], design, program and change pattern matching [22, 32, 51, 66], program slicing and dicing [130], analysis of static and dynamic dependencies [9, 67], object-oriented metrics [24], and software exploration and visualisation [58]. In general, these analyses have been successful at treating the software at the syntactic level to address specific information needs and to span relatively narrow information gaps.

However, the code does not contain all the information that is needed. Typically, knowledge about architecture and design tradeoffs, engineering constraints, and the application domain only exists in the minds of the software engineers [6]. Over time, memories fade, people leave, documents decay, and complexity increases [46]. Consequently, an understanding gap arises between known, useful information and the required information needed to enable software change. At some point, the gap may become too wide to be easily spanned by the syntactic, semantic, and dynamic analyses provided by traditional programming tools.

Thus when being focus only at the low levels of abstraction, software engineers miss the big picture behind the evolution of a software system [43]. There is a need to focus future research on the more significant levels of the business processes and the software architecture. For example, knowledge of software architecture from multiple user perspectives is needed to make large scale, structural changes [73], and the capability to perform architecture reconstruction is becoming increasingly important [6, 62]. Developers need information about the impacts of potential changes. Managers need information to assign and coordinate their personnel. If the information to create this knowledge can be maintained continuously, a software engineer could generate the required perspectives on a continuous basis without costly reverse engineering efforts.
Because such analyses are rarely performed today, current system evolution efforts often experience a time of crisis at which the gap between desired information and available information becomes critical. At that point reverse engineering techniques are inserted in a “big bang” attempt to regain useful understanding and insight. The structural, functional, and behavioural code analyses [51], however, require intensive human input to construct from scratch. These analyses are difficult to interpret, and are costly efforts with high risk.

**Continuous Program Understanding**

To avoid a crisis, it is important to address information needs more effectively throughout the software lifecycle. Software engineers need to better support the forward and backward traceability of software artifacts. For example, in the forward direction, given a design module, it is important to be able to obtain the code elements that implement it. In the backward direction, given a source or object file, an engineer needs to be able to obtain the business rule to which it contributes. In addition it is important to determine when it is most appropriate to focus the analysis at different levels of abstraction [15, 44].

For understanding purposes, traceability is especially important. Software engineers need to be able to take a pattern of change, such as updating a tax law, and map this law explicitly into software structures. Part of program comprehension is to reconstruct mappings between the application and implementation domains [19]. Thus, to ease long-term understanding, these mappings must be made explicit, recorded, reused, and updated continuously. The vision is that reverse engineering would be applied incrementally, in small loops with forward engineering, rather than as a desperate attempt at resurrecting a poorly understood system.

Several research issues, formulated as questions, need to be addressed to enable this capability for “continuous program understanding” [72].

- What are the long-term information needs of a software system?
- What patterns of change do software systems undergo?
- What mappings need to be explicitly recorded?
- What kind of software repository could represent the required information?
- What are the requirements of tool support to produce and manipulate the mappings?
- How can this support coexist with traditional, code dominated tools, users, and processes?

**Reverse Engineering Process**

In addition to an emphasis on “continuous program understanding,” it is important to focus efforts on a better definition of the reverse engineering process. Reverse engineering has typically been performed in an ad hoc manner. To address the technical issues effectively, the process must become more mature and repeatable, and more of its elements need to be supported by automated tools.

For example, a developer might require the software components that contribute to a specific system responsibility. The subsystem view to present this information should not require tedious manual manipulation. Instead, the mapping between responsibility and components should be consulted and a script would then generate the required view, with the option for minor, personal customization by the user.

Such a script is an instance of a reverse engineering pattern [72], a commonly used task or solution to produce understanding in a particular situation. By cataloguing such patterns and automating them through tool support, engineers would improve the maturity of the reverse engineering process.

Thus, the insights of the SEI Capability Maturity Model (CMM) framework [38, 42] ought to apply to reverse engineering as well as forward engineering. Future research ought to focus on ways to make the process of reverse engineering more repeatable, defined, managed, and optimized.

Increased process maturity would enable better assessment of the risks, costs, and economics of reengineering activities. With poorly understood processes, the success of a reengineering project rests solely on the ingenuity of the people involved – ingenuity that disappears when the project ends. For evolving large software systems over long periods of time, an appreciation of both product and process improvement is needed.
Research Direction

In summary, for future research in reverse engineering, it is important to understand software at various levels of abstraction and maintain mappings between these levels. Categories of information, tool, and process requirements are needed as a prelude to enabling continuous program understanding [104]. Useful reverse engineering processes need to be identified and better supported, as an important step to make the discipline of reengineering more rational. Reverse engineering tools and processes need to evolve with the development environment that stresses components, the Web, and distributed systems [1].

2.3 Program Comprehension

Challenges in understanding programs are all too familiar from even before the days of the first software engineering workshop [4]. Since that time, the field of program comprehension as a research discipline has evolved considerably. The goal of the community is to build an understanding of these challenges, with the ultimate objective of developing more effective tools and methods. From these early days we have come to accept that there is no silver bullet [5], but the community has made advances which have helped software engineers tackle important problems such as the Y2K problem.

Software engineers now have a wide variety of theories that provide rich explanations of how programmers understand programs and can provide advice on how program comprehension tools and methods may be improved. In response to these theories, and in some cases in parallel to the theory development, many powerful tools and innovative software processes have evolved to improve comprehension activities.

The field of program comprehension research has been rich and varied, with various shifts in paradigms and research cultures during the last few decades. A multitude of differences in program characteristics, programmer ability and software tasks have led to many diverse theories, research methods and tools. In this thesis, it provides a review of this work in an attempt to create a landscape of program comprehension research. Such a view emphasises how the theories and tools are related and should reveal if parts of the landscape have not received much attention. This review, combined with an excursion to newer areas of software engineering theory and practice, directs us to specific areas for the future of program comprehension research.
A Review of Cognitive Theories

Francois Détienne’s book, “Software Design - Cognitive Aspects” [6], provides an excellent review of the history of cognitive models and related experiments over the past twenty or so years. The research delves back to a time, in the early 1970’s, when experiments were done without theoretical frameworks to guide the evaluations. Consequently, it was neither possible to understand nor to explain to others why one tool might be superior to other tools.

The lack of theories was recognized as being problematic. As the field of program comprehension matured, research methods and theories were borrowed from other areas of research, such as text comprehension, problem solving and education. Using these theoretical underpinnings, cognitive theories about how programmers understand programs and how tools could support comprehension were developed.

Concepts and Terminology

A mental model describes a developer's mental representation of the program to be understood whereas a cognitive model describes the cognitive processes and temporary information structures in the programmer’s head that are used to form the mental model. Cognitive support assists cognitive tasks such as thinking or reasoning [91].

Programming plans are generic fragments of code that represent typical scenarios in programming. For example, a sorting program will contain a loop which compares two numbers in each iteration [4].

Beacons are recognizable, familiar features in the code that act as cues to the presence of certain structures [5, 47, 48, 50, 61, 69, 133]. Rules of programming discourse capture the conventions of programming, such as coding standards and algorithm implementations [1].

Top-Down Comprehension

Brooks theorizes that programmers understand a completed program in a top-down manner where the comprehension process is one of reconstructing knowledge about the domain of the program and mapping this knowledge to the source code [86, 107]. The process starts with a hypothesis about the general nature of the program. This initial hypothesis is then refined in a hierarchical fashion by forming subsidiary hypotheses.
Subsidiary hypotheses are refined and evaluated in a depth-first manner. The verification (or rejection) of hypotheses depends heavily on the absence or presence of beacons [84].

Soloway and Ehrlich [122] observed that top-down understanding is used when the code or style of code is familiar. They observed that expert programmers use beacons, programming plans and rules of programming discourse to decompose goals and plans into lower level plans. They noted that delocalised plans complicate program comprehension.

**Bottom-Up Comprehension**

The bottom-up theory of program comprehension assumes that programmers first read code statements and then mentally chunk or group these statements into higher level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is attained [73]. Shneiderman and Mayer's cognitive framework differentiates between syntactic and semantic knowledge of programs [116]. Syntactic knowledge is language dependent and concerns the statements and basic units in a program. Semantic knowledge is language independent and is built in progressive layers until a mental model is formed which describes the application domain.

Pennington also describes a bottom-up model [106]. She observed that programmers first develop a control flow abstraction of the program which captures the sequence of operations in the program. This model is referred to as the program model and is developed through the chunking of microstructures in the text (statements, control constructs and relationships) into macrostructures (text structure abstractions) and by cross-referencing these structures. Once the program model has been fully assimilated, the situation model is developed. The situation model encompasses knowledge about data-flow abstractions and functional abstractions (the program goal hierarchy).

**Opportunistic and Systematic Strategies**

Stumme et al. observed programmers enhancing a personnel database program [121]. They observed that programmers either systematically read the code in detail, tracing through the control-flow and data-flow abstractions in the program to gain a global understanding of the program, or that they take an as needed approach, focusing only on the code relating to a particular task at hand. Subjects using a systematic strategy
acquired both static knowledge (information about the structure of the program) and causal knowledge (interactions between components in the program when it is executed). This enabled them to form a mental model of the program. However, those using the as-needed approach only acquired static knowledge resulting in a weaker mental model of how the program worked. More errors occurred since the programmers failed to recognize causal interactions between components in the program.

Letovsky observed activities called inquiries [91]. An inquiry may consist of a programmer asking a question, conjecturing an answer, and then searching through the code and documentation to verify the answer. Inquiry episodes often occur as a result of delocalized plans.

**The Integrated Metamod**

The Integrated Metamodel, developed by von Mayrhauser and Vans, builds on the previous three models as well as the knowledge based model by Letovsky [91]. Their model consists of four major components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform a comprehension process:

- The top-down (domain) model is usually invoked and developed using an as-needed strategy, when the programming language or code is familiar. It incorporates domain knowledge as a starting point for formulating hypotheses.

- The program model may be invoked when the code and application is completely unfamiliar. The program model is a control-flow abstraction.

- The situation model describes data-flow and functional abstractions in the program. It may be developed after a partial program model is formed using systematic or opportunistic strategies.

- The knowledge base consists of information needed to build these three cognitive models. It represents the programmer's current knowledge and is used to store new and inferred knowledge.

Understanding is formed at several levels of abstraction simultaneously by switching between the three comprehension processes.
Program Characteristics

Programs that are carefully designed and well documented will be easier to understand, change or reuse in the future. Pennington's experiments showed that the choice of language has an effect on comprehension processes. COBOL programmers consistently fared better at answering questions related to data-flow than FORTRAN programmers, and FORTRAN programmers consistently fared better than COBOL programmers for control-flow questions.

Object-Oriented (OO) programs are often seen as a more natural fit to problems in the real world because of ‘is-a’ and ‘is-part-of’ relationships in a class hierarchy and structure, but others argue that objects do not always map easily to real world problems [56]. In OO programs, abstractions are achieved through encapsulation and polymorphism. Message-passing is used for communication between class methods and hence programming plans are dispersed (i.e. scattered) throughout classes in an OO program.

Individual Programmer Differences

There are many individual characteristics that will impact how a programmer tackles a comprehension task. These differences also impact the requirements for a supporting tool. There is a huge disparity in programmer ability and creativity which cannot be measured simply by their experience.

Vessey presents an exploratory study to investigate expert and novice debugging processes [18]. She classified programmers as expert or novice based on their ability to chunk effectively. She notes that experts used breadth-first approaches and at the same time were able to adopt a system view of the problem area, whereas novices used breadth-first and depth-first approaches but were unable to think in system terms. Dijkstra also notes that experts make more use of external devices as memory aids [51, 52]. Experts tend to reason about programs according to both functional and object-oriented relationships and consider the algorithm, whereas novices tend to focus on objects.
2.4 Business Rule Extraction

Legacy software systems usually contain business rules (e.g., billing regulations, business decisions) that have been coded into the system over years. Telephone billing rates are an example of business rules. Every time a customer makes a call, the billing program of the phone company is responsible to keep track of all the relevant information, and make appropriate charges according to billing rate.

Business rules are subject to changes as the markets and technology change. When an update occurs in the business practices or rules, the corresponding segments of the software must be changed. In course of time and numerous update phases, both the software and the text documents become larger and increasingly difficult to understand and maintain. The software programmers gradually tend to focus only on the software and lose confidence in the text documents. This creates a maintenance problem.

Business Rule Extraction Criteria

The requirements of business rule extraction are summarised as below:

- Faithful representation: Any business rules extracted from the code must reflect the true state of the software. This is the most important criterion, and if this criterion conflicts with other criteria then this criterion should supersede. The reason for critical nature of this criterion is that programmers trust the code more than they trust the associate documents.

- Multiple Representation and Hierarchical abstraction: Different people require different representations of business rules. For example, a programmer would like to have business rules represented as code segments, while managers may prefer decision tables, decision trees, and structured charts. Business rules should be represented in a hierarchical manner. Business rules are often rather complex because they must meet various constraints, such as legal, marketing and technology constraints. It will be extremely difficult to trace business rules without some form of abstractions or decomposition.

- Domain-Specific Business Policies: Most software maintainers prefer business rules to be expressed in domain vocabulary that represent domain concepts in the application. The tool should provide a means to identify important concepts
including data and algorithms related to business rules from other supporting program entities.

- Human-assisted Automation: As legacy systems are huge, it is extremely difficult, if not impossible, to devise an automatic tool for business rule extraction [30]. The software maintainer prefer to have an interactive tool that allow them to extract business rules, simplify their representations, and provide linkage to the code, rather than providing a black-box tool that generate business rules automatically.

- Maintenance Tool: Business rules extracted should be useful in helping other software maintenance activities. The rules extracted should be maintained together with the software using the same tool. The tool should provide the mapping from any business rule to its corresponding code segments that implement the rule. This capability will allow the software personnel to focus attention to only those segments (and functions) of the software that are relevant to a particular update operation.

**Overall Approach to Business Rule Extraction**

A business rule is usually centred on certain data, either input or output data. For example, a simple business rule can be attached either to the output variable Profit, two input variables, or both in the text documents. Thus, a data-centred approach [77] is taken to perform business rule extraction. Recall that a business rule is a form $O = F(I)$, and we can first identify either $I$, $O$, or both to start business rule extraction. For example, if it is wished to extract business rule that deals with 1-800 calls in phone billing program, we may first identify the variables that represent the 1-800 phone charges, and then extract the code segments that either directly or indirectly manipulate these variables to obtain the business rules related to 1-800 changes. This is the data-centred approach to business rule extraction.

Thus, the first step is to identify important business variables from the code that can be used to express business rules. In a typical large business application, it can be found that hundreds of thousands of code variables, but only a subset of them is suitable for expressing business rules, i.e., those that have domain concepts in the application. These variables are called domain variables, and it needs mechanism to identify those domain variables from other code variables.
2.5 Summary

Automating business intelligence recovery involves techniques in data mining, domain knowledge recovery and data warehouse. These techniques laid a foundation for business intelligence recovery. But the automating business intelligence recovery means to explore a path to have business intelligence directly from software system with reverse engineering techniques. Based on this, it is found that existing research work on the area of business intelligence elicitation from source code is mainly capable of extracting information up to program structural and algorithmic level. For the few research works on assigning business intelligence concept to source code, heavy-weight techniques are often used to automate the process. As a result, these methods are not capable of dealing with large-scale commercial software evolution tasks in the real world efficiently. To address this issue, a new solution is proposed to break down both business intelligence and source program into smaller pieces so that the business intelligence recovery task can be carried out in smaller scale. In particular, monolithic business intelligence are partitioned into business intelligence slice a source program is partitioned into business intelligence oriented program modules; business intelligence slices are recovered from program modules; and recovered business intelligence slices are further modelled with formal concept analysis to present the business logic.

In this chapter, the backgrounds of business intelligence recovery research are presented. Software evolution is one kind of activity which is driven by business evolution. The maintenance and re-engineering activities are performed to fulfil needs form business requirements. Reverse engineering is the kernel methodology to realise evolution of an existing software system. It is the first step to have system abstractions and requirements. Program comprehension is realised by reverse engineering techniques and built on base of programming knowledge. Business rule extraction aims at understanding business rules embedded in a software system. These background literature reviews are investigated to interpret business intelligence recovery theory. However, the research effort of business intelligence recovery is to open a new chapter in software engineering research. Business intelligence recovery will be further studied in the following chapters.
Chapter 3

Related Research

Business intelligence recovery methodology is built on traditional reverse engineering research. However, it is tightly related to the other three research fields: data mining, knowledge engineering, and data warehousing. The following sections discuss each of these three research fields and their related topics with business intelligence recovery.

3.1 Data Mining

Data mining is the process of extracting hidden patterns from data. As more data is gathered - the amount of data doubling every three years [95] - data mining becomes an increasingly important tool to transform this data into knowledge. It is commonly used in a wide range of applications such as, marketing, fraud detection and scientific discovery. Data mining can be applied to data sets of any size, and while it can discover hidden patterns, it cannot discover patterns which are not already present in the data set.

Backgrounds

Traditionally, business analysts have performed the task of extracting useful information from recorded data, but the increasing volume of data in modern business and science calls for computer-based approaches. As data sets have grown in size and complexity, there has been a shift away from direct hands-on data analysis toward indirect, automatic data analysis using more complex and sophisticated tools. Major improvements in computer technology have aided data collection. However, the captured data needs to be converted into information and knowledge to become useful. Data mining is the entire process of applying computer-based methodology, including new techniques for knowledge discovery, to data [98].

Data mining identifies trends within data that go beyond simple analysis. Through the use of sophisticated algorithms, non-statistician users have the opportunity to identify
key attributes of business processes and target opportunities. However, abdicating control of this process from the statistician to the machine may result in false-positives or no useful results at all.

Although data mining is a relatively new term, the technology is not. For many years, businesses have used powerful computers to sift through volumes of data such as supermarket scanner data to produce market research reports (although reporting is not always considered to be data mining). Continuous innovations in computer processing power, disk storage, and statistical software are dramatically increasing the accuracy and usefulness of data analysis.

The term data mining is often used to apply to the two separate processes of knowledge discovery and prediction. Knowledge discovery provides explicit information that has a readable form and can be understood by a user (e.g., association rule mining). Forecasting, or predictive modelling provides predictions of future events and may be transparent and readable in some approaches (e.g., rule-based systems) and opaque in others such as neural networks. Moreover, some data-mining systems such as neural networks are inherently geared towards prediction and pattern recognition, rather than knowledge discovery.

Metadata, or data about a given data set, are often expressed in a condensed data-minable format, or one that facilitates the practice of data mining. Common examples include executive summaries and scientific abstracts.

Data mining relies on the use of real world data. These data are extremely vulnerable to co-linearity precisely because data from the real world may have unknown interrelations. An unavoidable weakness of data mining is that the critical data that may expose any relationship might have never been observed. Alternative approaches using an experiment-based approach such as Choice Modelling for human-generated data may be used. Inherent correlations are either controlled for or removed altogether through the construction of an experimental design.

Recently, there were some efforts to define a standard for data mining, for example the CRISP-DM standard for analysis processes or the Java Data-Mining Standard. Independent of these standardization efforts, freely available open-source software systems like RapidMiner and Weka have become an informal standard for defining data-mining processes.
Data are any facts, numbers, or text that can be processed by a computer. Today, organisations are accumulating vast and growing amounts of data in different formats and different databases. This includes:

Operational or transactional data such as, sales, cost, inventory, payroll, and accounting

Nonoperational data, such as industry sales, forecast data, and macro economic data

Meta data – data about the data itself, such as logical database design or data dictionary definitions

**Common Data Mining Tasks**

Data mining is commonly used to perform these three tasks [127]:

- **Classification** - Arranges the data into predefined groups. For example an email program might attempt to classify an email as legitimate or spam. Common algorithms include Nearest Neighbour, Naive Bayes classifier and Neural Network.

- **Clustering** - Is like classification but the groups are not predefined, so the algorithm will try to group similar items together [39].

- **Regression** - Attempts to find a function which models the data with the least error. A common method is to use Genetic Programming.

**3.2 Domain Knowledge**

Most generally, domain knowledge is the knowledge which is valid and directly used for a pre-selected domain of human endeavour or an autonomous computer activity. Specialists and experts use and develop their own domain knowledge. If the concept domain knowledge or domain expert is used, a specific domain will be emphasised which is an object of the discourse/interest/problem.

More particular, in software engineering, domain knowledge is knowledge about the environment in which the target system operates, for example, software agents. Domain knowledge is important, because it usually must be learned from software users in the domain (as domain specialists/experts), rather than from software developers.
Communicating between end-users and software developers is often difficult. They must find a common language to communicate in [70]. Developing enough shared vocabulary to communicate can often take a while.

Expert’s domain knowledge (frequently informal and ill structured) is transformed in computer programs and active data, for example in a set of rules in knowledge bases, by knowledge engineers.

**Knowledge Engineering**

Knowledge engineering (KE) has been defined by Feigenbaum, and McCorduck in 1983 as follows: “KE is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise” [57].

At present, it refers to the building, maintaining and development of knowledge-based systems [83]. It has a great deal in common with software engineering, and is used in many computer science domains such as artificial intelligence [102, 112], including databases, data mining, expert systems, decision support systems and geographic information systems. Knowledge engineering is also related to mathematical logic, as well as strongly involved in cognitive science and socio-cognitive engineering where the knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.

Various activities of KE specific for the development of a knowledge-based system:

- Assessment of the problem
- Development of a knowledge-based system shell/structure
- Acquisition and structuring of the related information, knowledge and specific preferences (IPK model)
- Implementation of the structured knowledge into knowledge bases
- Testing and validation of the inserted knowledge
- Integration and maintenance of the system
- Revision and evaluation of the system.

Being still more art than engineering, KE is not as neat as the above list in practice. The phases overlap, the process might be iterative, and many challenges could appear.
Recently, emerges meta-knowledge engineering as a new formal systemic approach to the development of a unified knowledge and intelligence theory.

**Knowledge Engineering Principles**

Since the mid-1980s, knowledge engineers have developed a number of principles, methods and tools that have considerably improved the process of knowledge acquisition and ordering. Some of the key principles are summarized as follows:

- Knowledge engineers acknowledge that there are different types of knowledge, and that the right approach and technique should be used for the knowledge required.
- Knowledge engineers acknowledge that there are different types of experts and expertise, such that methods should be chosen appropriately.
- Knowledge engineers recognize that there are different ways of representing knowledge, which can aid the acquisition, validation and re-use of knowledge.
- Knowledge engineers recognize that there are different ways of using knowledge, so that the acquisition process can be guided by the project aims (goal-oriented).
- Knowledge engineers use structured methods to increase the efficiency of the acquisition process.

**Metaknowledge**

Metaknowledge or meta-knowledge is knowledge about a preselected knowledge. It is relative concept, it means, if K is a knowledge about a domain D, and K’ is a knowledge about D’, then if D’= K then K’ is a meta-knowledge. K can be analysed and used from different points of view therefore we may have different meta-knowledge related to the same domain. From the systemic perspective, generalized metaknowledge is domain-independent knowledge which performs or enables operations on every another more or less specific domain-dependent knowledge in different domains/areas of human activities [103].

For the reason of different definitions of knowledge in the subject matter literature, meta-information is or is not included in meta-knowledge. Detailed cognitive, systemic and epistemic study of human knowledge requires a distinguishing of these concepts.
But in the common language knowledge includes information, and, for example, bibliographic data are considered as a meta-knowledge.

Meta-knowledge is a fundamental conceptual instrument in such research and scientific domains as, knowledge engineering, knowledge management, and others dealing with study and operations on knowledge, seen as a unified object/entities, abstracted from local conceptualizations and terminologies. Examples of the first-level individual meta-knowledge are methods of planning, modelling, learning and every modification of domain knowledge. The procedures, methodologies and strategies of teaching, coordination of e-learning courses are individual meta-meta-knowledge of an intelligent entity (a person, organisation or society). Of course, universal meta-knowledge frameworks have to be valid for the organisation of meta-levels of individual meta-knowledge.

### 3.3 Data Warehouse

Data warehouse is a repository of an organisation's electronically stored data. Data warehouses are designed to facilitate reporting and analysis.

This classic or general definition of the data warehouse focuses on data storage [135]. However, the means to retrieve and analyse data, to extract, transform and load data, and to manage the data dictionary are also considered essential components of a data warehousing system. Many references to data warehousing use this broader context. Thus, an expanded definition for data warehousing includes business intelligence tools, tools to extract, transform, and load data into the repository, and tools to manage and retrieve metadata.

In contrast to data warehouses are operational systems which perform day-to-day transaction processing. The process of transforming data into information and making it available to the user in a timely enough manner to make a difference is known as data warehousing.

**Background**

The concept of data warehousing dates back to the late 1980s when IBM researchers Barry Devlin and Paul Murphy developed the “business data warehouse”. In essence, the data warehousing concept was intended to provide an architectural model for the
flow of data from operational systems to decision support environments. The concept attempted to address the various problems associated with this flow - mainly, the high costs associated with it. In the absence of a data warehousing architecture, an enormous amount of redundancy of information was required to support the multiple decision support environments that usually existed. In larger corporations it was typical for multiple decision support environments to operate independently. Each environment served different users but often required much of the same data. The process of gathering, cleaning and integrating data from various sources, usually long existing operational systems (usually referred to as legacy systems), was typically in part replicated for each environment. Moreover, the operational systems were frequently re-examined as new decision support requirements emerged [18]. Often new requirements necessitated gathering, cleaning and integrating new data from the operational systems that were logically related to prior gathered data.

Based on analogies with real-life warehouses, data warehouses were intended as large-scale collection/storage/staging areas for corporate data. Data could be retrieved from one central point or data could be distributed to "retail stores" or "data marts" which were tailored for ready access by users.

Key developments in early years of data warehousing were:

- **1960s** - General Mills and Dartmouth College, in a joint research project, develop the terms dimensions and facts.
- **1970s** - ACNielsen and IRI provide dimensional data marts for retail sales.
- **1983** - Teradata introduces a database management system specifically designed for decision support.
- **1988** - Barry Devlin and Paul Murphy publish the article An architecture for a business and information systems in IBM Systems Journal where they introduce the term “business data warehouse”.
- **1991** - Prism Solutions introduces Prism Warehouse Manager, software for developing a data warehouse.
- **1991** - Bill Inmon publishes the book Building the Data Warehouse.
• 1995 - The Data Warehousing Institute, a for-profit organisation that promotes data warehousing, is founded.

• 1996 - Ralph Kimball publishes the book The Data Warehouse Toolkit.

• 1997 - Oracle 8, with support for star queries, is released

The Future of Data Warehousing

Data warehousing, like any technology niche, has a history of innovations that did not receive market acceptance. A 2007 Gartner Group paper predicted the following technologies could be disruptive to the business intelligence market [114]:

• Service Oriented Architecture

• Search capabilities integrated into reporting and analysis technology

• Software as a Service

• Analytic tools that work in memory

• Visualisation

Another prediction is that data warehouse performance will continue to be improved by use of data warehouse appliances, many of which incorporate the developments in the aforementioned Gartner Group report.

Data Warehouse Architecture

Architecture, in the context of an organisation's data warehousing efforts, is a conceptualisation of how the data warehouse is built. There is no right or wrong architecture. The worthiness of the architecture can be judged in how the conceptualization aids in the building, maintenance, and usage of the data warehouse.

One possible simple conceptualization of data warehouse architecture consists of the following interconnected layers:

• Operational database layer: The source data for the data warehouse - An organisation’s EIS systems fall into this layer.

• Informational access layer: The data accessed for reporting and analysing and the tools for reporting and analysing data - Business intelligence tools fall into
this layer. And the Inmon-Kimball differences about design methodology, discussed later in this article, have to do with this layer.

- Data access layer: The interface between the operational and informational access layer - Tools to extract, transform, load data into the warehouse fall into this layer.

- Metadata layer: The data directory - This is often usually more detailed than an operational system data directory. There are dictionaries for the entire warehouse and sometimes dictionaries for the data that can be accessed by a particular reporting and analysis tool.

**Top-Down versus Bottom-Up Design Methodologies**

**Bottom-up design:** Ralph Kimball, a well-known author on data warehousing, is a proponent of an approach frequently considered as bottom-up, to data warehouse design. In the so-called bottom-up approach data marts are first created to provide reporting and analytical capabilities for specific business processes. Data marts contain atomic data and, if necessary, summarised data. These data marts can eventually be union together to create a comprehensive data warehouse. The combination of data marts is managed through the implementation of what Kimball calls “a data warehouse bus architecture” [75].

Business value can be returned as quickly as the first data marts can be created. Maintaining tight management over the data warehouse bus architecture is fundamental to maintaining the integrity of the data warehouse. The most important management task is making sure dimensions among data marts are consistent. In Kimball words, this means that the dimensions “conform”.

**Top-down design:** Bill Inmon, one of the first authors on the subject of data warehousing, has defined a data warehouse as a centralized repository for the entire enterprise [75]. Inmon is one of the leading proponents of the top-down approach to data warehouse design, in which the data warehouse is designed using a normalized enterprise data model. “Atomic” data, that is, data at the lowest level of detail, are stored in the data warehouse. Dimensional data marts containing data needed for specific business processes or specific departments are created from the data warehouse. In the Inmon vision the data warehouse is at the centre of the “Corporate Information Factory” (CIF), which provides a logical framework for delivering business intelligence
(BI) and business management capabilities. The CIF is driven by data provided from business operations.

Inmon states that the data warehouse is:

- **Subject-oriented**: The data in the data warehouse is organised so that all the data elements relating to the same real-world event or object are linked together.
- **Time-variant**: The changes to the data in the data warehouse are tracked and recorded so that reports can be produced showing changes over time.
- **Non-volatile**: Data in the data warehouse is never over-written or deleted - once committed, the data is static, read-only, and retained for future reporting.
- **Integrated**: The data warehouse contains data from most or all of an organisation's operational systems and this data is made consistent.

The top-down design methodology generates highly consistent dimensional views of data across data marts since all data marts are loaded from the centralized repository. Top-down design has also proven to be robust against business changes. Generating new dimensional data marts against the data stored in the data warehouse is a relatively simple task. The main disadvantage to the top-down methodology is that it represents a very large project with a very broad scope. The up-front cost for implementing a data warehouse using the top-down methodology is significant, and the duration of time from the start of project to the point that end users experience initial benefits can be substantial. In addition, the top-down methodology can be inflexible and unresponsive to changing departmental needs during the implementation phases [75].

### Evolution in Organisation Use of Data Warehouses

- Organisations generally start off with relatively simple use of data warehousing. Over time, more sophisticated use of data warehousing evolves. The following general stages of use of the data warehouse can be distinguished:

- Off line Operational Database: Data warehouses in this initial stage are developed by simply copying the data of an operational system to another server where the processing load of reporting against the copied data does not impact the operational system's performance.
• Off line Data Warehouse: Data warehouses at this stage are updated from data in the operational systems on a regular basis and the data warehouse data is stored in a data structure designed to facilitate reporting.

• Real Time Data Warehouse: Data warehouses at this stage are updated every time an operational system performs a transaction (e.g., an order or a delivery or a booking).

• Integrated Data Warehouse: Data warehouses at this stage are updated every time an operational system performs a transaction. The data warehouses then generate transactions that are passed back into the operational systems.

• Frontline Data Warehouse: Data warehouses at this stage provide frontline applications and users with immediate access to a full source data-set that reflects the current state of the business, providing continuous availability, and the ability to analyse data in-place without moving data to a middle-tier analytics engine.
Chapter 4

Proposed Research Method and Approach

In Chapter 4 business intelligence recovery method and software reverse engineering techniques have been introduced. Business intelligence is an atomic comprehensibility unit in understanding software system. It answers questions like “What are the functionalities of a piece of code? What are the relationships among parts in it? What is the business behaviour/meaning of the code?” And so on. In another word, not only does business intelligence recovery take program code as a substance to examine and understand, but also interprets the business intelligence related information. Business intelligence generates program code and business data into comprehensibility of business software system. Comprehensibility is complementary in source code, however, it is hard to elicit and interpret. More specifically, it lacks a method to handle the atomic unit of comprehensibility. Business intelligence recovery bridges the software analysis [76] and business analysis, which is our proposed solution to solve the issue. In this chapter, we introduced our solution.

In this Chapter, Section 4.1 introduces information elicitation techniques. Section 4.2 introduces the research method and solution on business intelligence recovery. Linear program partition, concept recovery, and formal concept analysis are presented in Section 4.3. Section 4.4 evaluates our solution and makes a summary.

4.1 Information Elicitation from Program Code

Program understanding is, essentially, a set of activities of acquiring useful information from source code. In this section, various kinds of information that can be elicited from software code and a collection of fundamental techniques that can support automated information elicitation from source code will be introduced. In what follows in this section, Section 4.1.1 makes a comparison among some related terms that can be frequently found in the literature. In Section 4.1.2 several stages for business
intelligence recovery from source code are introduced. Various kinds of business intelligence that can be elicited from source code are listed in Section 4.1.3. Supporting techniques for automated business intelligence recovery are gathered in Section 4.1.4. Finally the problems are identified and new solutions are proposed in Section 4.1.5.

### 4.1.1 Related Terms

One of the most frequently-used terms in the field of information elicitation from source code is reverse engineering. Traditional software development moves from business-level abstractions down to the physical implementation of code, which is often called forward engineering. Reverse engineering, on the other hand, moves from physical implementation and source code up to high-level abstraction, ideally to requirement. Figure 4.1 illustrates such a comparison.

Information elicitation from source code is a rather wide term that can be related to the fields of reverse engineering, design recovery, program understanding, reverse specification [6] and restructuring. These terms were defined by Chikosky and Cross [45] and Byrne [37]:

- **Design Recovery** is related to reverse engineering in which business information, external information, and deduction information or hypothetic design models are added to the observations of the subject system to identify
meaningful higher level abstractions beyond those obtained directly by examining the system itself. Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), recovery experience, and general knowledge about business and application.

- **Program Understanding** is a related term to reverse engineering. Program understanding implies always that understanding begins with the source code while reverse engineering can start at a binary and executable form of the system or at high level descriptions of the design. Program understanding can be achieved in an ad hoc manner and no external representation has to arise, while reverse engineering is the systematic approach to develop an external representation of the subject system. Program understanding is comparable with design recovery because both of them start at source code level.

- **Reverse Specification** is a kind of reverse engineering where a specification is abstracted from source code or design description. Specification in this context means an abstract description of what the software does [89]. In forward engineering, the specification tells us what the software has to do. But this information is not included in the source code. Only in rare cases can it be recovered from comments in source code and from the people involved in the original forward engineering process.

- **Restructuring** is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behaviour (functionality and semantics). Having the above prerequisites, we can informally define information elicitation from source code as a sequence of information elicitation activities that have source code as their initial starting point. Here, the number of the activities in the sequence can be 1. Although theoretically, any sequence of information elicitation activities starting from source code can be justified by this definition, only a handful of such sequences can be found in practice, which will be introduced in the sections that follow.

- **Information Elicitation Process Models:** The IEEE Standard for Software Maintenance [3] suggests that the process of reverse engineering evolves through six steps: dissection of source code into formal units; semantic description of formal units and creation of functional units; description of links for each unit (input/output schematic of units); creation of a map of all units and
succession of consecutively connected units (linear circuits); declaration and semantic description of system applications; and creation of an anatomy of the system. The first three steps concern local analysis on a unit level (in the small), while the other three steps are for global analysis on a system level (in the large). Two important factors have not been included in this process model, namely, the business intelligence a reverse engineering system processes and the role a tool has in this process.

- **Business Intelligence Recovery:** Software forward engineering is the translation of a requirement into code through development steps. During the forward engineering process, it is difficult to judge whether coding or business model in coding behaviour is more responsible for the success or failure of a software development. This is why it is important to understand processes collaboration between both code and business organisation in the total forward engineering process. On contrary, it is the same processes collaboration that occurs between business intelligence recovery and software reverse engineering.

Business intelligence recovery process is achieved with reverse engineering techniques. It is partitioning both large source program and business intelligence into small manageable units (functional, linear, and connected), mapping the relationships and connections to interpret meaning of the units.

Business intelligence recovery techniques, by the moment, are divided into two groups: data mining and data analysis. But, the best result should come true in technique collaboration with software engineering, e.g. program partitioning, program slicing, concept recovery, and formal concept analysis. The derivative tools of business intelligence recovery are emerging because of the evolving of code languages, programming standards, application domain and so on. No single one of the tools can deal with all the questions or fulfil all needs.

### 4.1.2 Business Intelligence Recovery Process

The process of business intelligence recovery is compared with software reverse engineering. They both pass through several steps to have system abstractions. Generally, our research of business intelligence recovery is partitioned into 4 stages in Figure 4.2:
• Legacy code is the starting point of research. It is the input of software reverse engineering and business intelligence recovery. Business intelligence data is elicited from source code. It includes algorithms, and embedded concepts. On the other side, reverse engineering program code is interpreted into program structure.

• Business intelligence algorithm is abstracted into mathematics representations. It leverages the level of abstraction to interpret mathematic meaning in code organisation. On the other side, reverse engineering program module structure is further interpreted into component models.

• Formal concept based physics model is built in business intelligence recovery to enriches the comprehensibility between business model and code organisation. On the other side, software architecture is built in reverse engineering to understand component organisation.

• Business intelligence physics model is interpreted into business logic, which is always connected with a goal of business behaviour. On the other side, reverse engineering program architecture is interpreted into system specifications, which is fulfilled in software requirement.

Our research focuses on first three stages. Firstly, the source program and business intelligence are partitioned into manageable modules and units. Secondly, business intelligence is elicited from program code while performing program partition. Finally, business intelligence is modelled via formal concept analysis.

Figure 4.2: Proposed Research Steps of Business Intelligence Recovery
Chapter 4 Proposed Research Method and Approach

4.1.3 Proposed Techniques of Business Intelligence Recovery

This research is applying automatic techniques to finish business intelligence recovery in software reverse engineering process. In Figure 4.3, the results of business intelligence recovery by the chosen techniques are listed in the first column. The key point of this table is every business intelligence result is produced upon the result of the former one. Therefore, the techniques must be sufficient to have two business intelligence results, or a single result could be produced by two techniques separately. In the table, there are one NO spaces need us to fill YES instead. In another words, we are going to further develop concept recovery method to take advantage of the program partitioning result which is business intelligence oriented program modules.

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>Algorithms</th>
<th>Mathematics Model</th>
<th>Physics Model</th>
<th>Business Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Slicing</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Partition</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept Recovery</td>
<td>NO</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formal Concept Analysis</td>
<td></td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Survey of Techniques for Business Intelligence Recovery

In summary, our proposed techniques on business intelligence recovery from program code to program model, and from business intelligence data to business logic are program partitioning, concept recovery, and formal concept analysis.

4.2 Research Route of Business Intelligence Recovery

The research route of business intelligence is build on research of software reverse engineering, which covers research steps from program code to program structure, object and component, object model and component model, architecture, specification and requirement.
4.2.1 Business Intelligence Recovery and Reverse Engineering

Figure 4.4 contains three key factors: relation of business intelligence recovery and reverse engineering, research route, and research points. Business intelligence recovery and reverse engineering are put into a Plane Cartesian coordinate system, because there is a certain relationship between those two processes. With a certain point of view in mathematics, the two processes can be seen as vertical rectangular cross-ties. The research route is passing through all cross-points located within the Plane Cartesian coordinate system. And it is our research work to find each of them. In Figure 4.5, five zones are drawn to indicate each different abstraction level both in business intelligence recovery process and in software reverse engineering. These five zones limit the scope of activities of any cross-point our research might come across:

- Program Code v.s. Business Intelligence Data
- Program Structure v.s. Business Intelligence Algorithm
- Component Model v.s. Business Intelligence Mathematics Model
- Architecture v.s. Business Intelligence Physics Model
- Requirements v.s. Business Logic
Chapter 4 Proposed Research Method and Approach

Figure 4.4: Business Intelligence Recovery and Reverse Engineering

Figure 4.5: A Five-Tier Perspective on Business Intelligence Recovery
4.2.2 Proposed Research Methodology

In this subsection, three research methodologies in business intelligence recovery are further introduced. These three methodologies are covered by the past and current publications in Software Evolution and Reengineering Group (SERG) [78, 79, 80, 81, 82, 109, 136, 137]. The methodologies are summarised, see Figure 4.5:

**First Methodology**
\[(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (2,2) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,4) \rightarrow (4,4) \rightarrow (4,5) \rightarrow (5,5)\]

The first methodology focuses on reverse engineering, which is vertical upward. It passes the white points above black ones in Figure 4.4. Researchers with this method believe that at the end of each step of software reverse engineering, the abstractions of an artifact are research target for business intelligence recovery. Apparently, the method eases the effort in business intelligence recovery task. However, it makes the research route line to the Cartesian coordinate system point (0, 0) axe, anti-clockwise to move closer to the vertical axis. Obviously, result is that the second reverse engineering step will be much more difficult than the first one. The recovery results are code and some incomplete or incorrect program modules.

**Second Methodology**
\[(0,0) \rightarrow (1,0) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,3) \rightarrow (4,4) \rightarrow (5,4) \rightarrow (5,5)\]

The second methodology focuses on business intelligence recovery, which is horizontal right-hand side. It passes the white points below back ones in Figure 42.3. The second method is the anti-case of first one. It makes the research route line to the Cartesian coordinate system point (0, 0) axe, clock-wise to move closer to the horizontal axis. The result obviously is that researchers could hardly have an idea of what business data and program code mean. Without program partition, no one could understand business data and program code completely.

We present these two methodologies to makes the issue clear that business intelligence recovery and software reverse engineering are two processes being complement to each other. The complementary between them makes them dependent coexistence.
Chapter 4 Proposed Research Method and Approach

Our Method = (0,0) ⇒ (1,1) ⇒ (2,2) ⇒ (3,3) ⇒ (4,4) ⇒ (5,5)

Therefore, we put forward our methodology to address and solve the issue. The method is that we carry out reverse engineering and business intelligence recovery separately, but at the same time, at the same level. Between the two parallel processes, there are mutual guidelines, interdependence and interrelated set at every abstraction level. And our research is to localise every cross-point. Section 4.3 will present the approach and developed techniques to support our method.

In this thesis, we only present a current research of business intelligence recovery in the first zone, which covers from data to business intelligence algorithms, accordantly in reverse engineering, from program code to program structure. But we fully present the methods and techniques of business intelligence recovery in software reverse engineering research. The other part of the research is not present because of the pages limit.

Figure 4.6: Research Methods on Business Intelligence Recovery
4.3 Proposed Approach on Business Intelligence Recovery

Our proposed approach for business intelligence recovery is linear program partition plus business intelligence matrix decomposition. Linear program partition is a linear programming style based program partition method, which is developed to partition program and to elicit business intelligence data. And business intelligence matrix decomposition is a linear algebra based business intelligence matrix decomposition method, which is developed to sound proof elicitation result and take advantage of the elicitation result.

There are two steps in this section, linear programming style based program partition, and recovering business intelligence slices from program modules. But first of all we need to introduce a concept, business intelligence slices.

**Business Intelligence Slices**

Business intelligence is the high abstract level of a good understanding of software products. The major problem of traditional concept assignment methods is their heaviness and lack ability of scale up. In order to design a kind of intelligence representation for recovering large-scale business intelligence from source code, the nature of this task must be studied carefully.

Given a source program, the major clue for the existence of business intelligence is the names of variable styles and procedures and certain program constructs implying business relationships. However, the names embedded in source program often occur as abbreviations and interpreting certain program constructs as some business relationships may not always be appropriate, which makes the business intelligence embedded in the source program ambiguous. This fact suggests that a complete and comprehensive recovery of business intelligence from source code is rather difficult. Technically, the recovery of business intelligence from source code involves a match between business intelligence and source code; once a successful match between the two is found, we can then conclude that the business intelligence is recovered from the source code. On the other hand, if parts of the business intelligence base are not successfully matched during
a business intelligence recovery process, the decision on acknowledging the whole semantic network as an interpretation of the source program will be hard to make.

To minimise the effect that an unsuccessful match between business intelligence base and source code has on the process of business intelligence recovery from source program, we introduce a new concept called business intelligence slices into business intelligence representation. A business intelligence slice is defined as a set of strongly related business concepts linked by a set of relationships among these concepts. Multiple business intelligence slices could exist for a single set of business concepts depending on the number of different groups of relationships among these concepts. Business intelligence is therefore regarded as a collection of business intelligence slices which are linked with each other through the common concepts that these business intelligence slices share. The match between business intelligence base and source program is carried out between individual business intelligence slices and source program and the chance of a successful match between business intelligence base and source program is greatly enhanced, See Figure 4.7.

**Linear Program Partition**

Our method on linear program partition method in Figure 4.7 uses a special kind of heuristic knowledge – human programming styles which can partition a source program into business intelligence-oriented program modules at a linear time-scale.

![Figure 4.7: Proposed Techniques of Business Intelligence Recovery](image-url)
The process of recovering a business intelligence slice from a source program is, in its simplest form, a match between the concepts and relationships, and the source program. For a concept, the match involves searching for the name(s) of this concept among the variable style names and function names in the source program; while for a relationship, the match involves searching for particular program constructs which implies the relationship.

Let $P$ be a source program; $M_1, ..., M_n$ be a set of program modules that are the result of partitioning of $P$, each $M_i$, $1 \leq i \leq n$ contains independent business intelligence. Let $BIB$ be a business intelligence base; $BIB_1, ..., BIB_m$ be a set of partitions of $BIB$, each $BIB_j$, $1 \leq j \leq m$ contains independent sub-business intelligence from each other. Suppose we are going to recover $BIB_j$ whose sub-business intelligence from $P$. Two scenarios for this task can be:

**Scenario 1:** $M_1, ..., M_n$ is unknown. In this case the recovery of $BIB_j$, has to be conducted in the whole scope of $P$. For each concept in $BIB_j$, it will search among all the important names in $P$ to find a match; for each relationship in $IB_j$, it will check all the important program constructs in order to validate itself. In some cases, the validation of a single relationship may involve traversing multiple program constructs and resulting in a search space whole size is a non-linear function of the size of $P$. Since the size of $P$ can be huge, the computational effort of recovering $BIB_j$ from $P$ is exceptionally high.

**Scenario 2:** $M_1, ..., M_n$ is known. In this case, the exact sub-business intelligence that each $M_i$, $1 \leq i \leq n$ is associated with can also be made clear, either automatically or manually. Suppose only $M_i$ contains the sub-business intelligence that $BIB_j$ has. The concepts in $BIB_j$ it will only need to search among all the important names in $BIB_j$ for a match and for each relationship in $BIB_j$ it will only need to check the important program constructs within $BIB_j$, significant saving of computational effort can achieved since $BIB_j$ is only a portion of $P$. 
4.4 Summary

Reverse engineering practises system abstractions from legacy code. Accordantly, business intelligence recovery practices business intelligence recovery from existing code and data. It is believed that software reverse engineering at code level could hardly progress without business intelligence recovery of code organisation and business model in program understanding. This is because the complementary between the processes makes them dependently coexistent. Our proposed research method is mixing the techniques of program partition, information elicitation and representation techniques to have business intelligence automatically recovered from software system. The contribution is having these techniques overlapped and paralleled working together.
Chapter 5

Partitioning Business Intelligence into Business Intelligence Slices

As introduced in the previous chapter, the first step of designing an approach for automatically recovering business intelligence from source code is to construct a business intelligence base which can later be used, in conjunction with program partition, concept recovery and formal concept analysis, to establish the linkage between source code and business intelligence. In this chapter, the design principles of the business intelligence base deployed there will be discussed. Since business intelligence representation is a crucial issue, some frequently-used business intelligence representations will be introduced first. The proposed business intelligence representation in this thesis will then be described and the relationship between business intelligence representation and classical business intelligence representation will be made explicit.

The general requirement of a business intelligence representation technique will be given in Section 5.1. In Section 5.2 three classical business intelligence representation techniques is described. In Section 5.3 a new concept of business intelligence slices which is used as the foundation of this work is introduced.

5.1 General Requirement for Business Intelligence Representation

Every business intelligence recovery system can be thought of as a collection of business intelligence and usage of this business intelligence. There are generally three issues associated with business intelligence: (1) how to acquire business intelligence; (2) how to represent the business intelligence and (3) how to use the business intelligence.
A good system for the representation of business intelligence in a particular system domain should possess the following two properties:

- **Representational Adequacy** the ability to represent all the kinds of business intelligence needed in that system.

- **Acquisition Efficiency** the ability to acquire new business intelligence easily. The simplest case involves direct insertion, by a person, of new business intelligence into business intelligence base. Ideally, the program itself would be able to control business intelligence acquisition.

There is always a trade-off between adequacy and efficiency, i.e., the more adequate a representational mechanism is, the less efficient the acquisition mechanism will be, and vice versa. This has been demonstrated in Chapter 4 where the capacity of existing techniques for information elicitation from source code was discussed.

It is therefore not surprising that no single system that optimises all of the capabilities for all kinds of business intelligence has yet been found. As a result, multiple techniques for business intelligence exist. Many programs rely on more than one technique.

### 5.2 Business Intelligence Representation Techniques

In this section, two classical business intelligence representation methods are introduced. They are logic, and formal concept analysis.

#### 5.2.1 Logic

Logic is one particular way of representing facts. The logical formalism is appealing because it immediately suggests a powerful way of deriving new business intelligence from old mathematical deduction. In this formalism, it can be concluded that a new statement is true by proving that it follows from the statements that are already known. Thus the idea of a proof, as developed in mathematics as rigorous way to demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answer to questions and solutions to problems.

One of the early domains in which logic is explored was mathematical theorem proving, by which was meant proving statements in various area of mathematics. For example, the Logic Theorist [102], proved theorems from the first chapter of Whitehead and
Russell’s Principia Mathematica [131]. Another theorem Prover [63] proved theorems in geometry. The usefulness of some mathematical techniques extends well beyond the traditional scope of mathematics [96] and it turns out that mathematics is no different from any other complex intellectual endeavour in requiring both reliable deductive mechanisms and a mass of business intelligence.

The simplest form of logic is propositional logic in which no variable and quantification occurs. For example, to represent “BlackBerry Bold is a smart phone” in the form of propositional logic, we got BlackBerryBold(SmartPhone).

By incorporating variables and quantifications, we move to first-order predicted logic (or just procure logic). Unlike propositional logic, first-order logic does not process a decision procedure, even an exponential one. There do exist procedures that will find a proof of proposed theorem of indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. In other words, although first-order predicate logic is not decidable, it is semi-decidable. An example for using first-order logic to reason is shown below:

The logics to represent are:

1. All BlackBerry are smart phones.
2. BlackBerry Bold is a BlackBerry phone.
3. BlackBerry Bold is a smart phone.

The representations are expected in this way:

1. BlackBerry (V) → Smartphone (V)
2. BlackBerryBold (BlackBerry)
3. BlackBerry (Smartphone)

Note: 3 is the logical implication of 1 and 2.

There are two different kinds of way to reason with statements in predicate logic. One is Resolution, the other is Natural Deduction.
In this thesis, propositional logic, combined with other business intelligence representation methods, is used to present business intelligence base and the facts in the business intelligence recovery process.

### 5.2.2 Formal Concept Lattice

The use of Formal Concept Analysis [59] in software engineering is not new. Snelting et al. [120] use FCA to reengineer C++ class hierarchies, while Ar´evalo et al. [10, 17] analyse object oriented framework reuse using FCA. Closer to the work of this thesis are the techniques by Tonella et al. [125]. The former use FCA to detect instances of design patterns in source code. Since they specifically use the FCA algorithm for detecting such instances, they are not able to detect other kinds of design symptoms, as the approach in this thesis does. The latter use FCA to reveal the structure of single classes only. They partition the methods of a class, according to the fields these methods use, and then use the concept lattice to visualise and understand the structure of that class. Tilley et al. [123, 124] provide an overview of the use of FCA for several other software engineering purposes.

There are a large number of tools to verify the quality of the source code of an application. The spectrum ranges from very simple tools that detect basic coding errors, over specialised clone detection tools [23, 31, 36, 53], to tools that detect high-level bad smells [55, 83, 126] and propose appropriate refactorings.

Other tools exist that are capable of detecting high-level structures in source code, such as coding conventions and design patterns [7, 31, 99, 105]. The main difference between these tools and ours, is that the approach in this thesis requires no a priori knowledge. Most of these existing tools, however, rely on the fact that design pattern implementations follow particular naming conventions and guidelines.

The approach in this thesis is not targeted to detecting a specific kind of conventions, but is able to detect a variety of symptoms that reveal bad design/duplication [21, 85], good design, and opportunities for refactoring. This is particularly useful during initial understanding, whereas in a later phase, when the code is better understood, a more directed tool is preferable.

Research in the domain of aspect mining [29, 115] is also related. Tonella et al. [125] and Breu et al. [24] find aspect instances in existing applications by means of dynamic execution traces. The former work even uses formal concept analysis to that extent. The
approach in this thesis is to complement these approaches, since the combination of static analysis techniques with formal concept analysis is the aim. Marin et al. [97] use the fan-in metric in order to mine for aspects. The further detail discussion of FCA is in Chapter 8.

### 5.3 Business Intelligence Slices

In Chapter 4 it is mentioned that different layers of information can be elicited from source code and argued that the information at business intelligence layer should be paid more attention. Recently, software evolution become a major theme of software engineering and it is well established that software evolution often results from the jointing, migrating and transforming services of business intelligence. Locating business intelligence in source code thus becomes attractive.

Business intelligence refers to a set of business data and concepts that describe the contextual information of a particular business task/behaviour. It is the high abstract level information of business in software system, can give software engineers a good understanding of both business software system and their evolution needs.

Traditional business intelligence analysis methods often use semantics nets as a business intelligence representation [71]. The basic structure of semantics nets were introduced in by Biggerstaff [28], concepts and relationship are used to denote node and arc in a traditional semantic net. The technical details of Biggerstaff’s work were described in Section 5.4. As concluded in Section 5.5 the major problem of traditional concept assignment methods is their heaviness and lack of ability of scale up. In order to design a kind of business intelligence representation for recovering large-scale business intelligence from source code, the nature of this task must be studied carefully.

Given a source program, the major clue for the existence of business intelligence is the names of variable types and procedures and certain program constructs implying business oriented relationships. However, the names embedded in source program often occur as abbreviations and interpreting certain program constructs as some business intelligence oriented relationships may not always be appropriate, which makes business intelligence embedded in the source program ambiguous. This fact suggests that a complete and comprehensive recovery of business intelligence from source code is rather difficult. Technically the recovery of business intelligence from source code involves a match between business intelligence and source code; once a successful
match between the two is found, we can then conclude that the business intelligence is recovered from the source code. On the other hand, if parts of the business intelligence base (in the representation of formal concept lattice) are not successfully matched during a business intelligence recovery process, the decision on acknowledging the whole business intelligence concept lattice as an interpretation of the source program will be hard to make. Existing formal concept analysis methods have not tackled this problem well. They treat the business intelligence base as a whole, i.e., every node and arc in the formal concept lattice will be involved in the same training process and the activation signals are propagated in a single-way across the whole formal concept lattice given the evidence available. Since both the training and recognition of large-scale business intelligence concept lattice is time-consuming, existing business intelligence concept assignment methods are only application to small-scale business intelligence assignment tasks.

To minimise the effort that an unsuccessful match between business intelligence base and source code has on the process of business intelligence recovery from source program, we introduce a new concept called business slice into business intelligence representation. A business intelligence slice is defined as a set of strongly related business intelligence concepts linked by a set of relationship among these concepts. Multiple business intelligence slices could exist for a single set of business intelligence concepts depending on the number of different groups of relationships among these concepts. Business intelligence is therefore regarded as a collection of business intelligence slices which are linked with each other through the common concepts that these business intelligence slices which are linked with each other through the common business intelligence concepts that these business intelligence slices share. The match between business intelligence base and source program is carried out between individual business intelligence knowledge slices and source program and the change of a successful match between business intelligence base and source program is greatly enhanced.

Since the unnecessary relationships, e.g. the weaken relationship in Figure 5.1, between business intelligence concepts are removed as the business intelligence base is partitioned into business intelligence slices, the size of overall business intelligence base is significantly reduced. This means that we can get a cost-effective business intelligence base and benefits will also be gained when recovering business intelligence from source code using this business intelligence base.
The idea of partitioning business intelligence base into business intelligence slices and using the business intelligence slices to recovery business intelligence from source code is in accordance with Finite Element method [35] commonly used in mechanical engineering fields. In Finite Element method, a finite number of basic structures, such as triangles and circles, are used to approximate more complex shapes in the real world. Since each basic structure is easier to compute, the computation of arbitrary and complex shapes is therefore manageable and overall computation effort is significantly reduced.

Although both aim to handle the complexity of real-world problems by breaking them into basic structures, the proposed concept of business intelligence slice is different from Resolutional Theory [111] in that the latter is (1) the transformation of a problem rather than the approximation of the problem; (2) the breaking of a problem into substance at a layer that contains only positive atoms and negative atoms, where business intelligence is lost.

In order to accommodate this idea, we change the formal concept lattice into two-layer lattice, namely, concrete concept lattice and abstract concept lattice. Concrete concept lattice contain detailed information on concepts and relationships among them. Each concrete concept lattice is associated with a single business intelligence slice. Abstract concept lattice contain only business intelligence concepts and links to corresponding concrete concept lattice. The concrete concept lattice is used as templates for the recovery of individual business intelligence slices from source code (see Chapter 6) whereas the abstract concept lattice can facilitate business intelligence slice analysis (see Chapter 7).

Figure 5.2 illustrates the simplification of monolithic business intelligence into loosely-coupled business intelligence slices. The upper part of this figure shows a trunk of monolithic business intelligence that contains concepts (denoted as circle nodes) and relationship between these concepts (denoted as arcs). The relationships are further labelled as strong relationships SRi and weak relationships WRi. The labelling of relationships as SRi or WRi pertains to the nature of the particular business intelligence addressed. Although a subjective matter, the labelling of relationships can be decided on once the business intelligence is fixed.

The lower part of Figure 3.1 shows the partitioned business intelligence slices. Each business intelligence slice contains only SRi, and WRi are removed. Given a trunk of monolithic business intelligence, there are different alternatives to partition it into a
group of business intelligence slices. The factors that affect the partition choices include the size of each business intelligence slice, the coherence of each business intelligence slice and the coupling between different business intelligence slices. Since all these factors are largely subjective matters and the partitioning of business intelligence base into business intelligence slices can be down off-line, it will not be discussed in this thesis. But it should be noted that once the business intelligence base is labelled properly and the partitioning criteria is well defined, it is possible to automate the process of partitioning business intelligence base into business intelligence slices. In this thesis, it is assumed that a well partitioned business intelligence base has been obtained.

At the lower part of Figure 5.2, circle Ci connected by solid lines denote a concrete business intelligence concept lattice. Un-circled Ci, together with dotted lines denotes an abstract business intelligence concept lattice. To give the reader a picture of what an abstract business intelligence concept lattice and concrete business intelligence concept lattice looks like, Figure 5.2 shows an abstract business intelligence concept lattice in a telecommunication system, whereas, Figure 5.2 is an example of concrete business intelligence concept analysis. In Figure 5.3, the concepts and relationships bound by an arc constitute a single knowledge slice. If there is no arc on an edge, the edge and the two concepts connected by the edge will constitute a business intelligence slice themselves. Since the existence rather than the meaning of the relationship is of prior concern in the abstract business intelligence concept, all the edges in the abstract business intelligence lattice are un-directional. This is contrasted with concrete business intelligence networks where the detailed meaning of a relationship needs to be represented. In Figure 5.2, the edges are therefore directional.
Chapter 5 Partitioning Business Intelligence into Business Intelligence Slices

Figure 5.1: Simplification of a Complex Business Intelligence Base

Figure 5.2: Simplification of a Business Intelligence Base of an Online Mobile Retailing System
5.4 Business Intelligence Object and Action

In a general business intelligence concept lattice, the content of nodes and arcs can be anything imaginable. This brings complexity in handing this content. In the context of recovering business intelligence from source code, the nature of business intelligence and source code, again, needs to be carefully studied.

Software is normally developed to fulfil certain business functionality/operation. These operational functionalities are exemplified by a set of objects and a set of actions operating on these objects, and there is no need to include other descriptive elements in the business intelligence base.

In order to describe business intelligence succinctly, Verb-Noun as the basic elements for describing the operational business intelligence is introduced. In particular, the concepts in the abstract/concrete business intelligence concept slice are classified into two categories, namely, objects and actions. An object represents a class, an instance, a feature, etc., whereas an action represents an operation or an event which occur between several objects. The relationships in concrete concept lattice can, accordingly, be classified into the relationships between objects and objects, objects and actions, and actions and actions. Table 5.1 describes relationship examples in each category.

5.5 Summary

In this chapter, the issue of business intelligence representation for the proposed approach is addressed, in particular, it is proposed to use business intelligence slices as
the building blocks of business intelligence base. It is advocated that the principle of “simplicity” for business intelligence representation and proposed objects and actions are used to represent entities and the relationships between object and object, action and action, and object and action. The business intelligence partitioning algorithm is not given, since business intelligence partitioning can be done off-line and is not the top research priority. In the following chapters, it is assumed that a well partitioned business intelligent base has been obtained.
Chapter 6

Linear Programming Style based Program Partitioning

As mentioned earlier, traditional program partitioning methods are non-linear; in most case, the computational effort needed for partitioning a source program will increase exponentially as the size of the source program grows. This causes a computational barrier for program analysis, since the partitioning of source program must be conducted on-line and the efficiency of any program partition source program into program modules. This method uses a special kind of knowledge of programming styles which can partition a source program into business model oriented program modules at linear time-scale.

The following sections will be organised as follows. The necessity of program partition for the task of business intelligence recovery from source code is firstly discussed in Section 6.1. The motivation of the programming style based program partition method is then stated in Section 6.2. In section 6.3 a collection of program styles that are used to partition source program into program modules is presented and the implementation of each program style is made explicit. Finally several algorithms for program partition are described in Section 6.4.

6.1 Implication of Program Partitioning for Business Intelligence Recovery

In Chapter 4 we mentioned that a business intelligence slice is composed of a set of business intelligence units and a set of relationships between these units. The process of recovering a business intelligence slice from a source program is, in its simplest form, a match between the units and relationships, and the source program. For a unit, the
match involves searching name(s) of this unit among the variable style names and function names in the source program; while for a relationship, the match involves searching for particular program constructs which implies the relationship.

### 6.1.1 Independent BI

Let \( P \) be a source program; \( M_1, \ldots, M_n \) be a set of program modules that are the result of partitioning of \( P \), each \( M_i, 1 \leq i \leq n \) contains independent business intelligence. Let \( IB \) be a business intelligence base; \( BIB_1, \ldots, BIB_m \) be a set of partitions of \( IB \), each \( BIB_j, 1 \leq j \leq m \) contains independent sub-business intelligence from each other. Suppose we are going to recover \( BIB_j \) whose sub-business intelligence is known from \( P \), two scenarios for this task can be:

**Scenario 1:** \( M_1, \ldots, M_n \) is unknown. In this case, the recovery of \( BIB_j \) has to be conducted in the whole scope of \( P \). For each concept in \( BIB_j \), it will search among all the important names in \( P \) to find a match; for each relationship in \( BIB_j \), it will check all the important program constructs in order to validate itself. In some cases, the validation of a single relationship may involve traversing multiple program constructs and resulting in a search space whose size is a non-linear function of the size of \( P \). Since the size of \( P \) can be huge, the computational effort of recovering \( IB_j \) from \( P \) is exceptionally high.

**Scenario 2:** \( M_1, \ldots, M_n \) is known. In this case, the exact sub-business intelligence that each \( M_i, 1 \leq i \leq n \) is associated with can also be made clear, either automatically or manually. Suppose only \( M_i \) contains the sub-business intelligence slice that \( BIB_j \) has. The recovery of \( BIB_j \) from \( P \) can therefore be conducted in the scope of \( M_i \). In particular, for each concept in \( BIB_j \) it will only need to search among all the important names in \( M_i \) for a match and for each relationship in \( BIB_j \) it will only need to check the important program construct within \( M_i \). Even if computational effort of recovery \( BIB_j \) from \( M_i \) is still a non-function of the size of \( M_i \), significant save of computational effort can be achieved since \( M_i \) is only a portion of \( P \).

### 6.1.2. Dependent BI

Let \( P \) be a source program; \( M_1, \ldots, M_n \) be a set of program modules that are the result of partitioning of \( P \), each \( M_i, 1 \leq i \leq n \) contains dependent business
intelligence. Let IB be a business intelligence base; \( BIB_1, \ldots, BIB_m \) be a set of partitions of \( BIB \), each \( BIB_j \), \( 1 \leq j \leq m \) contains dependent sub-BIB from each other. Suppose recovery \( BIB_j \) whose sub-business intelligence from \( P \) is going to be recovered. Two scenarios for this task can be lower triangular and up triangular.

Suppose the matrix \( A \) is a product of two matrices, \( L \cdot U = A \), where \( L \) is lower triangular (has elements only on the diagonal and below) \( U \) is upper triangular (has elements only on the diagonal and above). For a case of a \( 4 \times 4 \) matrix \( A \), for example, \( A \) would look like this:

\[
\begin{bmatrix}
\alpha_{00} & 0 & 0 & 0 \\
\alpha_{10} & \alpha_{11} & 0 & 0 \\
\alpha_{20} & \alpha_{21} & \alpha_{22} & 0 \\
\alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33}
\end{bmatrix}
\begin{bmatrix}
\beta_{00} & \beta_{01} & \beta_{02} & \beta_{03} \\
0 & \beta_{11} & \beta_{12} & \beta_{13} \\
0 & 0 & \beta_{22} & \beta_{23} \\
0 & 0 & 0 & \beta_{33}
\end{bmatrix}
= \begin{bmatrix}
\alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\
\alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\
\alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\
\alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33}
\end{bmatrix}
\]

We can use a decomposition to solve the linear set \( A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b \) by first solving for the vector \( y \) such that \( L \cdot y = b \), and then solving \( U \cdot x = y \).

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equation is quite trivial. Thus, \( L \cdot y = b \) can be solved by forward substitution as follows:

\[
y_0 = \frac{b_0}{a_{00}}
\]

\[
y_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=0}^{i-1} a_{ij} y_j \right) \quad i = 1, 2, \ldots, N - 1
\]

while \( U \cdot x = y \) can then be solved by back substitution exactly as:

\[
x_{N-1} = \frac{y_{N-1}}{\beta_{N-1,N-1}}
\]

\[
x_i = \frac{1}{\beta_{ii}} \left( y_i - \sum_{j=i+1}^{N-1} \beta_{ij} x_j \right) \quad i = N - 2, \ldots, 0
\]

Dependent business intelligence modules partitioning is the one of the keys in business intelligence recovery. Here, in this section, we only present a simple idea of matrix decomposition method, which we are using to breaking up one dependent linear set into two successive independent ones.

In short, properly partitioning source program into business intelligence oriented modules can significantly reduce the computational effort needed in the process of business intelligence recovery from source code. But if the process of partitioning source program into program modules itself involves a non-linear computation effort,
the benefit of partitioning source program into program modules will vanish. The reason is that both the partitioning of source program and the recovery of business intelligence from the source program must be conducted on-line, and the non-linear computational effort incurred by the former overshadows that of the latter.

Existing methods for program partitioning are non-linear. The common idea of these methods [34, 38] is to cut program modules out of a source program based on their coupling and coherence. These methods generally involve three steps:

1. Find all the possible program modules;
2. Compute the coupling and coherence of each proposed program modules;
3. Choose the modules that have good value of coupling and coherence.

The calculation of coupling and coherence does not involve much computational effort once a program module has been decided and choosing good modules based on their value is just a simple sorting program. The most computation-consuming step is, however, the first step where candidate program modules are proposed. Given a source program $P$, proposing possible program modules is actually a task of finding all the combination of basic program element (such as procedures) within $P$, i.e., the number of possible modules is an exponential function of the size of $P$.

In order to reduce the computational order of program partitioning, new heuristic rules need to be sought to build efficient program partitioning algorithms.

### 6.2 Linear Program Partitioning

In order to design an efficient program partitioning method, the process of software development needs to be revisited.

It is well known that a large software program is generally co-written by a group of programmers with each programmer being responsible for part of the whole program. It is because business functional roles and the software functional parts are coexisted connected, but taken by different programmer. Each of these parts of the program is usually a self-contained program module with relative independent functionality. The task in this thesis is to identify these different program modules, thus make business intelligence recovered.
Since each of these program modules is associated with a particular programmer, the task of identifying these different program modules is equal to the task of identifying the different programmers who, together, wrote the whole source program. The question is: what is the signature that each programmer has in the source program? The three general requirements for such a kind of signature are:

**Distinctive.** Since the signature of each programmer is used to distinguish program from one another, each signature must be different.

**Stable.** The program module a programmer is responsible for can be spread across the different regions of the source program. A successful program partitioning method requires that these separated program portions written by a programmer can be put together. It is therefore needed that the signature of a programmer is stable across different regions of a source program.

**Easy to compute.** Computing the signature of programmers is the basic component of program partitioning method; if such computation is too costly, the benefit of program partitioning will be discounted.

In search of signatures of programmers we came across human program styles. Empirical studies [117] suggest that each program, having different training background and temperament, tends to use a particular code-writing style consistently. As a result, if different program style in a program can be identified when this information can be used as a signature to partition the program into smaller self-contained sub-modules. In this case, the first requirement and the second requirement mentioned above are satisfied.

In order to satisfy the third requirement, we identified three groups of features in source code that can be used to distinguish different programming styles. These are style of comments, style of names and style of indentation. The implication of these features for the signatures of programmer will be discussed in Section 6.3.

Let \( PS_1, \ldots, PS_m \) be \( m \) different groups of programming styles; let \( ps_{i,j}, 1 \leq i \leq m, 1 \leq j \leq nps_i \) be the different programming styles within group \( i \), where \( nps_i \) is the total number of programming styles in group \( i \). If only one programming style \( ps_{i,j} \) in one group \( PS_i \) is allowed to use to distinguish programmers, we will have totally \( nps_1 \times nps_2 \times \ldots \times nps_m \) different signatures to distinguish programmers. In other words, the source program can be, at the most, partitioned into \( nps_1 \times nps_2 \times \ldots \times nps_m \) different program modules.
Theoretically if we can list the entire possible group, we will virtually be able to distinguish all the programmers who contributed to a source program. In practice, some commonly-used groups of programming styles will be sufficient to partition a source program into reasonable small program modules.

However, one more thing to keep in mind is that when modifying an existing source file, the modifications should be coded in the same style as the file being modified. A consistent style is important, even if it is not the one usually you use.

6.3 Programming Styles

The perfect program partition result comes from well defined programming style. A good style guide could enhance the understanding quality of the code we face to. It is, however, the end itself that is important. Deviations from the standard style are acceptable if they enhance understanding and code maintenance. Major deviations require an explanatory comment at each point of departure so that later maintainers will know that you did not made a mistake, but purposefully are doing a local variation for a good cause. We list the common programming style to assist program understanding:

Files code should compile without error or warnings:

- **File Naming Conventions** Try to pick filenames that are meaningful and understandable. File names are not limited to 14 characters.

- **File Organisation** Although there is no maximum length requirement for source files, files with more than about 1000 lines are cumbersome to deal with. Lines longer than 80 columns should be avoided.

- **Header File Content** Header file should be functionally organised, with declarations of separate subsystems placed in separate header files. For class definitions, header files should be treated as interface definition files.

**Identifier Naming Convention** Identifier naming conventions make programs more understandable by making them easier to read. They also give information about the purpose of the identifier. Each subsystem should use the same conventions consistently. For example if the variable offset in blocks from the beginning of the file.

- **Identifier Style** Identifiers are upper caps, mixed case, or lower case. If an identifier is upper caps, word separation in multi-word identifiers is done with an underscore (for example, RUN_QUICK). If an identifier is mixed case, it
starts with a capital, and word separation is done with caps (for example, RunQuick). If an identifier is lower case, words are separated by underscore (for example, run_quick). Pre-processor identifiers and template parameters are upper case. The mixed case identifiers are global variables, function names, styles (including class name), class data members, enum members. Local variables and class member functions are lower case.

- **Namespace Clashes** There are two strategies: (1) minimise the number of clashable names, or (2) choose clashable names that minimise the probability of a clash. Strategy (1) is preferable, but clashable names cannot be totally eliminated.

**Using White Space** Blank lines and blank spaces improve understanding by offsetting sections of code that are logically related.

- **Long Lines** Occasionally an expression will not fit in the variable space in a line; for example, a procedure calls with many arguments, or a logical expression with many conditions. Such occurrences are especially likely when blocks are nested deeply or long identifiers are used. If a long line needs to be broken up, you need to take care that the continuation is clearly shown. For example, the expression could be broken after the last comma of a function call (never in the middle of a parameter express), or after the last operator that fits on the line. If they are needed, subsequent continuation lines could be broken in the same manner, and aligned with each other.

- **Comments** Comments should be used to give an overview of the code and provide additional information that is not readily understandable from the code itself. Comments should only contain information that is germane to reading and understanding the program. In general, avoid including in comments information that is likely to become out of date. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment in a source file. Discussion of nontrivial design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for such redundant information to get out of date.

**Constant Style** Numerical constant must be coded so that they can be changed in exactly one place. The usual method to define constant is to use cons or enum. The
enum data style is the preferred way to handle situations where a variable takes on only a discrete set of values because of added style checking done by compiler.

**Classes** It is very important to make sure that the class acts like a black box. The interface exported to clients and subclass should reflect precisely what they need to know and nothing more.

**Functions** Function declarations should be lined up in accordance with indentation of variables. Function parameters should be listed as many per line as reasonable. However, if a function takes only a few parameters, the declaration can be strung onto one line. If the function takes no parameters, both the opening and closing parenthesis must be on the same line. Function parameter names must be included in the function declaration, not just the parameter styles. This applies as well for usages where a function prototype is being used as a style. The only exception is for operators and single argument constructors where the meaning of the parameter is clear from that context.

**Statements** Each line must contain at most one statement. In particular, do not use the comma operator to group multiple statements on one line, or avoid using braces.

- **Compound Statements** Compound Statements are statements that contain list of statements enclosed in braces. The enclosed list must be indented one more level than the compound statement itself. The opening left brace must be at the end of the line beginning the compound statement and the closing right brace must be alone on a line, positioned under the beginning of the compound statement.

- **if/else Statements** An else clause is joined to any preceding close curly braces that is part of its if.

- **for Statements** The control structure of a for statement do not fit on one line, they each should be placed on separate lines or broken out of the loop.

- **Empty Loops** Loops that have no body must use the continue keyword to make it obvious that this was intentional.

- **goto Statements** While not completely avoidable, use of goto is discouraged. In many cases, breaking a procedure into smaller piece, or using a different language construct will enable elimination of a goto. The main place where a goto can be usefully employed is to break out of several nested levels of switch, for, or while nesting when an error is detected.
In this section, three different groups of programming styles are chosen for study and used to describe signatures of programmers, namely, comments, naming convention and program indentation. Some frequently-occurring patterns for each group of programming style are collected and their psychological implementations are discussed.

Let us begin with general discussion of human programmers and their tendency. Let $HP_1, \ldots, HP_n$ be $n$ human programmers; $A_1, \ldots, A_m$, be $m$ common attributes of human programmers, each $A_i$, $1 \leq i \leq m$ can be a distinctive dimension in tendency of programming habit. Each of these psychological attributes can have several values describing different kinds of behaviour or tendency within each psychological dimension, e.g., rigid, independent, and economical.

It is the observation that the programming styles in a source program have a high correlation with the values in the psychological dimensions and therefore the programming styles can be used to indicate a particular value in a dimension. This issue will be further discussed in Section 6.3.1, Section 6.3.2 and Section 6.3.3.

Let $n_{a_i}$ be the number of values in $A_i$, the total number of distinctive patterns $TNP$ can therefore be $n_{a_1} \times n_{a_2} \times \cdots \times n_{a_m}$. In the process of allocating $HP_1, \ldots, HP_n$ to $TNP$ different patterns, if $TNP < n$ there must be the case where more than one human programmer share the same patterns.

It should be noted that the task of program partitioning requires us to identify different programmers rather than different patterns. In the case of more than one program falling into the same pattern, they can still be distinguished by their programming styles since more than one kind of program style can be associated with one particular value in a dimension. Nevertheless, the implications of different program styles can justify the programming style based program partitioning method and help us understand what is happening behind the scene.

### 6.3.1 Programming Style of Comment

Comments are frequently used to make annotation on the functionality of a procedure or meaning of a variable. At the stage of pre-compilation of source program, comments are often removed from the program and therefore the content of comments does not affect the final executable code. Because of this feature, programmers usually write any information into comments in the style they prefer. In fact program comments are the areas where the richest programming styles can be found.
In order to add more dimensions to program comments, we further classify program comments into single-line comment and multiple-line comments. Single-line comment refers to the situation where there is only one line of comment for a piece of program comment; multiple-line comment, accordingly, refers to the situation where there is more than one line of content for a piece of program comment. The programming styles for a single-line comment and multiple-line comments are described in Script 6.1 and Script 6.2 respectively.

<table>
<thead>
<tr>
<th>Comments</th>
<th>Style 1:</th>
<th>/* X */</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Style 2:</td>
<td>/* X</td>
</tr>
<tr>
<td></td>
<td>Style 3:</td>
<td>/*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*/</td>
</tr>
<tr>
<td></td>
<td>Style 4:</td>
<td>/*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Naming Conventions</th>
<th>Style 1:</th>
<th>Connection-Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Style 2:</td>
<td>Connection_Mode</td>
</tr>
<tr>
<td></td>
<td>Style 3:</td>
<td>ConnectionMode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Indentation</th>
<th>Style 1:</th>
<th>bX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Style 2:</td>
<td>bbX</td>
</tr>
<tr>
<td></td>
<td>Style n:</td>
<td>b...bX</td>
</tr>
</tbody>
</table>

Table 6.1: Programming Styles

In each Script, the column of Style shows the general layout of a particular kind of comment. The column of Pattern generalises the layout of the comment in the same row into pattern represented in the form of a regular expression. The Code is a symbol denoting the programming style in this row.

To make notation more clearly, we have:

1. ‘/n’ denotes the control character ‘return’;
2. X stands for an arbitrary sequence of characters without ‘/n’, ‘*’;
3. \{X\} denotes that X can occur one or more times.

Some explanation for the programming styles in Script 6.1 and Script 6.2 are:

**SSC1** When displaying a single-line comment, SSC2 is more succinct than SSC1. Unfortunately in early versions of the C language, SSC2 is not supported; SSC2 is supported by C# which was invented later than C. A programmer who keeps using SSC1 can indicate himself/herself as an “old” C programmer.

**SSC2** A programmer who always uses SSC2 either suggests that he/she has the latest training on C# or is an old C programmer but can catch up with new programming concepts quickly.

**SMC1** This kind of programming style reflects a rigid personality of a programmer, i.e., he/she always makes things tidy.

The same programming style can also have a different explanation depending on the experience of the person who is trying to interpret it. Since the interpretation of programming styles at dimensions is not the major topic on this thesis, we will not give a further discussion.

In order to identify programming styles from source program, each piece of program comment is located and checked against different patterns; once a pattern is matched with the current piece of program comment, the associated code for this programming style is recorded.

### 6.3.2 Programming Style of Name Convention

Naming Convention refers to the style in which programmers write the name of a variable or a function in the source program. Similar to program comments, the compiler of a programming language gives programmers a certain degree of freedom to write names in the way they prefer. Normally the bottom-line for a name is that it must be a sequence of non-broken characters, e.g., without blank spaces inside the string.

In order to add more dimensions to naming conventions, we further break naming conventions into naming convention for variables, and naming convention for functions. In a source program, the function names are usually associated with operational concepts in an application domain; while the variable names entity concepts. Programmers therefore prefer to distinguish these two different kinds of names by
writing them in different styles. We present naming convention for variable names and function names in Script 6.3 and Script 6.4 respectively.

<table>
<thead>
<tr>
<th>Style for Naming Convention for variable names</th>
<th>Patterns</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style 1: Connection-Mode</td>
<td>W{-W}</td>
<td>SVN1</td>
</tr>
<tr>
<td>Style 2: Connection-Mode</td>
<td>W{-W}</td>
<td>SVN2</td>
</tr>
<tr>
<td>Style 3: ConnectionMode</td>
<td>W{W}</td>
<td>SVN3</td>
</tr>
</tbody>
</table>

**Script 6.3**

<table>
<thead>
<tr>
<th>Style for Naming Convention for function names</th>
<th>Patterns</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style 1: Connection-Mode</td>
<td>W{-W}</td>
<td>SFN1</td>
</tr>
<tr>
<td>Style 2: Connection-Mode</td>
<td>W{-W}</td>
<td>SFN2</td>
</tr>
<tr>
<td>Style 3: ConnectionMode</td>
<td>W{W}</td>
<td>SFN3</td>
</tr>
</tbody>
</table>

**Script 6.4**

Since the total set of programming styles for each of these two kinds of naming convention are similar, we discuss them mainly based on Script 6.3.

In Script 6.3 we, again, have three columns. The first column denotes examples of different naming conventions; the second column lists the generalised pattern for the naming convention in the same row; the third column is a list of code denoting these programming styles.

A variable name or a function names is usually composed of a single word or multiple words. We call an individual word an atomic name. In Script 6.3, W is used to represent an atomic name. \{W\} denotes that the occurrence of W can be once or more than once.

To write multiple words in a single non-broken sequence of characters, the connecting
manner among these words can vary. These connecting manners are collected as programming styles; some explanations are:

**SVN1 or SFN1** This kind of style suggests that the programmer who writes the code was a strong nature of cooperation.

**SVN2 or SFN2** A programmer who writes source code in this style has less cooperative nature than those who prefer SVN1 or SFN1.

**SVN3 or SFN3** A programmer who prefers this kind of Style has lowest cooperative nature. Also he/she must have good eyesight.

The pattern in Script 6.3 and Script 6.4 are used to match with variable names or function names in the source code. Once a match is found the associated code is recorded.

### 6.3.3 Programming Style of Program Indentation

Program indentation is used to make the logical structure of a source program visually clear. For example, the structure of the program show in Example 6.1 will be more readable by programmers than that in Example 6.2, although this does not make any difference to a compiler.

```cpp
int main()
{
    const double RESIDENT_RATE = 0.070;
    const double NON_RESIDENT_RATE = 0.045;

    int employee_count;
    char employee_code,
    state_code,
    response;
    double pay_rate,
    hours,
    regular_pay,
    overtime_pay,
    gross_pay,
    tax,
    net_pay,
    total_gross_pay,
    total_tax,
    total_net_pay;

    cout << setprecision(2)
```
Example 6.1

```cpp
int main()
{
    const double RESIDENT_RATE = 0.070; const double NON_RESIDENT_RATE = 0.045;
    int employee_count;
    char employee_code, state_code, response;
    double pay_rate, hours, regular_pay, overtime_pay, gross_pay, tax, net_pay,
    total_gross_pay, total_tax, total_net_pay;
    cout << setprecision(2)
        << setiosflags(ios::fixed)
        << setiosflags(ios::showpoint);
    // Display banner
    cout << "This program calculates the net pay for each employee."
        << endl << endl;
    // Initialize accumulators
    employee_count = 0;
    total_gross_pay = total_tax = total_net_pay = 0.00;
    // Main processing loop
    do
    {
        ...
    }
```

Example 6.2
6.4 Program Partitioning Steps

Having stated the motivation and rationale of programming style based program partitioning method, we will give several algorithms that work together to partition a source program into domain-oriented program modules. The thesis will then show that this proposed program partitioning method operates at linear time-scale.

A program will go through a partitioning process which has four stages, namely, programming style, sampling, programming style filtering, program cutting and program re-healing.

6.4.1 Program Style Sampling

At the programming style sampling stage, the source program is traversed and different kinds of programming styles are recorded when particular program single-line program comments, multiple-line program comments, variable name definitions, function name definitions, un-nested program statements and nested program statements. The candidate programming style pattern for a particular dimension are matched with the program construct encountered and the matching result is recovered.

An algorithm for creating sampling function for programming styles is given in Script 6.8. Some abbreviations are Programming Styles (PS), Current Program Line (CPL), Sampling Function (SF), and Sampling Interval (SI).

Algorithm for Program Style Sampling:

PS ← null
CPL ← 1
SF ← null
WHILE CPL <> END-OF-PROGRAM DO
    PS ← programming style in CPL
    IF ps in PS THEN
        PS[ps] ← PS[ps] + 1
Some technical notes for Script 6.8 are:

1. \(\leftarrow\) stands for the operation of assignment;
2. \(\leq\) stands for the operation of adding an element to a set;
3. SI can be fixed or flexible. In the system of this thesis which is dealing with C language, we let SI be flexible where each sampling interval is corresponding to the length of a procedure. In other words we give sampling to each procedure.

The programming style sampling function therefore operates by traversing the whole source program and collecting all kinds of programming styles and their number of occurrences for each procedure.
6.4.2 Program Style Filtering

A programmer can occasionally use other programming styles. This can be reflected in the programming style sampling function where there is more than one programming style occurring in one dimension for a particular function. A mechanism is therefore needed to remove “noise” from programming style sampling function.

One strategy is to set a threshold for the programming style sampling function. Those programming styles whose number of occurrences is lower than the threshold will be treated as noise and be filtered out. However, this could still lead to the situation where more than one programming style exist for a single programming dimension in a procedure. Nevertheless, this kind of “conflicting” information does not seem to be meaningless – a programmer may use more than one programming style in a dimension simultaneously and such information can also used to differentiate programmers.

To simplify the process, we adopted another strategy – for each dimension, we preserve only one single programming style which has the largest number of occurrences in a procedure. In other words, all other programming styles collected are filtered out.

In this case, the programming style filtering process can be supported by only a simple sorting algorithm. The algorithm is not listed here.

6.4.3 Program Cutting

Once a filtered programming style sampling function is obtained, it will be able to cut the program into modules based on some kind of criteria.

A source program is usually composed of multiple source program files; each source program file can be treated as a sequence of statements. The purpose of program cutting is to cut each source program file into a sequence of program sections where neighbouring program sections have different programming styles. In this way all the source program files will be broken into fractions that are ready for re-healing, to be discussed in Section 6.4.4.

It should be noted that some programming styles can be shared by different programmers. This is quite understandable since some common tendencies can be shared by different programmers and programming styles are a reflection of these tendencies.
The correct cutting points for a source program are the locations between the program sections where new programming styles are conflicting if they belong to the same dimension but have different pattern. Script 6.9 provides an algorithm for programming cutting. Some abbreviations are pointer of Sampling function (Sp), number of samples (SN), Cutting Points (CPs), Sampling Function (SF), Programming Style for single-line Comment (PSSC), Programming Style for multiple-line Comment (PSPN), Programming Style for Un-nest Indentation (PSUI) and Programming Style for Nested Indentation (PSNI).

**Algorithm for Program Cutting:**

```
Sp ← 1
CPs ← null
WHILE Sp <> SN DO
    IF ( SF [Sp] [PSSC] <> SF [Sp+1] [PSSC] ) OR
        ( SF [Sp] [PSVN] <> SF [Sp+1] [PSVN] ) OR
        ( SF [Sp] [PSUI] <> SF [Sp+1] [PSUI] ) OR
        ( SF [Sp] [PSMC] <> SF [Sp+1] [PSMC] ) OR
        ( SF [Sp] [PSPN] <> SF [Sp+1] [PSPN] ) OR
        ( SF [Sp] [PSNI] <> SF [Sp+1] [PSNI] )
    THEN
        CPs <= Sp
    ENDIF
    Sp ← Sp + 1
ENDWHILE
```

Script 6.9: Program Cutting

The initial program package is a complete one, so sampling function (Sp) is set to 1. Therefore the cutting point (CP) is null. If the number of Sp is not equal to number of sample (SN), six further conditions will be further considered: whether or not the entire number of programming styles for three different comment and three different indentations are identified. If not, the number of Sp+1 is set to Sp.
Chapter 6 Linear Programming Style based Program Partitioning

Note:
1. ← stands for the operation of assignment
2. <= stands for the operation of adding an element to a set
3. Programming Style (PS)
   - Current Program Line (CPL)
   - Sampling Function (SF)
   - Sample Interval (SI)
   - Pointer of Sampling Function (Sp)
   - Name of Samples (SN)
   - Cutting Points (CPs)
   - Programming Style for Single-line Comment (PSSC)
   - Programming Style for Multiple-line Comment (PSMC)
   - Programming Style for Naming of Variable (PSVN)
   - Programming Style for Naming of Procedure (PSPN)
   - Programming Style for Un-nest Indentation (PSUI)
   - Programming Style for Nested Indentation (PSNI)

The algorithm operates by traversing the filtered sampling function and collecting cutting points between program sections that have different programming styles. It should also be noted that if, in a program style sampling function, the number of occurrences of a programming style is zero, i.e., the programming style for that dimension is unknown, the dimension in that program region is allowed to be matched with any programming style in the same dimension of its neighbouring program sections.

By allowing an empty programming style to be successfully matched with any programming style, there arises a risk of mis-cutting source program. Suppose, we have three program sections, A, B and C. All these three program sections are subsequently connected together. A and B have distinctively different programming styles while B has most of its dimensions empty. If B can both match A and C, these three program sections will be put into one programming section. This is an undesirable result since what we want is to separate A from C. theoretically unacceptable, in practice, however, such adverse cases are rare and the benefit making empty programming style a universal matcher is significant: most program sections do not have to present programming styles in all kinds of dimensions. For example, a program section may not use comments at all. In this case, we should avoid cutting program sections unnecessarily.
Once a program is cut, each cut section of the program is indexed with the valid
programming style patterns; each dimension is specified only one pattern (or null of
there is no valid pattern). If there exist multiple patterns for one dimension, the most
significant one will be used to index that section of the program.

6.4.4 Program Re-healing

The cutting of a program is done in a continuous program space, whereas a self-
contained program module can spread across separated program regions or even source
program files. For example, an application contains several functional modules, each of
these modules provides a unique service to the user through a Graphical User Interface
(GUI). The functions related to the GUI can be collected in separate source program
files, while these functional modules are spread across multiple source program files. In
order to understand the services provided by this application, it is therefore needed to
collect the source program sections belonging to each functional module, including the
program sections in the GUI files.

Program re-healing is aimed to collect program sections that belong to an individual
program module together. This is conducted by classifying the program sections
produced by the program cutting algorithm mentioned in Section 6.4.3 into multiple
categories. Each category, collectively, has a unique programming style, i.e., at least
one dimension has a unique pattern.

Let $PD_1, ..., PD_m$ be different dimensions, $nps_i, 1 \leq i \leq m$ be the number of possible
patterns for programming style in $PD_i$, there will be, at most, $nps_1 \times nps_2 \times ... \times nps_m$
different categories in which cut program sections can be put. The program re-healing is
therefore a simple dispatch process where each program section is dispatched into a
suitable category based on its programming styles.

Once program re-healing is done, the program modules obtained by uniting multiple
program sections need to be assigned overall programming style patterns. This is done
by taking the entire non-null individual programming style pattern and uniting them
together.
6.4.5 Enlarging the Granularity of Partitioned Program Modules

After re-healing the program sections, each program module contains only program sections whose distance from each other is 0. Since a programmer may slightly change his/her programming style when writing different program sections, it would be beneficial if we can further merge those program modules whose distance is within a certain specified value. In this way the granularity of partitioned program modules can increase and a more meaningful program partitioning result will be obtained.

Task 1: merge programming style categories whose distance is ≤ 1. This can be done in the following two steps.

Step 1: generate all the combinations of programming style categories and preserve those combinations within which the distance between every two programming style categories is 0.

Step 2: pick up the largest preserve combination of programming style categories and delete those preserved combination of programming style categories in which at least one programming style category has been picked up; pick up the second largest preserved combination of programming style categories in which at least one programming style category has been picked up; ...; this process will continue until there is no more preserved combination of programming style categories.

Task 2: merge programming style categories whose distance is ≤ n.

The process of Task 2 is the same as the one described above if we change the distance checking criteria from “distance is ≤ 1” to “distance is ≤ n”.

Once the organisation of programming style categories based on the distance is decided on, the actual merging of partitioned program modules into a larger trunk of program modules involves only dispatching them into the combination of programming style categories in which programming style category can be matched.

6.4.6 Complexity of the Algorithm

Programming style based program partition is concluded by the discussion of the computational complexity of this new program partitioning method. Using this method,
a source program can be partitioned into business intelligence oriented programming modules at a linear time-scale. The reasons are as follows.

- **Checking Programming Style Pattern.** At each relevant program construct, several program style patterns associated with a dimension are checked with the program construct. The number of dimensions and the number of programming style patterns in each dimension are both bounded. The computation complexity of checking each programming style pattern with the program construct is linear with the size of the program construct which has bounded size and therefore is constant compared to the size of the source program. The overall complexity for checking the programming style of a program construct is therefore a constant.

- **Programming Style Sampling.** At the stage, the source program is traversed in a way that each program element is visited only once. Since the computational complexity of checking the programming style of a program construct is a constant compared to the size of a source program, the computational complexity for generating programming style sampling function is therefore linear with the size of the program.

- **Programming Style Filtering.** At this stage, programming style sampling function is traversed and the most significant programming style is each dimension is persevered. Since the number of dimensions and programming style in each dimension are both bounded, the computational complexity for picking up the programming styles which has the largest number of occurrences in a single procedure is therefore a constant compared to the size of the source program. The size of programming style sampling function is at the same scale with the size of the source program and therefore the overall complexity of programming style filtering is linear with the size of the program.

- **Program Cutting.** At this stage, filtered programming style sampling function is traversed to set a cutting point between two neighbouring program sections have different programming styles involves only a bounded number of comparisons and therefore is a constant compared to the size of the source program. The overall complexity for program cutting is therefore linear with the size of the source program.

- **Program Re-healing.** At this stage, programming sections are dispatched into bounded number of programming style categories. The decision of putting a
program section into a category involves a bounded number of comparisons and is therefore a constant compared to the size of the source program. The overall complexity of program re-healing is therefore linear with the size of the source program.

- **Enlarging the Granularity of the Partitioned Program Module.** At this stage, program modules are merged into larger trunks of program modules based on the distance between these program modules. Since we have a bounded number of programming style categories, the number of combinations of these programming style categories is also a constant. The task of merging partitioned program modules into larger trunks of program modules is a simple dispatch task and the overall complexity is linear with the size of source program.

Since the overall source program partitioning algorithm is composed of the steps of Programming Style Sampling, Programming Style Filtering, Program Cutting, Program Re-healing and Enlarging the Granularity of Partitioned Program Module subsequently and the computational complexity of each of these steps is linear with the size of the source program, the overall computational complexity of the source program partitioning algorithm is therefore linear with the size of the source program. In Section 11.3 of Chapter 11 we have dedicated a case study to evaluate the effectiveness of this method.

It should be finally noted that the new programming style based program partitioning method is never aimed to completely replace other program partitioning methods. Different methods have both strengths and weaknesses. There are opportunities where this method can, combined with other methods, provide a more satisfactory program partitioning result. For example, this linear program partitioning method is used in proposing caudate program modules at the first stage and other methods are used to further validate the partitioned program modules based on other criteria such as the degree of coherence and coupling. This is beyond the scope of this thesis.

### 6.6 Summary

In this chapter, an innovative linear program partitioning method has been introduced. It is based on programming styles. In particular, the styles of three kinds of program elements – program comments, naming convention and program indentation – are used
to distinguish programmers from each other and therefore partition the source program into business intelligence-oriented program modules. The whole program partitioning algorithm is composed sequentially of the following steps: programming style sampling, program style filtering, program cutting, program re-healing, and enlarging the granularity of partitioned program.
Chapter 7

Business Intelligence Concept Recovery

In Chapter 6 initiatives were proposed to partition the business intelligence base into business intelligence slices and a source program into business intelligence-oriented program modules respectively. The task of recovering business intelligence from source program is therefore transformed into the task of recovering business intelligence slices from program modules. Essentially the process of recovering business intelligence concepts from program modules is a process of matching between business intelligence slices and program modules.

Once the concepts in a program module have been recovered, the relationship between these concepts must be generated also. This is done in a relationship recovery module where relationship recovery rules are used to extract relationships from program constructs such as a sequence of procedure calls, a single procedure call, construct information, database schema, etc. Since program constructs can indicate relationship at only structural level, we call these recovered relationships raw relationships. Generally speaking legacy software system contain not only operational code like C but also description code such as database schema; in this thesis we address the issues of business intelligence recovery from both C source program and database schema. Section 7.3 and Section 7.4 will describe the generation of raw relationships between recovered concepts in a C program module and a database schema respectively.

The following sections are organised as follows. In Section 7.1 concepts from a C program module are recovered based on concept recovering rules. In Section 7.2 concepts from a database schema are recovered. Section 7.3 introduces concepts relationship generation in a program module. Section 7.4 introduces concepts relationship generation in a database schema.
7.1 Recovery Concept from Program Modules

In Section 5.4 of Chapter 5 concepts in the context are classified into objects and actions. An object is usually corresponding to a variable style name in a program module; while an action can find its counterpart as a function name in the source comments, where action and object can occur. These are not reliable sources from which to recovery concepts, since the program comments may not be updated and therefore contain incorrect information about the program. Also program comments are highly unstructured information sources which name it difficult to recover information from them.

In approach, only procedure names and variable style names are considered to be the candidates from which to recover domain concepts. The first step is to collect all the procedure names and variable style names from the program module. Once this has been done, concept recovery rules can then be applied to each of these names to recover concepts lattice, which is the physics representations.

A single business intelligence concepts can be represented as either an atomic name or compound name connected by symbols like '_' or '-', etc. Each concept can be associated with a set of concept recovery rules. These concept recovery rules are classified into direct matching, irregular atomic name recovery rules, regular compound name recovery rule and irregular compound name recovery rule.

7.1.1 Direct Matching

This is used in the situation where a variable style/function name in the program module can be completely matched by a concept or its alias in the data dictionary. Once such a match is found, the location of the name in the program module and the concept in the data dictionary will be recorded, and 1.0 is assigned to this concept to describe the belief in it.

The other rules deal with partial matches between concepts or its alias in the data dictionary and the abbreviated form of variable style/function names in the program module.
7.1.2 Regular Atomic Name Recovery Rule

This refers to a set of concept recovery rules that are universally applicable to all the atomic names. One example for regular atomic name recovery rule can be read as “first three letters, 0.30” which means if the first three letters of a concept or its alias is found in a program module, the concept will be recovered and become a candidate concept for this program module with a belief value assigned to be 0.3.

7.1.3 Irregular Atomic Name Recovery Rule

This deals with those atomic names whose abbreviations are formed by the pronunciation of the names ad for which regular atomic name recovery rules can not be easily designed. Different from a regular atomic name recovery rule that is applicable to call the atomic names, an irregular atomic name recovery rule is only applicable to one particular atomic name. The rules for name recovery are:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example</th>
<th>Belief</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Matching</td>
<td>Completely match</td>
<td>Address (address)</td>
</tr>
<tr>
<td>Regular Atomic Name</td>
<td>First five letters</td>
<td>promo (promotion)</td>
</tr>
<tr>
<td>Irregular Atomic Name</td>
<td>One to one</td>
<td>Stmtnt (statement)</td>
</tr>
<tr>
<td>Regular Compound Name</td>
<td>Completely Match</td>
<td>email_push (Ø)</td>
</tr>
<tr>
<td>Irregular Compound Name</td>
<td>Mix Match</td>
<td>Fname (first name)</td>
</tr>
</tbody>
</table>

Table 7.1: Business Intelligence Concept Recovery

7.1.4 Regular Compound Name Recovery Rule

This is applied by checking each of its composite atomic names using atomic name recovery rules. Let k be the total number of atomic names in a given name N that is to be checked; C = (C₁, ..., Cₙ), where C is the compound name of a concept, (C₁, ..., Cₙ) be n composite atom name in C. Let (Cᵣ(₁), ..., Cᵣ(m)) be r(m) atomic names within C that have been recovered, where 1 ≤ r(i) ≤ n, 1 ≤ i ≤ m; CF(Cᵣ(₁)), ..., CF(Cᵣ(m)) be
the belief of these recovered names; \( CF(C) \) be the belief of \( C \). \( CF(C) \) can therefore be calculated as:

\[
CF(C) = \frac{\sum_{i=1}^{m} CF(C_{r(i)})}{k + n - \sum_{i=1}^{m} CF(C_{r(i)})}
\]

This rule suggests that the percentage of match between \( N \) and \( C \) is used to indicate the belief of Concept \( C \) given a name \( N \).

### 7.1.5 Irregular Compound Name Recovery Rules

These are a collection of commonly-used abbreviations of compound names from which generalised rules are hard to design, for example:

<table>
<thead>
<tr>
<th>Compound Name</th>
<th>Composite Names</th>
<th>Irregular Cases</th>
<th>Brief</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business-Name</td>
<td>Business, Name</td>
<td>BN</td>
<td>0.7</td>
</tr>
</tbody>
</table>

### 7.1.6 Process of Recovering Concepts from a Program Module

The general algorithm for recovering concepts from a program module proceeds by traversing the data dictionary in an attempt to recover concepts one by one. To recover a single concept, direct matching rule, all the regular name rules and a set of irregular name recovery rule for this concept will be applied to this concept. When executing one of these rules, the transformed name of this concept will traverse all the function names and variable style names in the program module trying to find matches. Once a match is found, the location of the matched name in the program module and the value of belief for that rule will be recorded. In our system, the highest level of belief is given to the direct matching rule, and generally, irregular name recovery rules have a higher level belief than regular name recovery rules. If none of these rules for a concept has been
matched in a program module, 0 will be assigned to the belief in this concept which indicates we do not know whether or not the concept exists in the program module. This is a reasonable assumption since the data dictionary has not contained a complete concept in an application and we do not want to reject any concept that has not been recovered at this stage. This is also the reason why there is no negative value of belief in recovered concepts.

For each concept in the data dictionary we have collected various aliases for the concept. Each alias is a complete name. The irregular cases of the concept are also recorded in the data dictionary. By using the data dictionary and the name recovery rules, we can recover more names from the program module.

7.2 Recovering Concepts from Database Schemas

A large number of database applications fall into the category of legacy systems. Similar to C source program, there is also a great need to recover business intelligence from legacy database applications. Once such kind of business intelligence has been recovered, a better understanding of the database application can be made. The recovered business intelligence can also help re-engineer the legacy database application to a new database platform.

A typical database application usually consists of two parts: database definition part and database operation part. Since the database definition part is the most faithful place in a database application to recover business intelligence, in this thesis, the research focuses on this part.

The database definition part generally contains a set of schemas; each schema contains business intelligence and data in a sub-business intelligence; a schema is usually composed of multiple tables, each table contains a slice of business intelligence and data in the sub-business intelligence; a table is constructed with a list of field names describing certain attributes and multiple rows of data associated with these field names.

Generally speaking, the modular structure of a database program is much clearer than that of a C program. The reason is that in a database program we have reserved words such as schema, table and field that explicitly define the modular structures at different levels of granularity; in a C program, however, the reserved words like \textit{struct} and procedure cannot sufficiently distinguish the level of granularity of the program modules.
Given the above fact, the partitioning of a database definition part into business intelligence oriented modules becomes straightforward: we can simply treat each database schema as a program module.

In order to recover business intelligence from a database schema, it is needed to decide the candidate names in a database schema where concepts can be recovered. In this thesis, let these candidate names be schema names, table names, and field names.

This is a reasonable assumption, a single database schema is usually associated with a small application domain and is designed by a single programmer. Different attributes across tables are usually given different names.

Once the candidate names have been collected, the process of recovering concepts from these names will be the same as the process described in Section 7.1.

It should be noted that since database schema have no operational semantics, all the concepts recovered from a database schema are objects.

7.3 Generating Raw Relationships among Concepts in a Program Module

Since the recovered concepts are only associated with variable style names and procedure names in a C program module, therefore it is needed to consider only the relationships between viable style names and procedure names. In particular, there are three groups of relationships: variable style to variable style, procedure to procedure, and variable style to procedure.

7.3.1 Variable Style to Variable Style

Variable Styles represent a kind of static semantics of the program. The whole application domain can be though as a root variable style, which is further divided into variable styles at the secondary level representing sub-domains; the secondary level variable styles are, in turn, divided into the third level variable styles, ... until the nth level variable styles are created. Normally all the variable styles at these n levels exist in a program module; these variable styles not only characterise the state of the application but also act as glue to connect operations of sub-domains at different abstract levels.

Given two variable styles, it is therefore important to identify whether one variable style
at an abstract level is dominated by the other one at a higher abstract level, i.e., one variable style is component of the other variable style. In the C language, the most faithful program construct from which to derive the dominance relationship between two variable styles is `struct` that is shown as follows.

```c
struct T1{
    T11 v1;
    T12 v2;
    ...
}
```

In the above case, T11 and T12 are directly dominated by T1. The dominance relationship is also transitive, i.e., if A dominates B and B dominates C then we have A dominates C. We therefore need to traverse multiple `struct` statements in a program module in order to fully check the dominance relationship between two variable styles.

Let us extract all the direct dominance relationships from the `struct` statements and connect them into a single graph connected by the common objects they share. Let A and B be two variable styles in this graph.

<table>
<thead>
<tr>
<th></th>
<th>A is dominated by B</th>
<th>A is not dominate by B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B is dominated by A</td>
<td>NA</td>
<td>-1</td>
</tr>
<tr>
<td>B is not dominated by A</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1: Belief of Dominated (A,B) in a Tree

<table>
<thead>
<tr>
<th></th>
<th>A is dominated by B</th>
<th>A is not dominate by B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B is dominated by A</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>B is not dominated by A</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.2: Belief of Dominated (B,A) in a Tree

**Assumption 1:** this graph is a directed tree. Since each node in a directed tree or a directed forest has maximum of one parent node, there exist only three possible
relationships between A and B: (1) A dominates B; (2) A is dominated by B and (3) the relationship between A and B is unknown. It can be seen that no conflict arises between A and B in this scenario. This conflict in this context refers to the situation where there is evidence suggest that A dominates B and B dominates A.

In order to assign a value to the relationships between A and B, *Dominated (A, B)* and *Dominated (B, A)*, we need to go through the following steps.

Step 1: Checking whether A is dominated by B. We start by locating A in a tree and traverse back towards the root of the tree; if B is found in the path towards the root then we mark ‘A is dominated by B’; if B is not found in the path then we temporarily mark ‘A is not dominated by B’. We cannot decide the value of *Dominated (A, B)* or *Dominated (B, A)* yet, since more evidence needs to be obtained.

Step 2: Checking whether B is dominated by A. This step is similar to Step 1. If A is found in the path towards the root then we mark ‘B is dominated by A’; if A is not found in the path then we temporarily mark ‘B is not dominated by A’.

Based on the evidence gathered from the steps above we can now assign beliefs to *Dominated (A, B)* and *Dominated (B, A)* which are shown in Table 7.1 and Table 7.2 respectively.

From Table 7.1 we can see that (1) if A is dominated by B and B is not dominated by A then *Dominated (A, B)* is given the highest level of belief, i.e., true; if A is not dominated by B and B is dominated by A then *Dominated (A, B)* is given the lowest level of belief, i.e., false; if A is not dominated by B and B is not dominated by A then *Dominated (A, B)* is given a neural level of belief, i.e., ignorance. This is in accordance with intuition. NA indicates no such situation can occur where A is dominated by B and B is dominated by A. Similar explanations apply to Table 7.2.

Suppose the size of a program module is n, the height for the tree is, on average, \(O(\log(n))\). In other words the computational complexity of step 1 and step 2 is in logarithmic order with the size of the program module.

Although assuming the graph is a tree or a forest can bring simplicity to analysis, this assumption is too strong and the graph can be an acyclic graph where a node may have more than one parent.

<table>
<thead>
<tr>
<th></th>
<th>A is dominated by B</th>
<th>A is not dominate by B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B is dominated by A</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
Assumption 2: this graph is a directed acyclic graph. Although a node in a directed acyclic graph may have more than one parent, no such situation can occur where there is a loop in the graph. In other words, if A is dominated by B the B cannot be dominated by A, i.e., no conflict between A and B can occur. The principle of assigning value to the beliefs of $\text{Dominated}(A,B)$ and $\text{Dominated}(B,A)$ is therefore the same as Assumption 1 as shown in Table 7.3 and Table 7.4.

The difference between the situation of three and acyclic graph are the procedures to check the dominance relationship between A and B. For an acyclic graph, a node may have multiple choices of path to the root. This can be done by a typical acyclic traversing all the possible paths from A to the root. This can be done by a typical acyclic traverse algorithm where a recursive function can be arranged. Once a node has been visited it will be marked as ‘visited’ so that the later attempts to visit it can be suppressed. Suppose the size of a program module is n, the computation complexity of traversing a directed acyclic graph is in the same order with n, i.e., $O(n)$.

<table>
<thead>
<tr>
<th></th>
<th>A is dominated by B</th>
<th>A is not dominate by B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B is dominated by A</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>B is not dominated by A</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.5: Belief of Dominated $(A, B)$ in a Directed Acyclic Graph
<table>
<thead>
<tr>
<th></th>
<th>A is dominated by B</th>
<th>A is not dominate by B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B is dominated by A</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B is not dominated by A</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.6: Belief of Dominated (B, A) in a Directed Acyclic Graph

**Assumption 3: this graph is a directed cyclic graph.** A direct cyclic graph is the general form of an arbitrary directed graph, where a node may have multiple parent nodes and a node may eventually revisit itself by simply following the directed links. Although it sounds a bit more complex than directed acyclic graph, the algorithm for checking whether A is dominated by B or B is dominated by A is the same as that of a directed acyclic graph. The only difference between the two situations is the way beliefs in the relationships $\text{Dominated} (B, A)$ and $\text{Dominated} (A, B)$ are specified. Table 7.5 and Table 7.6 show the details.

Since in a directed cyclic graph, a conflict between A and B can occur, i.e., A is dominated by B and B is dominated by A, the value of $NA$ in Assumption 1 and Assumption 2 is replaced with 0. This suggests our view that when conflict occurs, we hold it in an attempt to resolve it later as more evidence is available.

### 7.3.2 Procedure to Procedure

Procedure names are associated with actions in an application domain. Similar to objects that can have a hierarchy, actions in an application domain also have a hierarchical structure, where the action at the second abstract level to fulfil its task and the sub-actions at the second abstract level can be further divided into sub-actions at the third level, and so on. It is therefore important to check whether one procedure name is a sub-procedure of another procedure. Let A and B be two procedure names, we say A is dominated by B if A is a sub-procedure of B. Formally we denote this as $\text{Dominated} (A, B)$.

Another important relationship between two actions is the execution order of the two. Given two procedure names A and B we denote $\text{Before} (A, B)$ is A needs to be executed earlier than B.
In the C language, the most faithful program construct from which the two relationship mentioned above can be derived is procedure definition that is shown as follows.

P1(...){
  P11(...);
  P12(...);
  ...
}

In the case above, we can derive three relationships: Dominated \((P_{11}, P)\), Dominated \((P_{12}, P)\) and Before \((P_{11}, P_{12})\). Moreover, \(P_{11}\) and \(P_{12}\) are directly dominated by \(P_{1}\). \(P_{1}\) is therefore called the parent node of \(P_{11}\) and \(P_{12}\); \(P_{11}\) is the immediate brother node before \(P_{12}\).

Let us extract all the direct dominance relationship from the procedure definitions and connect them into a single graph linked by the common actions the share. In constructing such a graph, the sequent orders of brother nodes remain. Let \(A\) and \(B\) be two procedure names.

Since the dominance relationship between two procedure names is exactly the same as the between two variable styles as described in Section 7.1.1, we will not discuss it any further. The algorithms for checking dominance relationship and the table for assigning belief value can be used without change.

In this section we focus on checking the relationship of Before between two procedure names.

In a C procedure, procedure calls can occur in three kinds of (nested) flow control statements such as sequential, loop, and if-else as shown below.

\[
P1();
\]

...  

\[
P2();
\]

(a) Sequential

\[
if (E1){
  P1();
} else {
\]
P2 ();
}

(b) Choice

while (E){
P1 ();
...
P (2);
}

(c) Loop

For (a) there exist a subsequent control between P1 and P2 and it can be inferred that Before \((P1, P2)\) is true. For (b) the control flow is broken into two parallel sub-flows for P1 and P2 and joined together at the end of the if statement; in this case, Before \((P1, P2)\) is unknown. For (c), there exist a loop between P1 and P2, let us take the following two steps:

Step 1: start from P1 and trace back in the control flow towards the entry point of P. If there exist multiple paths between P1 and the entry points then all these paths will be traversed. If P2 is found in the path then we say that “P2 occurs before P1”; otherwise we say “P2 does not occur before P1”.

Step 2: similarly, start from P2 and trace back towards entry point of P. If P1 is found in the path then we say that “P1 occurs before P2” and “P2 does not occur before P1” can we say that Before \((P1, P2)\).

Let \(P_i\) be a procedure name, \(P_{i+1,1}, \ldots, P_{i+1,j-1}, P_{i+1,j}, P_{i+1,j+1}, \ldots, P_{i+1,j+j_2}\) be \(j + j_2\) procedure calls occurring in \(P_i\), where \(P_{i+1,1}, \ldots, P_{i+1,j-1}\) be \(j - 1\) procedure calls that have Before \((P_{i+1,k}, P_{i+1,j})\), \(1 \leq k \leq j - 1\) and Before \((P_{i+1,j}, P_{i+1,j+k})\), \(1 \leq k \leq j_2\). The set of procedure names that occur before \(P_{i+1,j}\) is therefore the union of \(P_{i+1,1}, \ldots, P_{i+1,j-1}\), all the procedure names dominated by \(P_{i+1,1}, \ldots, P_{i+1,j-1}\), all the procedure names \(P_i\) for which Before \((P_i, P_i)\) is true.

This is recursive definition. The calculation of all the procedure names dominated by a procedure name A is straight-forward: start from A, trace down towards the leaves direction, traverse all the paths A can reach and collect all the procedure names it reaches. The collection of procedure names are those dominated by A.
Chapter 7 Business Intelligence Concept Recovery

<table>
<thead>
<tr>
<th></th>
<th>B is before A</th>
<th>B is not before A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A is before B</td>
<td>NA</td>
<td>-1</td>
</tr>
<tr>
<td>A is not before B</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.7: Belief of Before (B, A) in a Tree

<table>
<thead>
<tr>
<th></th>
<th>B is before A</th>
<th>B is not before A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A is before B</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>A is not before B</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.8: Belief of Before (A, B) in a Tree

Assumption 1: the dominance relationship in this graph forms a directed tree and the before relationship among the nodes under each node forms a fully ordered chain. Since all the node in a tree are fully ordered by the relationship of before. We need traverse only those procedure names that occur before A to search B. The computational complexity is in the same order as the size of the program module n. Table 7.7 and Table 7.8 are put together in a similar way to those in Section 7.1.1

Some explanations for Table 7.7 are:

- Since a tree is fully ordered by the relationship of before, B is before A and A is before B cannot both be true.
- The situation of B is not before A and A is not before B is where A and B are in different threes in the program module. The sequent order between A and B is therefore unknown.

<table>
<thead>
<tr>
<th></th>
<th>B is before A</th>
<th>B is not before A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A is before B</td>
<td>NA</td>
<td>-1</td>
</tr>
<tr>
<td>A is not before B</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.9: Belief of Before (B, A) in an Acyclic Graph
Table 7.10: Belief of Before (A, B) in an Acyclic Graph

<table>
<thead>
<tr>
<th></th>
<th>B is before A</th>
<th>B is not before A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A is before B</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>A is not before B</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Assumption 2: the dominance relationship in this graph forms a directed acyclic graph and the before relationship among nodes under each node forms a directed acyclic graph. Since a node in a directed acyclic graph can have multiple parent nodes, the nodes in the graph are not fully ordered any longer. Moreover, conflict could arise when both A is before B and B is before A claim to be true. In the worse case, we need to traverse the whole program module to check whether B occurs before A. Table 7.9 and Table 7.10 contain the decision on assigning value to the belief of Before (B, A) and Before (A, B).

It can be noticed that NA is changed to 0, indicating the existence of conflict and our tactic to tolerate the initial conflict.

Assumption 3: the dominance relationship in this graph forms a directed cyclic graph and the before relationship among nodes under each node forms a directed cyclic graph. The way of handling Assumption 3 is exactly the same as that of Assumption 2.

7.3.3 Variable Styles to Procedure

A variable style indicates an object in an application domain and a procedure name indicates an action. An application domain is usually modelled as a set of objects and a set of actions operating on these objects. Given a procedure name, it is therefore important to know which variable styles are related to the procedure names.

In the C language, the most faithful program construct to derive the relationship between variable styles and procedure names is the header part of a procedure definition.

`T0 P(T1, T2, ..., Tm)`
In the case above, two kinds of relationship can derived: (1) $T_1, T_2, \ldots, T_m$ are direct input variable styles of $P$; (2) $T_0$ is direct output variable style of $P$. We denote these two kinds of information as *Input* ($T_1, P$) and *Output* ($P, T_0$) respectively.

Let us extract the direct input and output relationships from the headers of all the procedure definitions and connect them into a single graph, where variable styles and procedure names are linked by the direct dominance relationships between variable styles mentioned in Section 7.1.1; direct dominance relationships and before relationship between procedure names mentioned in Section 7.7.2 are all preserved.

Besides the situation mentioned above where a variable style $T$ can be related to a procedure name $P$, there also exist other ways that they can be related, for example,

- A variable style that dominates $T$ is the direct input/output of $P$. In this case, it can be concluded that $T$ is also the input/output of $P$.
- $T$ is the direct input/output of a procedure name that is dominated by $P$. Generally speaking, it cannot be made certain whether $T$ is also an input/output of $P$. But if the procedure name that is dominated by $P$ is the first/last some or grandson of $P$ then we can conclude that $T$ is also the input/output of $P$.

In this thesis, we are only interested in the situation where a variable style is the input/output of a procedure name, since this information can give us the view of the interface of a procedure names. We denote these two kinds of information as *Input* ($T_1, P$) and *Output* ($P, T_0$) respectively.

The process of checking whether $T$ is a direct input/output of $P$ is described as follows. Let $TP (T_1, T_2, \ldots, T_m, \ldots)$ be the header of the procedure $P$, $T_i, 1 \leq i \leq m$ be $m$ input variable styles of $P$, $T_0$ be the output variable style of $P$; $P_1, P_2, \ldots, P_n$ be $n$ procedure calls in $P$.

**Step 1:** Check whether $T$ is an input of $P$. The set of variable styles that are input of $P$ is the union of the following sets: $T_i, 1 \leq i \leq m$, the variable styles dominated by $T_i, 1 \leq i \leq m$, the input variable styles of the first son $P_1$ of $P$.

This is a recursive definition. Once $T$ is found in the union set constructed above, then we mark ‘$T$ is an input style variable style of $P$’.
Step 2: Check whether T is an output of P. The set of variable styles that are output of P is the union of the following sets: $T_0$, the variable styles dominated by $T_0$, the output variable styles of the last son $P_n$ of $P$.

This is again a recursive definition. Once T is found in the union set constructed above then we mark ‘T is an output variable style of P’.

We call the following situation a conflict between T and P if T is an input/output and T is also the output/input of P. This is in accordance with intuition – an action should fulfil some meaningful operation; if the action takes a domain object as input, it should produce a different domain object as output.

<table>
<thead>
<tr>
<th></th>
<th>T is the input of P</th>
<th>T is not the input of P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T is the output of P</td>
<td>NA</td>
<td>-1</td>
</tr>
<tr>
<td>T is not the output of P</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.11: Belief of Input $(T, P)$ in a Directed Tree

<table>
<thead>
<tr>
<th></th>
<th>T is the input of P</th>
<th>T is not the input of P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T is the output of P</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>T is not the output of P</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.12: Belief of Output $(P, T)$ in a Directed Tree

**Assumption 1:** both the dominance relationship between variable styles and the dominance relationship between procedure names in this graph form directed trees. Since each node in the tree has, at most, one parent node, we only need to search the sub-trees of variable styles whose roots are direct input/output of P, the first/last son of P, the first/last grandson of P, etc. Also noted is that in this case there is also a conflict between T and P, i.e., if T is the input/output of P, it cannot be the input/output of P at the same time. Table 5.11 and Table 5.12 present the decision to assign value to the belief of Input $(T, P)$ and Output $(P, T)$. 
Assumption 2: the dominance relationship between variable styles and the dominance relationship between procedure names in this graph form directed acyclic graphs. Since in an acyclic graph a node can have multiple parent nodes, a situation can arise where an object $O$ is dominated by another object $O_1$ which is the input of an action $A$ and $O$ is also dominated by an object $O_2$ who is the output of $A$. In this situation, a conflict between $O$ and $A$ arises. This, however, is not counter-intuitive; $O$ can be a minor component of both $O_1$ and $O_2$ and what $A$ is interested in could be other components in $O_1$. In this situation, $O$ may simply have no influence on $A$, i.e., $O$ is not important to $A$. In this case a neural belief is given to the relationship between $O$ and $A$. Table 7.13 and Table 7.14 present the decision on assigning value to the belief in $Input (T, P)$ and $Output (P, T)$.

Assumption 3: the dominance relationship between variable styles and dominance relationship between procedure names in this graph form directed cyclic graph. In this case the discussion is the same as Assumption 2.
7.3.4 Process of Generating Raw Relationships between Recovered Concepts

In Section 7.1 it is described that the process of recover business intelligence concepts from a program module. Once we obtained these recovered business concepts, raw relationships between business concepts will be generated. Let $C_1, C_2, ..., C_m, C_{m+1}, ..., C_{m+n}$ be $m+n$ business concepts recovered from the program module; $T_1, T_2, ..., T_m, T_{m+1}, ..., T_{m+n}$ be $m$ variable style names and $n$ procedure names associated with the recovered concepts. Table 7.15 is therefore generated and needs to be filled in.

Since only relationship between two elements are generated, the table is therefore two dimensional. In this thesis we do not consider a relationship between a single element, hence the diagonal line of the table is invalid. In Table 7.15 T-T, P-P and T-P stand for the relationship between variable style and variable style, procedure and procedure, and variable style and procedure respectively. The taxonomy of these kinds of relationship and the process of checking these relationships have been introduced in Section 7.1.1, Section 7.1.2 and Section 7.1.3 separately. In particular we need to check $Dominated (T_i, T_j)$ and $Dominated (P_i, P_j)$ where $1 \leq i \leq m, 1 \leq j \leq m$ for T-T; $Dominated (P_i, P_j)$, $Dominated (P_j, P_i)$, $Before (P_i, P_j)$, $Before (P_j, P_i)$ where $1 \leq i \leq m+n, m+1 \leq j \leq m+n$ for P-P; $Input (T_i, P_j)$ and $Output (P_j, T_i)$ where $1 \leq i \leq m, m+1 \leq j \leq m+n$ for T-P. This can be shown in Table 7.16.

Although in each of Section 7.1.1, Section 7.1.2 and Section 7.1.3 we list three assumptions, i.e., directed tree, directed acyclic graph and directed cyclic graph for the graph extracted from the program module, in practice we assume the graph is directed cyclic that is of a more generic form than the other two.

Once the graph is traversed in a way that the algorithm for checking each relationship requires, the value of the belief in the relationship between two concepts will be filled in Table 7.16. This table will be used later for generating business intelligence to be introduced in Chapter 11.

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>...</th>
<th>$T_m$</th>
<th>$P_{m+1}$</th>
<th>...</th>
<th>$P_{m+n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>NA</td>
<td>$T-T$</td>
<td>$T-T$</td>
<td>$T-P$</td>
<td>$T-P$</td>
<td>$T-P$</td>
</tr>
</tbody>
</table>
### Table 7.15: Raw Relationship Table for Recovered Business Intelligence Concepts (1)

<table>
<thead>
<tr>
<th></th>
<th>( T )</th>
<th>( P )</th>
<th>( T )</th>
<th>( T )</th>
<th>( T )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_m )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( P_{m+1} )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>( T )</th>
<th>( T )</th>
<th>( T )</th>
<th>( P )</th>
<th>( P )</th>
<th>( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{m+n} )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
</tr>
</tbody>
</table>

Table 7.16: Raw Relationship Table for Recovered Business Intelligence Concepts (2)

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T )</th>
<th>( P_{m+1} )</th>
<th>( P_{m+n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>NA</td>
<td>Dominated, Before</td>
<td>Input, Before</td>
<td>Input, Before</td>
</tr>
<tr>
<td>( T )</td>
<td>Dominated</td>
<td>NA</td>
<td>Input, Before</td>
<td>Input, Before</td>
</tr>
<tr>
<td>( P_{m+1} )</td>
<td>Output</td>
<td>Output</td>
<td>NA</td>
<td>Dominated, Before</td>
</tr>
<tr>
<td>( P_{m+n} )</td>
<td>Output</td>
<td>Output</td>
<td>Dominated, Before</td>
<td>NA</td>
</tr>
</tbody>
</table>

Let \( n \) be the size of the program module, the size of Table 7.16 is of the same order as the size of \( n \). The checking of each relationship in Table 7.16, normally, will traverse a space that is the same order of the size of \( n \). The overall computation complexity for generating raw relationships for recovered business concept is therefore \( O(n^2) \).

Remember that in this thesis we consider only relationship between business concepts. Generally if we consider relationships among \( m \) business concepts, the computational complexity for checking these relationship in the program module will be \( O(n^m) \). Here we see the benefit of partitioning source program into program modules.
7.4 Generating Raw Relationships between Concepts in a Database Schema

In Section 7.1 we discuss the process of recovering business intelligence concepts from a database schema. Once the concepts have been recovered, the raw relationships between each two concepts can be established. Since all the recovered concepts in a database schema are objects, the raw relationships are therefore relationships between object and objects.

The most faithful program constructs for deriving relationships between two names in a database schema are schema definitions and table definitions. A schema definition is shown as follows.

```
Schema S {
  Table T1;
  Table T2;
  ...
}
```

For the schema above we can derive that T1 and T2 are directly dominated by S. A table definition looks like:

```
Table T{
  Field F1;
  Field F2;
  ...
}
```

From this table definition we can derive two kinds of relationship: (1) F1 and F2 are directly dominated by T; (2) F1 and F2 are directly grouped together.

The rationale for the above scenario is stated as follows. A table name in a database schema usually represents an entity in an application domain and the filed names within the table represent the attributes associated with the entity. It is meaningful to let an attribute be dominated by an entity it belongs to and let the attributes that belong to the
same entity grouped together. In this way close relationship between the entity and the
attributes can be established.

A schema name can represent an application domain and the entities in an application
domain are usually loosely coupled together. Although it is useful to identify the
domination relationship between the names of the application domain and the names of
entities contained in it, the entities do not have to be grouped together.

Let us extract all the direct domination relationships from the database schema
definition and connect them into a single graph. Let $N_1$ and $N_2$ be two names in the
graph. Let us say $N_1$ is dominated by $N_2$ if there is a path of direct domination
relationships between $N_1$ and $N_2$. Formally we denote this kind of relationship as
$Dominated (N_1, N_2)$. It can be seen that this graph can be either a directed tree or a
directed acyclic graph, i.e., it cannot be a directed cyclic graph. The reason is that a
name cannot occur as (1) both a field name and table name or a schema name; (2) both a
table name and a schema name.

The process of checking $Dominated (N_1, N_2)$ is therefore the same with that for
checking the dominance relationship between two variable styles in Assumption 2 of
Section 7.1.1. Let $n$ be the size of the database schema, the computational complexity of
checking $Dominated (N_1, N_2)$ is $O (\log(n))$.

The decision of assigning value to the belief of $Dominated (N_1, N_2)$ and
$Dominated (N_2, N_1)$ is given in Table 7.17 and Table 7.18.

Let us extract all the direct grouping relationships from the table definitions in a
database schema and connect them in a single graph in which they are linked by the
common names they share. Let $N_1$ and $N_2$ be two names in the graph. We say $N_1$
grouped with $N_2$ if there exists a path of direct grouping relationships between $N_1$ and
$N_2$. Formally we denote this kind of relationship as $Grouping (N_1, N_2)$.

<table>
<thead>
<tr>
<th>$N_2$ is dominated by $N_1$</th>
<th>$N_1$ is dominated by $N_2$</th>
<th>$N_1$ is not dominated by $N_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>$N_1$ is dominated by $N_2$</td>
<td>$N_1$ is not dominated by $N_2$</td>
</tr>
<tr>
<td>$N_2$ is not dominated by $N_1$</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.17: Belief of Dominated ($N_1, N_2$) in a Directed Acyclic Graph
For grouping relationship we have Grouping \((N_1, N_2)\) or Grouping \((N_2, N_1)\) and therefore no confliction situation will occur between \(N_1\) and \(N_2\). The process of checking Grouping \((N_1, N_2)\) can be described as: locate \(N_1\); traverse all the paths in the graph that \(N_1\) can reach; if \(N_2\) is found in a path then Grouping \((N_1, N_2) = 1\) and Grouping \((N_2, N_1) = 1\), else \((N_1, N_2) = 0\) and Grouping \((N_2, N_1) = 0\).

Let \(n\) be the size of a database schema, the computational complexity of checking Grouping \((N_2, N_1)\) is \(O(n)\).

Besides the two kinds of relationships mentioned above, we also preserve the Table \((N_1, N_2)\) as the raw relationship. Here, Table \((N_1, N_2)\) denotes the fact that \(N_1\) and \(N_2\) are in the same table.

The general process of generating raw relationship between recovered concepts in a database schema is similar to that for a C program module mentioned in Section 7.1.4.

Let \(N_1, N_2, \ldots, N_m, N_{m+1}, \ldots, N_{m+n}\) be \(m + n\) names recovered from a database schema; \(S, T_2, \ldots, T_m, F_{m+1}, \ldots, F_{m+n}\) be 1 schema name, \(m - 1\) table names and \(n\) field names. Table 7.18 is therefore generated and needs to be filled in.

Let \(n\) be the size of a database schema, the size of Table 7.18 is of the order \(O(n)\); the computational complexity of generating raw relationships for recovered business concepts from a database schema is therefore \(O(n^2)\).

### 7.5 Summary

In this chapter the issue of how to recover business intelligence slices from a source program has been addressed. Uncertainty issues were also addressed to deal with the ambiguity in recovering the concepts, the relationships and the business intelligence slices. Two kinds of source programs are of the particular interests, they are C program and Database Schema. The processes of recovering business intelligence slices from...
these two kinds of source code were discussed separately. In general cases, the computational complexity of this business intelligence recovery algorithm is $O(n^2)$ where $n$ is the average size of a program module.
Chapter 8

Business Intelligence Modelling via Formal Concept Analysis

The use of formal concept analysis (FCA) in software engineering is not new. An overview of FCA application for several purposes of software maintenance, reverse engineering, re-engineering and evolution can find in [122]. The reason why FCA is chosen to model the recovered business intelligence in this thesis, as already been mentioned in Chapter 3, is that FCA is more flexible and evolvable from a conceptual level in business oriented program comprehension. To help readers get a understanding of modelling business intelligence via FCA, this chapter is organised as follows: Section 8.1 makes a short introduction about FCA from definition and implication; Section 8.2, Section 8.3, and Section 8.4 discuss the business intelligence modelling process via FCA which contains the following steps: filtering business intelligence concepts, generating tabular concept context, and building concept lattice. Section 8.5 is the summary.

8.1 Introduction

Formal Concept Analysis is a method for data analysis [40], knowledge representation and information management. It is often applied as an application-level technique for program comprehension. The aim of FCA is to explore conceptual patterns in empirical data contexts. Generally, there is no immediately automatic way to derive the conceptual structures of data contexts which are based on object-attribute-value relationships. However, the main goal of research is to assess whether FCA could be used to analyse the recovered business intelligence and to discover meaningful business intelligence modelling views automatically. First of all, the mathematical expressions and definitions in FCA are introduced.
8.1.1 Definition

Formal Concept Analysis was firstly introduced by Wille [130] and is completely developed in [60, 61]. It is a mathematical technique that allows the underlying abstractions in a data table, formally a context, to be shown by means of the construction of the concept lattice [41]. The FCA has been used in works related to symbolic data analysis and knowledge representation [64]. It has also been used in themes related to software engineering, both for the study of inheritance [65], and the study of the ways in which the class characteristics in a hierarchy are used [65, 118]. Both approaches have been combined to represent the construction process of domain frameworks. It has also been used to module identification within monolithic code. It has also been used in the area of databases [119]. Discussion will be dealt with in more details in Section 8.5.

**Definition**: Let us call a tuple of sets, \((G, M, I)\), that verify \(I \subseteq (G \times M)\), formal context. G is usually called a set of objects and M a set of attributes. The binary relation \(I\) gives the instance of the set of attributes on the set of objects. It is then possible to define the following applications, in whose definitions the notions, respectively, of the set of attributes that certain objects possess, and the set of objects that certain attributes possess, can be seen:

\[
\varphi: \wp(M) \rightarrow \wp(G)
\]

\[
A \mapsto A^\dagger = \{m \in A | (g, m) \in I, \forall g \in G\}
\]

\[
\psi: \wp(G) \rightarrow \wp(M)
\]

\[
B \mapsto B^\dagger = \{g \in B | (g, m) \in I, \forall m \in M\}
\]

These two definitions above allow the following definition to be made, that reflects the informal notion of concept as a set of objects and attributes that are mutually determined.

**Definition**: Let us call a pair \((A, B)\) that verifies \(A^\dagger = B \) and \(B^\dagger = A\), a formal concept. Normally, the first set in the pair will be called the concept extent and the second, the concept intent. The set of formal concepts associated to a context \((G, M, I)\) will be denoted as \(G(G, M, I)\).
On $G(G,M,I)$ a partial order relation can be defined through the following formula where $(A,B),(A',B') \in G(G,M,I)$:

$$(A, B) \leq (A', B') \iff A \subseteq A' \land B \supseteq B'$$

From this definition, the following result can now be proved:

Theorem (Fundamental for concept lattice): The set $G(G,M,I)$ with the defined partial order relation forms a complete lattice in which the lowest and highest are given by the following formulas where $T$ denotes a set of indices, not necessarily finite, and $\forall t \in T, (A_t, B_t) \in G(G,M,I)$:

$$\bigwedge_{t \in T} (A_t, B_t) = \left( \bigcap_{t \in T} A_t , \left( \bigcup_{t \in T} B_t \right)^\perp \right)$$

$$\bigvee_{t \in T} (A_t, B_t) = \left( \left( \bigcup_{t \in T} A_t \right)^\perp , \bigcap_{t \in T} B_t \right)$$

The existence of the lowest and highest for any set of concepts allows the following functions to be defined:

$$\gamma: G \to G(G,M,I)$$

$$g \mapsto \bigwedge_{(A,B) \in (G,M,I) \mid g \in A} (A, B)$$

$$\mu: M \to G(G,M,I)$$

$$m \mapsto \bigvee_{(A,B) \in (G,M,I) \mid m \in B} (A, B)$$

It is easy to show that these functions admit a much simpler notation, as follows:
\[ \forall g \in G, \gamma(g) = (g^{11}, g^1) \]
\[ \forall m \in M, \mu(m) = (m^\dagger, m^{1\dagger}) \]

This provides a practical way of determining the largest concept in whose extent a certain object appears, or which other objects share all the attributes of a given object. Each node in such a diagram represents a formal concept and each arc indicates an order relation between two concepts, where the larger is placed above the smaller, with the restriction that no intermediate concept exists.

Labelling the nodes of the lattice with the complete description of the associate concept is difficult to read, so objects are normally labelled with the lowest concept in whose extent they appear, \( \gamma(g) \), while the attributes do so with \( \mu(m) \), the highest concept in whose intent they appear.

It should be pointed out that the original lattice and the context itself can be reconstructed from this representation. The sets of objects and attributes are directly obtained from the sets of labels, while the incidence matrix is obtained from the expression \((g, m) \in I \iff \gamma(g) \leq \mu(m)\).

Here is a small example of context table and concept lattice figure about different mobiles and their features on a web-based mobile retailing system.

<table>
<thead>
<tr>
<th></th>
<th>1: Email</th>
<th>2: 3G</th>
<th>3: GPS</th>
<th>4: Touch Screen</th>
<th>5: Wi-Fi</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: BlackBerry 9000</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>B: Apple iPhone 3G</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>C: Nokia N96</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>D: Sony Ericsson X1</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>E: T Mobile G1</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

Table 8.1: A Small Example of Mobile Context Table
8.1.2 Implication

Information Retrieval

There was some interest in the use of lattices for information retrieval by [64, 113] and others with respect to document/term lattices and lattices of Boolean query combinations [106] for a summary of these early attempts.

Carpineto & Romano’s [39] research was initially influenced by Godin’s work but has since then been independently advanced to a high level. Their Credo engine [40] facilitates a lattice-based metasearch of Google results. An overview of their work and FCA applications in information retrieval in general can be found in Carpineto & Romano [41].

Apart from Credo, a second FCA application that has reached professional quality is the Mail-Sleuth software [54]. This software is marketed by an Australian company and consists of a plug-in for the MS Outlook email software, which can be used to mine large email archives. The development of this software is based on earlier research on retrieval of information from semi-structured texts [47, 48].

In general, FCA software appears to show a promise for applications in information retrieval, however, with a few limitations. Similar to latent semantic analysis, LSA [54], FCA is not suited for direct manipulation of very large data sources. FCA technology...
claims to be human-centred due to its philosophical basis but only few practical usability studies exist [54]. It is to be hoped that at least Credo will be extensively tested for usability.

**Knowledge Representation and Knowledge Discovery**

Formal Concept Analysis provides a contrast to some of the traditional, statistical means of data analysis and knowledge representation because of its focus on human-centred approaches. In his first paper on FCA, Wille (1982) explains that he was influenced by concerns about the status of sciences in the modern world. FCA was started as an attempt at restructuring mathematical lattice theory [132] in a manner that both facilitates communication about mathematical theory to a wider non-mathematical audience and facilitates exploitation of mathematical theory for a wide range of applications. The concept lattices of FCA serve as a means for communication, exploration and discussion which compiles both with Habermas’s Theory of Communicative Action and Peirce’s pragmatism [130].

The use of diagrams for reasoning has been formally investigated by Dau [50]. He observes that mathematicians often include diagrams in their descriptions of mathematical facts but that normally such diagrams are not permissible as arguments themselves. By formally distinguishing between a mathematical structure and its diagrammatic representation, Dau provides a framework in which diagrams can be used for formal reasoning.

Thus in addition to an intuitive notion of the importance of visualisations, such as concept lattices, Dau can even formally evaluate their usefulness within a formal framework itself. Conceptual Knowledge Discovery [69] and [119]. In contrast to statistical software, which attempts to provide probable answers to narrow questions, Toscana systems facilitate browsing and interactive exploration of implicit and explicit structures. Because the preparation of data for input into a Toscana system is labor-intensive and requires substantial knowledge of FCA, Toscana systems are usually compiled by an FCA expert in co-operation with a domain expert. [130] argues that this is an advantage because the processes involved in creating a conceptual representation (in the sense of FCA) encourages the discovery of implicit information and facilitates the conversion of information into knowledge. Nevertheless, the effort required for setting up Toscana systems may be a reason why their use is not more wide-spread. It
should be emphasized, however, that only the preparation of a Toscana system requires expertise. End-users can utilize such a system after reading a brief introduction.

Toscana system can be less error prone than some statistical methods because a careful conceptual modelling prevents data misrepresentation. We believe that there are further research opportunities for FCA in the area of knowledge discovery which have not yet been exploited. Two workshops have been held so far on the topic of “concept lattices and knowledge discovery in databases”. But many of the papers at these workshops focused on algorithmic issues (such as Kuznetsov & Obiedkov [87, 88]) or abstract issues (such as Wille [130]). These issues are important but it would be more interesting to see more realistic applications.

### 8.2 Filtering Formal Business Concept

Since the goal in this thesis is to mine business intelligence and business intelligence concepts from source code, the FCA elements are mined from source-code entities like variable names, parameter names, and methods, see Table 8.2. The reason why additional entities like bundles, protocols and categories, are not considered is that, firstly, in order to avoid cluttering the results, it is chosen to be pragmatic and include initially only the most relevant source code entities; secondly, the approach is chosen to be language independent, and most other programming languages do not feature protocols, categories, etc.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Names</td>
<td>The symbol stands for a value that may vary.</td>
</tr>
<tr>
<td></td>
<td>• Integer</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td>Parameter Names</td>
<td>The parameter is a variable that takes on the meaning of a corresponding argument passed in a call to a subroutine.</td>
</tr>
</tbody>
</table>
• Call-by-value: a parameter acts within the subroutine as a local (isolated) copy of the argument
• Call-by-reference: the argument supplied by the caller can be affected by actions within the called subroutine

### Functional Methods

A subroutine or subprogram (also called procedure, function, method, or routine) is a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code.

- Objects
- Object classes

<table>
<thead>
<tr>
<th>Table 8.2: Filtering Source Code Entities</th>
</tr>
</thead>
</table>

It is as properties that the method takes simple substrings of the names of the chosen source-code entities. Evidently, an application’s source code contains a wealth of other information, such as call-graph or parse-tree information. Computing such information and reasoning with it, requires vastly more resources, however, which would make our approach much less lightweight [128]. For this very reason, it is chosen to resort to a simple “name matching” approach in Chapter 7. Preliminary experiments that reason about similarities in parse trees are ongoing at the moment, but should still be considered future work.

Because programmers often rely on naming conventions to reveal their intentions and to implement certain programming idioms and design patterns, keeping the properties with as much additional information as they can will make the generation and manipulation efficiently.

Nevertheless, in order to limit the number of properties, it does not consider all possible substrings. Instead, variable names, parameter names and method are split in substrings according to the capitals and other separators occurring in them. In addition, discarding...
substrings are with little conceptual meaning or they are used too often, such as ‘with’, ‘from’, ‘the’, ‘object’, as well as substrings that are too small (i.e., less than 3 characters), we also ignore colons, plurals and the difference in case when comparing substrings. For example, the properties associated with a class MobilePlanDataUsage are the substrings ‘Mobile’, ‘Plan’ and ‘Data’. The properties corresponding to a method named #unifyWithOverrolledMinutes:inEnv:myFreeMinutes:Index:inSource: are ‘unify’, ‘over roll’, ‘minutes’, ‘free minutes’, ‘index’ and ‘source’. See Script 8.3.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Splitting</td>
<td>MobilePlanDataUsage:: ‘Mobile’, ‘Plan’ and ‘Data Usage’</td>
</tr>
<tr>
<td>Discarding</td>
<td>With, In, …, My</td>
</tr>
</tbody>
</table>

Table 8.3: Functions of Filtering Business Intelligence Concepts

As it is seen in Table 8.3, for the considered cases the number of concepts discovered by FCA, before applying any filtering, is of the same order of magnitude as the number of considered elements. This would imply that a software engineer needs to look at a significant number of concepts in order to understand the source code. However, the concepts do contain a large amount of redundancy and noise that an engineer can easily filter.

A first filter ignores all concepts that contain two or less elements, since these concepts are generally too small to provide relevant information. Note that this filter discards most accessing method concepts, since these typically contain only two elements: an accessor and a mutator method. However, since accessing methods are rather fine-grained, since there are a lot of them, and since they can be inspected with standard rules easily.

A second filter ignores all concepts that share only one property (substring). Although this filter may discard some interesting concepts, it does throw away many more irrelevant concepts. It is considered that, during initial understanding of an
application, getting a quick and focussed idea of certain commonalities in the source-code is more important than getting a precise list of all possible commonalities. A nice improvement of this filter would be to discard those concepts of which the properties ‘cover’ only a small fraction based on some threshold, of the concept elements’ names. As such, the filter becomes relative to the size of the elements’ names.

**Whereas these two generic filters are independent of the kinds of elements being analysed, the third filter is more targeted.** It discards concepts that contain only classes (with a similar name) in the same hierarchy. These concepts typically do not provide very useful information – since classes belonging to the same hierarchy often have similar names – except if we want to discover exactly which naming convention these classes are relying upon.

### 8.3 Generating Tabular Concept Context

Applying a concept context generation to the elements and properties filtered in the previous step involves several hundreds to thousands of concepts, depending on the size of the application. The results in a large concept lattice will be further built in the next step in Section 8.4.

Let it be sufficient to give an illustrative example of a simple kind of concepts that is discovered by the FCA algorithm. Consider this example: \( O = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \), and \( A = \{\text{composite, even, odd, prime, square}\} \). The smallest concept including the number 3 is the one with objects \( \{3, 5, 7\} \), and attributes \{odd, prime\}, for 3 has both of those attributes and \( \{3, 5, 7\} \) is the set of object s having that set of attributes. The largest concept involving the attribute of being square is the one with objects \( \{1, 4, 9\} \) and attributes \{square\}, for 1, 4 and 9 are all the square numbers and all three of them have that set of attributes. It can readily be seen that both of these example concepts satisfy the formal definitions below.

The full set of concepts for these objects and attributes is shown in the illustration. It includes a concept for each of the original attributes: the composite numbers, square numbers, even numbers, odd numbers, and prime numbers. Additionally it includes concepts for the even composite numbers, composite square numbers (that is, all square numbers except 1), even composite squares, odd squares, odd composite squares, even primes, and odd primes.
A context may be described as a table, with the objects corresponding to the rows of the table, the attributes corresponding to the columns of the table, and a Boolean value (represented graphically below as a checkmark) in cell \((x, y)\) whenever object \(x\) has value \(y\):

<table>
<thead>
<tr>
<th></th>
<th>Composite</th>
<th>Even</th>
<th>Odd</th>
<th>Prime</th>
<th>Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>V</td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>V</td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.4: An Example of Generated the Tabular Concept Context

A concept, in this representation, forms a maximal subarray (not necessarily contiguous) such that all cells within the subarray are checked. For instance, the concept highlighted with shadow in the table above is the one describing the odd prime numbers, and forms a \(4 \times 2\) subarray in which all cells are checked.

### 8.4 Building Concept Lattice

It is taken as givens that a (formal) context consisting of a set of objects \(O\), a set of attributes \(A\), and an indication of which objects have which attributes.

A concept is defined to be a pair \((O_i, A_i)\) such that:
\[ O_i \subseteq O \]
\[ A_i \subseteq A \]
- every object in \( O_i \) has every attribute in \( A_i \)
- for every object in \( O \) that is not in \( O_i \), there is an attribute in \( A_i \) that the object does not have
- for every attribute in \( A \) that is not in \( A_i \), there is an object in \( O_i \) that does not have that attribute

\( O_i \) is called the extent of the concept, \( A_i \) the intent.

The concepts \((O_i, A_i)\) defined above can be partially ordered by inclusion: if \((O_i, A_i)\) and \((O_j, A_j)\) are concepts, we define a partial order \( \leq \) by saying that \((O_i, A_i) \leq (O_j, A_j)\) whenever \( O_i \subseteq O_j \). Equivalently, \((O_i, A_i) \leq (O_j, A_j)\) whenever \( A_j \subseteq A_i \).

Every pair of concepts in this partial order has a unique greatest lower bound (meet). The greatest lower bound of \((O_i, A_i)\) and \((O_j, A_j)\) is the concept with objects \( O_i \cap O_j \); it has as its attributes the union of \( A_i \), \( A_j \), and any additional attributes held by all objects in \( O_i \cap O_j \).

Symmetrically, every pair of concepts in this partial order has a unique least upper bound (join). The least upper bound of \((O_i, A_i)\) and \((O_j, A_j)\) is the concept with attributes \( A_i \cap A_j \); it has as its objects the union of \( O_i \), \( O_j \), and any additional objects that have all attributes in \( A_i \cap A_j \).

These meet and join operations satisfy the axioms defining a lattice. Conversely, any finite lattice may be generated as the concept lattice for some context. For, let \( L \) be a finite lattice, and form a context in which the objects and the attributes both correspond to elements of \( L \). In this context, let object \( x \) have attribute \( y \) exactly when \( x \) and \( y \) are ordered as \( x \leq y \) in the lattice. Then, the concept lattice of this context is isomorphic to \( L \) itself. See Figure 8.2.
8.5 Summary

The implementation of FCA in modelling recovered business intelligence is new, but as a program comprehension technique, FCA is not new to software engineering. The kernel research idea of this chapter is to explore a path of modelling the recovered business intelligence on base of the former steps and recovery results to have construction views about business intelligence embedded in the software system.

The method is not about to fulfil the needs of producing a business intelligence report in from the business intelligence base, in fact, reporting tasks in many cases are not part of software engineering work. It is about analysing the business intelligence figures and producing reports with dashboard reports.

The view of FCA enabled business intelligence modelling is to maximise the automation level of modelling and analysing the recovered business intelligence, thereafter the whole process drives as much work as in manual into automated digitalised computations.
Chapter 9

Case Study

Having introduced both the theoretical and technical aspects of the proposed methods of business intelligence recovery in the previous chapters, we move on, in this chapter, to describe how these methods were implemented in a business intelligence recovery task. A case study for evaluating different aspects of these methods is given.

The sections in this chapter are organised as follows. In Section 9.1, it describes some key implementation issues of the proposed approach. A case study of an online mobile retailing system is then given in Section 9.2 to evaluate the methods of recovering business intelligence from program code, the linear program partitioning method, the concept recovery method, and the formal concept analysis method. A Web toolkit is developed to evaluate the proposed Auto-BIR techniques in Section 9.3. Finally a summary of business intelligence recovery results is given in Section 9.4.

9.1. Key Issues of Automating Business Intelligence Recovery (BIR) Implementation

The practice of business intelligence recovery techniques involves four issues in the thesis:

- A business intelligence decomposition method is developed to partition monolithic business intelligence into manageable unit named business intelligence slices. The practice of matching business intelligence slices and program code guarantees the success of business intelligence recovery.

- A program partition technique was developed to obtain business intelligence oriented program modules from source code. The practice of partitioning program code into program modules is based on different programming styles.
- A concept recovery technique was developed to obtain business concepts and their relationships embedded in the source code. The practice is built on the of name interpretation.

- A formal concept analysis technique was developed to modelling business intelligence concept and relationship within partitioned program modules. The practice is built on the recovery results and the former BIR procedures.

9.2 Case Study: An Online Mobile Retailing System

In this section, a software evolution task of “re-engineering an existing Web-based mobile retailing system into an online mobile retailing system, which is enabled with online business intelligence accumulation from the existing system and the Web” is picked as case study. It is a 150,000-line of C and C# code system for a mobile retailing company to manage its daily mobile retailing business. The original Web-based system is mainly composed of three parts: Website, running system and database. The IT strategy department received the task from business administration department that requires: (1) the online mobile retailing system could automatically perform Web business intelligence accumulation from the Websites of major mobile operators in the UK, such as Vodafone, O2, Orange, T-Mobile and Hutchison 3G; (2) the online mobile retailing system could automatically perform business intelligence recovery from the existing Web-based mobile retailing system. Since the accumulation of Web business intelligence are not core research point in this thesis about performing reverse engineering on software systems, it is not going to be further discussed. All the case study is presented to demonstrate the automatic business intelligence recovery approaches which have been discussed in previous chapters.

The whole Web-based retail system includes a mobile retailing Website, a retailing database, and a running system of retailing service. The target business intelligence covers content such as mobile registration, price calculation, billing, payroll and accounting, and point of sell, etc. In practice, business intelligence of the mobile retail system is more than those been mentioned. Here in the thesis, part of the entire business intelligence, such as business intelligence value, business intelligence concept, and business intelligence lattice are presented. They are more than enough to present and interpret business intelligence in a software system.
The system code and recovered business intelligence results are attached in Appendix A and Appendix B.

9.2.1 Business Intelligence Partitioning

The purpose of this part in case study is to evaluate the proposed method of partitioning monolithic business intelligence into business intelligence slices. The application domain in this case study is “Credit Checking and Contract Signing”. In particular, business intelligence of an “online credit checking and contract signing” written in C is going to be recovered. Figure 9.1 gives us a whole picture of mobile retailing service. The brief idea is how to partition two pieces of business intelligence slice called “credit checking” and “contract signing” from the program code.

![Figure 9.1: Business Model of An Online Mobile Retailing System](image)

Defining Program Constructs Before Partitioning Business Intelligence

Firstly let us take the mind out from the overall picture and think more about C programming language. The effort will not do, unless programming constructs for C are
well explained. The following programming constructs for C have been identified, see Table 9.1:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1. Variable Type (VT) Definition | Struct VT {
|   |   |   | VT1; …, VTn;
|   |   |} |
| 2. Variable Name (VN) Definition | VT VN; |
| 3. Procedure Definition | VT PN (VT1, …, VTn) {
|   |   | Statements |
|   |} |
| 4. Statements | Statement1;
|   |   | … |
|   |   | Statementn; |
| 5. if Statement | if E then
|   |   | Statement1;
|   |   | Else
|   |   | Statement2;
|   |} |
| 6. while Statement | while E do
|   |   | Statement |
|   |   | endwhile |
| 7. procedure call statement | PN (); |
| 8. program comment | comment content |
| 9. Table Definition | Table TN {
|   |   | FN1, …, FNn |
|   |} |
| 10. Database Schema Definition | Schema SN {
|   |   | TN1, …, TNn |
|   |} |

Table 9.1 Programming Constructs for C and Database

Table 9.1 presents a sufficient set of C programming constructs and database programming constructs before the first step of performing business intelligence recovery from C code. In Table 9.1, statements 1-8 are the programming constructs for C, statements 9-10 are programming constructs for Database Schema. The Note of programming constructs for C and database schema is:
Once the programming constructs have been defined, any input of program code of C and database schema can be parsed. The raw generating of business intelligence slices starts thereafter.

The path to have business intelligence slices from start is composed of four steps: (1) generating raw business intelligence, (2) parsing C and database schema, (3) identify candidate names and recover concepts, (4) generating raw relationships, and (1), again.

**Generating Raw Business Intelligence**

The general function of a `creditchecking.c` file is to establish an individual communicative connection between credit checking and mobile contract assigning, once personal details, such as, living address and debit card information etc. are input into the Web system. This is usually done by the following procedures:

1. *Input bank card number and billing address.* The billing address is used as to check the history of mobile assignment with the address. The bank card number is used to check the history of account balance.

2. *Bind the mobile number with mobile contract.* The mobile number is bound with a mobile contract. It means the mobile service has been activated. The contract information is part of essential business intelligence information set, and it is the key information to mobile operating services.

3. *Record the monthly usage of Minutes, Text and Data.* The major function of this part is to record the usage of provided telecommunication services.
4. Assign agent to deal with bill statement and charging. Once a month rolling service has been finished, the mobile retailing system will assign an agent to deal with bill statement printing, mailing and charging.

The four steps given above describe the necessary business intelligence to establish a bank credit checking and mobile contract assigning. In order to represent this business intelligence, Input, Bind, Record, and Assign are used to represent the actions of business intelligence and Input-address, Bind-contract, Record-data, and Assign-bill to represent the relationship between different business intelligence units.

Among these business intelligence concepts, Before (Input, Bind), Before (Bind, Record) and Before (Record, Assign) are used to describe the sequential order among Input, Bind, Record, and Assign. Source-of (Address, Input), and Destination-of (Input, Bind-contract) are used to describe the function of Input. Source-of (Input, Bind-Contract) and destination-of (Bind-Contract, Record) are used to describe the function of Bind. Source-of (Bind, Record-Data) and Destination-of (Record-Data, Assign) are used to describe the function of Record. Source-of (Record, Assign) and Destination-of (Assign-Mail) are used to describe the function of Assign.

The graphical version of this business intelligence is shown in Figure 9.2. In this figure, the concepts and relationships enclosed by concepts and a line are composed into a single business intelligence slice. Hence, there are 14 business intelligence slices in Figure 9.2.
Figure 9.2 Graphical Version of Business Intelligence for Mobile Retailing

The detailed business intelligence base is shown in Script 9.1.

<table>
<thead>
<tr>
<th>Business Intelligence Slice</th>
<th>Operation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(Input 0.1)</td>
</tr>
<tr>
<td>2</td>
<td>(Assign 0.15)</td>
</tr>
<tr>
<td>3</td>
<td>(Assign 0.15)</td>
</tr>
<tr>
<td>4</td>
<td>(Assignment)</td>
</tr>
<tr>
<td>5</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>6</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>7</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>8</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>9</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>10</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>11</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>12</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>13</td>
<td>(Record 0.9)</td>
</tr>
<tr>
<td>14</td>
<td>(Record 0.9)</td>
</tr>
</tbody>
</table>

The detailed business intelligence base is shown in Script 9.1.
Script 9.1: Business Intelligence Base of An Online Mobile Retailing System

It is noted that there is a real number after each concept or relationship. The real number denotes the strength of that concept or relationship in corresponding business intelligence slice.

Parsing C and Database Schema

A C source program generally consists of several header files and body files. A header file contains the definitions of variable types, the header file definition of a procedure, etc. the name a header file ends with .h. A body file contains the detailed implementation of the procedures whose header definition occurs in its header file. The name of the body file ends with .c.

Script 9.2 shows the header file for credit checking and contract assigning program.

```c
/*********************************************************
********** bankcreditmobilecontract.h **********
**********************************************************/

/** variable type definition **/

struct input {
    int input_address;
};

struct host {
    int acct;
    addr addr;
};

struct call {
    int call_credit;
};
```
struct addr {
    int house_no;
    char* buf;
};

struct check {
    int check_credit;
};

/** procedure definitions **/

input* t_input(char*, int, int);
host* t_alloc_bind (input*, int, int);
call* t_alloc_call (input*, int, int);
int t_bind (input*, call*);
int t_input (input*, call*);
bind* t_accept_call (input*, call*);
void error (char*);
void exit ();
void fprint ();
void run_service(bind);

---

Script 9.2: bankcreditmobilecontract.h

The body file of bank credit checking and mobile contract assigning is given in Script 9.3.

/********************
****** bankcredit.mobilecontract.c ******
********************/

#include <tiuser.h>
#include <stropt.h>
#include <fcn1.h>
#include <stdio.h>
#include <bind-mobile_number-contract>

#define DISCONNECT -1
#define SRV_ADDR 1 /* well know address*/

service ()
{
    struct input *inp; /* inputing bank account number */
    struct host *host;
    struct call *call;
    struct bind *bind; /* bind established here*/

    if (((lis = t_input("/dev/tivc", O_RDWR, NULL))<0);
        error("open failed for inp");
        exit (1);
    });

    if (((host = t_alloc(inp, T_BIND, T_ALL)) == NULL) {
        error("alloc of host structure failed");
        exit (2);
    );
    host -> qlen = 1;
    host -> addr.len = sizeof (int);
    *(int *)host -> addr.buf = SRV_ADDR;

    if (t_bind (inp, host, host) < 0) {
        error("bind failed for inp");
        exit (3);
    });

    /*
    * was the correct address bound?
    */
    if (*(int *)host->addr.buf !=SRV_ADDR) {

Identify Candidate Names and Recover Concepts

In Script 9.2 and Script 9.3, the names embedded in *struct* definitions and *procedure* definitions are eligible as candidate names from which business intelligence concepts will be recovered. The names embedded in *struct* definitions and *procedure* definitions are listed in Script 9.4 respectively.

-----------------------------
in
put (01), host (02), call (03)
add
r (04), bind (05)

-----------------------------
t_input (A1), t_alloc_bind (A2), t_alloc_call (A3), t_bind (A4), t_input (A5),
t_accept_call (A6), error (A7), exit (A8), fprint (A9), run_service (A10), service (A11)

Script 9.4: Candidate Business Intelligence Concepts

It is noted that $O_i, 1 \leq i \leq 5$ or $A_j, 1 \leq j \leq 11$ in Script 9.4 denote the positions of the names in the C program.

By applying atomic and compound name recovery rules to those candidate names listed in Script 9.4, Script 9.5, the following concepts are recovered:

<table>
<thead>
<tr>
<th>Original Names (Position)</th>
<th>Recovered Concepts</th>
<th>Belief</th>
<th>Types of the Rules Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>input (01)</td>
<td>Input-address (01)</td>
<td>0.5</td>
<td>Regular</td>
</tr>
<tr>
<td>host (02)</td>
<td>Host (02)</td>
<td>1.0</td>
<td>Regular</td>
</tr>
<tr>
<td>call (03)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>addr (04)</td>
<td>Address (04)</td>
<td>0.4</td>
<td>Regular</td>
</tr>
<tr>
<td>bind (05)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>t_ingp (A1)</td>
<td>Input(A1)</td>
<td>0.7</td>
<td>Direct Match</td>
</tr>
<tr>
<td>t_bind (A2)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>t_alloc_call (A3)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>t_bind (A4)</td>
<td>Bind (A4)</td>
<td>0.7</td>
<td>Direct Match</td>
</tr>
<tr>
<td>t_input (A5)</td>
<td>Input (A5)</td>
<td>0.7</td>
<td>Direct Match</td>
</tr>
<tr>
<td>t_accept_call (A6)</td>
<td>Accept (A6)</td>
<td>0.7</td>
<td>Regular</td>
</tr>
<tr>
<td>error (A7)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>exit (A8)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>fprint (A9)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>run_service (A10)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
<tr>
<td>service (A11)</td>
<td>---</td>
<td>--</td>
<td>Fail</td>
</tr>
</tbody>
</table>

Script 9.5: Recovered Business Intelligence Concepts

From Script 9.5 it can be seen that those concepts whose value of belief is positive are recovered. In this case, concepts input-address (O1, 0.5), host (O2, 1.0), address (O4,
0.4), inp (A1, 0.7), bind (A4, 0.7), input (A5, 0.7) and accept (A6, 0.7) are recovered. The interpretation of input-address (O1, 0.5) is: a concept input-address is recovered from location O1 in the C code with a value of belief 0.5.

**Generating Raw Relationships**

The raw relationships among the recovered concepts are then generated. Script 9.6 shows the raw relationships between object and object, action and action, and object and action.

<table>
<thead>
<tr>
<th></th>
<th>input (01)</th>
<th>host (02)</th>
<th>addr(04)</th>
</tr>
</thead>
<tbody>
<tr>
<td>input(01)</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>host(02)</td>
<td>0</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>addr(04)</td>
<td>0</td>
<td>1.0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Dominated (Oi,Oj)**

<table>
<thead>
<tr>
<th></th>
<th>t_inp(A1)</th>
<th>t_bind(A4)</th>
<th>t_input(A5)</th>
<th>t_accept_call(A6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_inp(A1)</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t_bind(A4)</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t_input(A5)</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>t_accept_call(A6)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Dominate (Ai,Aj)**

<table>
<thead>
<tr>
<th></th>
<th>t_inp(A1)</th>
<th>t_bind(A4)</th>
<th>t_input(A5)</th>
<th>t_accept_call(A6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_inp(A1)</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>t_bind(A4)</td>
<td>-1.0</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>t_input(A5)</td>
<td>-1.0</td>
<td>-1.0</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>t_accept_call(A6)</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Before (Ai,Aj)**
Section 9.6: Generating Business Intelligence Concepts and Relationship

The recovered concepts and raw relationships are then matched with the business intelligence base to generate candidate business intelligence slices, as shown in Script 9.7.

Business Intelligence Slice 1 (WBIS 1):

Input(All, 0.7), Bind(A41, 0.7), Record(A51, 0.7), Assign(A61, 0.7),
Before(A11, A41, 1.0), Before (A41, A51, 1.0), Before (A51, A61, 1.0),
(A11, 0.10, 0.10), (A41, 0.15, 0.15), (A51, 0.5, 0.15), (A61, 0.15, 0.15),
(Before(A11, A41), 0.10, 0.40), (Before(A41, A51), 0.15, 0.15), (Before(A51, A61), 0.60, 0.50)

**Business Intelligence Slice 3 (WBIS 2):**

----------------------------------
Input(A12, 0.7), Device-Name(?2, 0), Input-Address(O12, 0.5),
Source-of(?2, A12, 0), Destination-of(O12, A12, 1.0),
(A12, 0.10, 0.10), (?2, 0.10, 0.10), (O12, 0.10, 0.10),
(Source-of(?2, A12), 0.20, 0.20), (Destination-of(A12, O12), 0.50, 0.50)

**Business Intelligence Slice 3 (WBIS 3):**

----------------------------------
Bind(A43, 0.7), Input-Address(O13, 0.5), Host(O23, 1.0),
Source-of(O13, A43, 1.0), Source-of(O23, A43, 1.0),
(A43, 0.10, 0.10), (O13, 0.10, 0.10), (O23, 0.10, 0.10),
(Source-of(O13, A43), 0.4, 0.4), (Source-of(O23, A43), 0.4, 0.4)

**Business Intelligence Slice 4 (WBIS 4):**

----------------------------------
Host(O24, 1.0), Address(O44, 0.4)
Part-of(O44, O24, 1.0)
(O24, 0.10, 0.10), (O44, 0.10, 0.10),
(Part-of(O44, O24), 0.9, 0.9)

**Business Intelligence Slice 5 (WBIS 5):**

----------------------------------
Input-Address(O15, 0.5), Input(A55, 0.7),
Source-of(O15, A55, 1.0),
(O15, 0.10, 0.10), (A55, 0.10, 0.10),
(Source-of(O15, A55), 0.9, 0.9)

**Business Intelligence Slice 6 (WBIS 6):**

----------------------------------
Input-Address(O16, 0.5), Bind(A66, 0.7),
Source-of(O16, A66, 1.0),
(O16, 0.10, 0.10), (A66, 0.10, 0.10)
It is noted that in each working business intelligence slice the following information is listed: recovered concepts, recovered relationships, and concept/relationship strength. \( \text{Input}(A11, 0.7) \) denotes that the belief of the concept Input recovered at \( A11 \) is 0.7. \( \text{Before}(A11, A41, 1.0) \) denotes that the business intelligence relationship Before between the concept at \( A11 \) and concept at \( A41 \) is 1.0. \( (A11, 0.10, 0.10) \) denotes that the concept and relationship strength of \( A11 \) in \( WBIS1 \) are 0.10 and 0.10 respectively. \( (\text{Before}(A11, A41), 0.10, 0.40) \) denotes that the concept and relationship strength of \( \text{Before}(A11, A41) \) in \( WBIS1 \) are 0.10 and 0.40 respectively.

As mentioned previously, the business intelligence partitioning process needs to roll around with the first step in the end. Therefore, at least, there are three stages of business intelligence partitioning result: initial stage, 1st round and 2nd round, as shown in Script 9.8.

<table>
<thead>
<tr>
<th></th>
<th>Initial Stage</th>
<th>1st Round</th>
<th>2nd Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concepts</td>
<td>CF0</td>
<td>D(CF1)</td>
<td>CF1</td>
</tr>
<tr>
<td>Input(A1)</td>
<td>0.70</td>
<td>0.59</td>
<td>0.88</td>
</tr>
<tr>
<td>Bind(A4)</td>
<td>0.70</td>
<td>0.63</td>
<td>0.89</td>
</tr>
<tr>
<td>Record(A5)</td>
<td>0.70</td>
<td>0.69</td>
<td>0.91</td>
</tr>
<tr>
<td>Assign(A6)</td>
<td>0.70</td>
<td>0.69</td>
<td>0.91</td>
</tr>
<tr>
<td>Device-Name (?)</td>
<td>0.00</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>Input-Address (01)</td>
<td>0.50</td>
<td>0.86</td>
<td>0.93</td>
</tr>
<tr>
<td>Host (02)</td>
<td>1.00</td>
<td>0.65</td>
<td>1.00</td>
</tr>
<tr>
<td>Address (04)</td>
<td>0.40</td>
<td>0.46</td>
<td>0.67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CF0</th>
<th>CF1</th>
<th>CF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationships</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Before(A11, A41)  1.00  0.43  1.00  0.02  1.00
Before(A41, A51)  1.00  0.43  1.00  0.02  1.00
Before(A51, A61)  1.00  0.43  1.00  0.02  1.00
Source-of(?, A12)  0.00  0.28  0.06  0.03  0.10
Destination-of(O12, A12)  1.00  0.28  1.00  0.01  1.00
Source-of(O13, A43)  1.00  0.36  1.00  0.00  1.00
Source-of(O23, A43)  1.00  0.36  1.00  0.00  1.00
Part-of(O44, O24)  1.00  0.46  1.00  0.00  1.00
Source-of(O15, A55)  1.00  0.46  1.00  0.00  1.00
Source-of(O16, A66)  1.00  0.46  1.00  0.00  1.00

AU0  AU1  AU2
Knowledge Slices  -----------  -----------  1.00
WBIS 1  0.00  0.85  0.90
WBIS 2  0.00  0.56  0.62
WBIS 3  0.00  0.71  0.73
WBIS 4  0.00  0.91  0.92
WBIS 5  0.00  0.91  0.92
WBIS 6  0.00  0.91  0.92

**Script 9.8: Recovered Business Intelligence Slices**

At initial stage, $CF_0$ contains the belief assigned by concept recovery rules and $AU_0$ is assigned to be 0. At the 1st round, $AU_1$ is calculated first, which gives an initial evaluation of the authenticity of these business intelligence slices. $D(CF_1)$ is then calculated for concepts and relationships and used to obtain the revised beliefs $CF_1$ for these concepts/relationships. At the 2nd round, $AU_2$ is, once again, calculated which indicates the enhanced evaluation of the authenticity of the business intelligence slices.

Since all the beliefs of the concepts are non-negative, it can be seen from Script 9.8 that the belief of the concepts and relationships and authenticity of the working candidate business intelligence slices are increasing as the iterative evaluation process goes. At the 1st round, the viewpoints on the authenticity of the business intelligence slices are, for the first time, formed and the value of $D(CF_1)$ at this round are significantly high, which contributes most to the belief revision of the concepts/relationships. At the 2nd round, the neighbouring working candidate business intelligence slices. However,
compared to $D(CF1)$, the value of $D(CF2)$ are much lower because “closely-related” working candidate business intelligence slices have already exchanged their main viewpoints on concepts and “non-neighbouring” working candidate business intelligence slices have less influence on revising the authenticity of each other.

Behaviours of some individual concepts in this belief enhancing process can also be observed. Having a certainty factor of 1.0, concept Host has a tremendous effect on the increment of beliefs of concept Bind and concept Address, which are its “neighbouring” concepts. Getting support from four working candidate business intelligence slices, concept Input-Address makes the most significant increment of its belief among all other concepts. The beliefs of concept Host and all the relationships have not increased since an upper bound of beliefs has been reached. Concept Device-Name, the belief in which is ignorance at initial round, is assigned a positive belief at the 2nd round resulting from the support of its co-operative concepts/relationships. A careful check for this concept in the code is therefore suggested to recover what previously failed to be recovered.

The benefits for utilising co-operative business intelligence in business intelligence recovery process, as shown in this case study, are not only that the reliability of the result is enhanced but also that the disambiguation of the result is enhanced. In this experiment, the upper bound threshold $\beta_{s+}$ for accepting working candidate business intelligence slices is set to be 0.6. At the 1st round, the authenticity (AU1) of business intelligence slices 1, 3, 4, 5 and 6 are greater than $\beta_{s+}$ and are therefore acknowledged. At the 2nd round, the authenticity of business intelligence slice 2 is further acknowledged. In this way, more business intelligence embedded in the source code can be recovered.

Since the working business intelligence slices contain links between business intelligence concepts and the location of variable names or procedural names, the important domain concepts can be easily located in the source code.

9.2.2 Business Intelligence Oriented Program Partitioning

The purpose of this part of the case study is to evaluate the proposed linear program partitioning method. In the following subsections, the following steps of the linear program partitioning method will be described: (1) generating programming style sampling function, (2) filtering programming style sampling function, (3) cutting source
program, (4) re-healing program, (5) improving the granularity of partitioned program modules.

To simplify this part of case study, a C# source file named mobileoperating.cs which is the largest source file in the running system part as our experimental case study. The functionality of this file is to maintain a real-time communication channel between running system and database. Since every command between running system and database will pass through this channel, the mobileoperating.cs indeed records business intelligence made on the online mobile retailing system over many years. Currently, mobileoperating.cs has 92 functions, residing in 1578 lines of C# code. Appendix A gives a complete list of the mobileoperating.cs source code.

**Generating Programming Style Sampling Function**

In accordance with the algorithm described in Chapter 6, three different groups of programming styles, comments, naming and indentation are traced. Appendix A.1 shows the programming style sampling function for comments. SSC1, SSC2, SMC1 and SMC2 are the codes of program styles for single-line comment and multi-line comment explained earlier in Chapter 6. Appendix A.2 shows the programming style sampling function for naming SVN1, SVN2, SVN3, SFN1, SFN2 and SFN3 are the codes of programming styles for variable names. Appendix A.3 shows that programming style function for indentation. SUI1, SUI2, SUI3, SUI4, SUI5, SNI1, SNI2, SNI3, SNI4 and SNI5 are code of programming styles for un-nested indentation and nested indentation. The number shown is Appendix A.1, Appendix A.2 and Appendix A.3 indicates the occurrence of corresponding programming style in each procedure contained in the C# code.

**Filtering Programming Style Sampling Function**

At this step, only the pattern that has the largest number of occurrence among all the patterns in one particular dimension is preserved. The resulting filtered programming style sampling function is given in Appendix A.4. It can be seen that there are six dimensions altogether in the filtered programming style sampling function. The positive number in the table of Appendix A.4 indicates the code of the most significant programming styles in each dimension. 0 indicates that the corresponding program style in a particular function is unknown.
**Cutting Source Program**

Given the filtered programming style sampling function, the C# code is then cut into program sections subsequently. Appendix A.5 illustrates how the cutting process is carried out. The cut program sections are listed in Script 9.9.

<table>
<thead>
<tr>
<th>Function</th>
<th>SSC</th>
<th>SMC</th>
<th>SVN</th>
<th>SFN</th>
<th>SU1</th>
<th>SNI</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>_StartDatabaseOntologyEngine</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>PS1</td>
</tr>
<tr>
<td>_ResetOntologyEngine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloseInterface</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>PS2</td>
</tr>
<tr>
<td>Yipes</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>PS3</td>
</tr>
<tr>
<td>Waiting</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>PS4</td>
</tr>
<tr>
<td>InitialiseInterface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RestartInterface</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>PS5</td>
</tr>
<tr>
<td>SendStringToLisp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetTextFromPipe</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>PS6</td>
</tr>
<tr>
<td>TopLevel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WhatTrans</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>PS7</td>
</tr>
<tr>
<td>BuildAbstractTable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FullMeneue</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>PS8</td>
</tr>
<tr>
<td>AutoMetrics</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>PS9</td>
</tr>
</tbody>
</table>
Script 9.9: Cutting Source Program

Re-healing Program

By re-healing the program sections in Script 9.9, the initial partitioned program modules can be restructured as those in Script 9.10.
Script 9.10: Re-Healing the Cutting Results

Part of the manual analyses of the function behind these program modules are:

PM1: Ontology-related commands and two interface-related functions;
PM2: A function for closing interface normally;
PM3: A function for closing interface in exception;
PM4: Establishing interface;
PM5: Transformation and communication through interface;
PM6: Abstraction;
PM7: A function for GUI;
PM8: Some transformation function;
PM9: Some transformation functions.

The ideal result is that ontology, transformation and interface and abstraction can be distinguished between each other. This initial result shows that ontology and abstraction can be partitioned from the rest of the program, although there is a bit messy in
partitioned ontology module that two interface-related functions are incorporated into. The transformation and interface function, however, scatter around the different partitioned program modules that need to be put together further.

**Improving the Granularity of Partitioned Program Modules**

In order to do so, the distance between each two programming style categories needs to be calculated. To simplify the explanation, only the relevant logical distance between programming style categories is listed. In this case it is the distance between each two program modules that is shown in Script 9.11.

<table>
<thead>
<tr>
<th></th>
<th>PM1</th>
<th>PM2</th>
<th>PM3</th>
<th>PM4</th>
<th>PM5</th>
<th>PM6</th>
<th>PM7</th>
<th>PM8</th>
<th>PM9</th>
<th>PM10</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PM2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PM3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM4</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM9</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM10</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Script 9.10: Improving the Granularity of PPM**

**Scenario 1:** The partitioned program modules, between which the distance is 1, are merged. In this case, the following schema of partitioned program modules is:

**Part 1:** PM1
**Part 2:** PM2
**Part 3:** PM3, PM4, PM8
Part 4: PM5  
Part 5: PM6  
Part 6: PM7  
Part 7: PM9  
Part 8: PM10

It can be seen that only PM3, PM4 and PM8 are further merged into one part.

**Scenario 2:** the partitioned program modules, between which the distance is $\leq 2$, are merged. In this case, the following schema of portioned program modules is:

Part 1: PM1  
Part 2: PM2, PM3, PM4, PM5, PM8, PM9, PM10  
Part 3: PM6  
Part 4: PM7

The above result is much clearer than the previous one shown in Scenario 1. In this case, only function FullMenu is mis-partitioned into an individual part of the source program. The result of partitioning the program into part 1 (ontology), part 2 (system support and transformation) and part 3 (abstraction) is exactly what we need.

### 9.2.3 Formal Concept Based Business Intelligence Modelling

The purpose of this part of the case study is to evaluate the proposed formal concept based business intelligence modelling method. In the following subsections, the following steps of the formal concept analysis method will be described: (1) filtering formal concepts, (2) generating tabular formal concept context, and (3) building concept lattice.

This part of case study is literally built on the results of business intelligence partition, business intelligence concept recovery, and business intelligence oriented program partition, a C# source file named *serviceanalysis.cs* is chosen as the experimental case.
The functionality of this file is to statistically analyse the mobile retailing, and perform the analysis online in mobile retailing system. The overall experiment aim is about to obtain the business intelligence lattice of online mobile retailing at Middle England from September to December 2008. This file maintains a communication channel between User Interface and Database. Since every command between User Interface and Database will pass through this channel, the serviceanalysis.cs indeed records business intelligence made on the online mobile retailing system over many years. Currently, serviceanalysis.cs has 64 functions, residing in 781 lines of C# code. Appendix B gives a complete list of the serviceanalysis.cs source code.

Filtering Formal Concepts

In this section, this is the first step of formal concept analysis that filtering formal concepts from the partitioned program module and the recovered concepts of business intelligence.

In accordance with Chapter 8, three different groups of business intelligence concepts: variable names, parameter names and methods are traced. Appendix B.1 and Appendix B.2 show the variable names, parameter names. SVN1, SVNC2, SVN3, SPN1 and SPN2, SPN3 are the business intelligence concepts explained earlier in Chapter 8. Appendix B.3 shows the functional method naming SMN1, SMN2, and SMN3, are the business intelligence concepts for methods. The number shown is Appendix B.1, and Appendix B.2 indicates the occurrence of corresponding business intelligence concept in each procedure contained in the C# code.

Generating Tabular Concept Context

The tabular concept context is grouped based on the properties such as ‘textRolling’ and ‘minutesRolling’ that are shared by these methods, but by no other methods, classes or parameters in the application. Therefore the tabular concept context would be had like Table 9.2:

<table>
<thead>
<tr>
<th>Minutes</th>
<th>FreeMinutesRolling</th>
<th>Text</th>
<th>FeeTextRolling</th>
<th>Data</th>
<th>FreeDataRolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>V</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9.2: Tabular Context of “PlanRolling”

The tabular context of business intelligence “plan rolling” in the application is a specific module to handle service calculation and control service rolling in accordance with the mobile contract and the status of service consuming. Table 9.2 could generate a piece of business intelligence. It looks like we have nearly had those business intelligence concepts and data; but still, it is very abstract for us to understand. As it has been mentioned, software engineers are lacking of business oriented program comprehensibility when face a large amount of data. A better idea will be building business intelligence lattice with those recovered business intelligence information.

Building Concept Lattice

In this step, a concept lattice of business intelligence “plan rolling” is built. It will help us to view the business intelligence in a graphic vision. And certainly, it enriches the comprehensibility.

As shown from the top in Figure 9.3, each mobile contract will quote different types of services, and the contracts are categorised according with free minutes, which is 100, 200, 400, 600, and 1000. 0 means the contract is pay as you go. These properties are “plan rolling” objects. Then we have 6 different attributes which are “minutes”, “minutes rolling”, “text”, “text rolling”, “data”, and “data rolling”. According to the tabular concept context, we put those business intelligence objects and attributes into a lattice which is strictly basing on lattice definitions in Chapter 8.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>V</th>
<th></th>
<th>V</th>
<th></th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>600</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>1000</td>
<td>V</td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>0</td>
<td>V</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
</tbody>
</table>
As shown in Figure 9.3, mobile plans 400 and 600 deal with minutes rolling and data rolling quite often. It makes sense since both plans quote 400 or 600 free-minutes with 15Gb internet fair usage every month. But for the plans 100, 1000 and 0, the situation is different, plans 100 and 0 do not quote any free internet services, while plan 1000 is designed for the premier customers who are willing to pay 75 Pound Sterling every month for 1000 free-minutes, 3000 text and unlimited data. The recovered business intelligence slice matches perfectly to the information in business.

To expand the scope of business intelligence of BIR, the demos of a bigger task in business intelligence recovery is further presented. It is about to recover “sale fact of the last quarter of 2008 in a certain local area” from the business intelligence base of Web-based mobile retailing system. As shown in Figure 9.4, the business intelligence base...
has all the related business intelligence information recovered. They are “customer”, “promotion”, “time”, “store”, and “product”.

Figure 9.4: Recovered Business Intelligence of an Online-Mobile Retailing System

However, not all of above information is valuable in this specific task, a business intelligence base named “sale fact of four smart phones in the last quarter of 2008 in middle England” is defined. Figure 9.5 is an example of the business intelligence base (BIB).
This BIB is grouped by business intelligence cubes, within which the monthly retailing information of every item is recorded. All the information ideally has been well recovered and stored in the BIB.

In this BIB, there are:

- 3 Dimensions: time dimension, product dimension, and location dimension, e.g., \{[BIB], ([time], [mobile], [location])\}
- 4*3 Member: four members in each dimension, e.g., \{[location], ([Leicester], [Nottingham], [Coventry], [Derby])\}
- 3 Levels of time dimension: year, season, and month, e.g., \{[time], ([year], [season], [month])\}
- 16*4 Sets: a BI cube is a BI set, e.g., \{[time, mobile, location]\}

Each business intelligence cube contains two sets of business intelligence information: BI objects and attributes. As it can be imaged, the overall BIB is a multi-dimensional database. In Figure 9.5, there are approximately 128 sub-dimensions in the BIB for this
business intelligence recovery task. It is the reason why business intelligence base need to be decomposed in the first step in Chapter 5.

In the following, we present three demos, which has been separately coloured in Figure 9.5. The first task is to have Apple iPhone 3G sale fact in December 2008; the second task is to have HTC TyTN II sale fact at Leicester in September 2008, the final task is to have BlackBerry Bold sale fact of Middle England in last season of 2008.

**Demo 1**: Modelling Apple iPhone 3G and December 2008, see Figure 9.6:

```latex
{[\text{BIS}],
  ([\text{Product}] = [\text{Apple_iPhone_3G}],
   [\text{Time}] = [\text{December_2008}])
}
```

![Figure 9.6: BIS Modelling Demo 1](image)

**Demo 2**: Modelling HTC TyTN II, Leicester, and September 2008, see Figure 9.7:

```latex
{[\text{BIB}],
  {[[\text{BIS} = \text{Product} ([\text{HTC_TyTN_II}])],
    [[\text{BIS} = \text{Location} ([\text{Leicester}])],
      [[\text{BIS} = \text{Time} ([\text{September_2008}])]
    ]}
  }
}
```
**Demo 3:** Modelling BlackBerry Bold, Middle England, and last season of 2008, see Figure 9.8:

```plaintext
{[BIB],
 {{[BIS = Product ([BlackBerry_Bold])]},

 {{[BIS = Location ([Middle_England])],
   ([Leicester],
   [Nottingham],
   [Coventry],
   [Derby])}

 {{[BIS = Time_Last Season_2008],
   ([Last_Season_2008],
   [October_2008],
   [November_2008],
   [December_2008])}}
}
}```
Chapter 9 Case Study

Figure 9.8: BIB Modelling Demo 3

All of these three demos demonstrate that business intelligence as a unit could be recovered automatically. In the Figure 9.6, Figure 9.7 and Figure 9.8, every circle represent a subset of business intelligence, it may involve other business intelligence objects and attribute for further BIR, which is the part of the work we have already done in the previous chapters from Chapter 5 to Chapter 8.

9.3 Web Toolkit of Auto-BIR

The Web toolkit platform is developed to evaluate a bunch of embedded Auto-BIR techniques: BI base decomposition, program partition, concept recovery, and formal concept analysis, etc. The ideal process of business intelligence recovery is automatic. Therefore, the Web toolkit platform is chose to demonstrate the directly interactions between an engineer and the system. The well developed Auto-BIR techniques and the sample source code is picked from the case study, see Figure 9.9, Figure 9.10, and Figure 9.11.
The Auto-BIR Web toolkit platform is developed under Eclipse Open IDE environment. It is enabled with jre to invoke Auto-BIR user interface. The demo of Auto-BIR process contains four presentation parts: the source package, recovered BI concepts and data, tabular context, and BI lattice as shown in Figure 9.9.
The concept recovery process as discussed in section 9.2.1 is shown in Figure 9.10, the confidence of the initial state, the first round and the second round of business intelligence concept recovery are the computation results. The business intelligence slices are further generated within the process of BI oriented program partition. There are 9 BI slices as shown in the figure.
In Figure 9.11, the result of formal business intelligence concept lattice is generated. This is to demo the result of program partition and business intelligence concept recovery results as discussed in section 9.2.3.

9.4 Summary

The case study of automatic business intelligence recovery (BIR) from software system had been presented in this chapter. Following each step in previous sections, the proposed BIR methodology and the accordant techniques have been examined.

Business intelligence base decomposition method simplifies the work of business intelligence recover by partitioning monolithic business intelligence into business intelligence slices; programming type based program partition technique partition large program code into program modules; concept recovery method recovers business intelligence concept and interprets the relationship of different business intelligence concept; finally, formal concept lattice models the recovered business intelligence.
Those four steps work together very well and demonstrate one of the biggest tasks in a business intelligence recovery from a Web-based mobile retailing system.

The entire Web-based mobile retailing system is 150K LOC in C and C#. The case study picks three parts (big, medium and small) inside from the system: 200LOC, 1.5K LOC with 92 functions, and 781 LOC with 32 functions in dealing with each question of BIR.
Chapter 10

Conclusions

10.1 Summary of Thesis

After nearly two decades of Web revolution, the software systems and applications have reached a stage where all the computing components and business modules are becoming more and more internet involved. Software maintenance faces another round of issues as same as the past, e.g., security and performance, etc. Those new issues are functionally equal to the old ones, but stopping at different levels of computing and business system, since the overall platform of software and business are expanding and being leveraged. Therefore, parts of the computing in companies shall be out-sourced to several specialised Web-Service providers, which could deal with questions and tasks from a very professional point of view. Software companies, instead of licensing software installations, are licensing Web-Services through internet. The phenomenon of software and entire software industry, thereafter, changes. Amazon, eBay, Google, and Carphone Warehouse, etc, are all software companies on the other hand, and certainly being specialised in computing of each business field.

The most crucial task is, however, the understanding of business behaviours in software system in the first place: which modules in the system should be further developed in case to source out as a professional Web-Service provider. Existing methods for program understanding are mainly capable of extracting software information at a low abstraction level such as structural level. For a small portion of methods that are capable of extraction information at the application level, their techniques are heavy-weight, i.e., the recovery of a single concept involves a comprehensive checking of the software at multiple abstraction levels. As a result, their techniques cannot cope with large-scale business intelligence recovery from software code efficiently.

In this thesis a new approach to automatic recovering business intelligence from software code at large-scale has been introduced. In particular, a new concept of
“business intelligence slice” for business intelligence recovery from software code is proposed. Centring on this term, several significant technical issues in the process of automatically recovering business intelligence from software code were addressed:

**Expressiveness of business intelligence representation.** There is a trade-off between the expressiveness of business intelligence representation. In this thesis we use a simplified concept lattice as the business intelligence representation, in particular, only necessary business intelligence in the representation of concepts and relationships among concepts are accommodated. In this way, no exhaustive validation of the properties of a single concept or relationship is needed. The result from the case studies shows that business intelligence recovered from software code in this form is sufficient for understanding the underlying program.

**Accuracy of the recovered business intelligence.** Unlike previous research work on software concept and relationships in the source program and therefore render a higher degree of accuracy of recovering business intelligence from source program that other methods cannot achieve.

**Unambiguity of the information.** Business intelligence usually presents itself in the source code in an ambiguous way. Modelling and handling this ambiguity is necessary for automating mechanism that is capable of capturing the ambiguity of concepts and relationships embedded in the source code. The ambiguity of recovered business intelligence slices is also measured and resolved through an initial evaluation and a dual-way belief propagation process. The result from the case studies shows that by handling the ambiguity arise in the knowledge recovery process, more business intelligence can be recovered that otherwise.

**Confidence of the result.** Given the ambiguous nature of automatically recovering business intelligence from source code, it is beneficial to improve the reliability of recovered business intelligence whenever possible. In the process of recovering business intelligence from source program, we may find multiple evidences in the source program directly or indirectly supporting a slice of business intelligence. In this thesis, we present a new mechanism called “dual-way” belief propagation that synthesises evidence from multiple information sources in an appropriate way. In this way, the overall reliability of the recovered business intelligence is enhanced.

**Scalability of the techniques.** The automatic recovery of business intelligence involves a considerable search across the source program and is therefore sensitive to the size of the source program. Furthermore, matching between a big piece of monolithic business
intelligence and a source program is not an efficient way to recover business intelligence from the source program even if this is done by machine. In this thesis, we introduced a linear program partition technique that can partition a source program into smaller program modules and a concept of business intelligence slice that is a building block for large scale business intelligence base. The recovery of business intelligence from a source program can therefore be carried out at the scale of business intelligence slices and program modules. In this way, the computational complexity is reduced and flexibility of business intelligence recovery is also obtained since the matching between business intelligence slices and program modules can be arranged in different ways. The case study for the linear program partitioning shows the prospect of the kind of program partitioning method.

**Automation of the process.** Given business intelligence slices and program modules, the recovery of business intelligence slices from the program modules still involves a considerable search; furthermore the calculation of the value of ambiguity or uncertainty of business intelligence slices or business concepts/relationships is complicated. In this thesis, algorithms and formulae are provided that can automate these tasks. The case studies suggest that automating the whole process does significantly improve the efficiency of program understanding.

### 10.2 Computational Complexity of BIR

In Chapter 5-8 analysis has been given to the computational complexity of sub-tasks in the process of recovering business intelligence from a source program separately. In this section we summarise them and given an overall analysis of the computational complexity of the whole process. We first list the key sub-tasks in the process and their computational complexity subsequently in Table 8.1, where \( n \) is the size of the source program, \( m \) is the average size of a partitioned program module.

The overall computation complexity of the whole business intelligence recovery process is therefore \( O = (n \times m) \).

### 10.3 Original Contributions

As set out in Chapter 1, one of the initiatives for this research is to explore the possibility of applying traditional reverse engineering techniques to the area of business
intelligence recovery. As the survey of the state of the art in this area went on, quite some work that utilise traditional reverse engineering techniques were indentified. It would be beneficial to find out what kind of reverse engineering techniques these work rely on. To this end, in this section, traditional reverse engineering techniques are classified into different categories and some related work that uses reverse engineering techniques are put into these categories. Based on this, the original contributions of this research work to the reverse engineering field can be easily identified.

Generally speaking, reverse engineering is a subject of extracting system abstraction and first of all we need to identify what is system abstraction. For the point of view of the author, the business intelligence can be classified into four categories namely business intelligence data, business intelligence concept, business intelligence conceptual model, and business logics. These are system abstraction from a business intelligence point of view.

Uncovering the coordinate relationship between business intelligence recovery and software reverse engineering. The idea of business intelligence recovery could be taken to insist program comprehension is it could enrich business oriented program comprehensibility, which, on the other hand, it the bottleneck to rule based software reverse engineering systems.

Decomposing the monolithic business intelligence into business intelligence slices to handle the overall task of business intelligence recovery. Business intelligence of a software system is more than vast for us to handle. It is wiser to decompose a complex target into manageable units, and reunite the research results of every single piece.

Partitioning source program according to different program styles to have business intelligence oriented program modules. The program partition method is online, linear, and simply fast. It is good enough to prepare business intelligence recovery and match the business intelligence slices with the partitioned program modules.

Recovering business intelligence concepts and their relationships. The concepts and their relationship recovery methods are built on evaluating beliefs of match. We defined concept recovery rules and raw relationships of concepts, generated and evaluated the recovered results.

Building formal concept lattice to represent business intelligence and business logics. When all of the needed business intelligence information has been elicited and
collected, it is the final step to represent the recovered results and reunite the business intelligence slices to make an entire business intelligence base.

10.4 Conclusions of Auto-BIR in the Thesis

Auto-BIR is enabled by a bunch of well-developed reverse engineering techniques to vertically elicit business intelligence from program code to business intelligence base, which is dealing with a task in software evolution. The conclusion can be summarised as below:

- Recovered business intelligence will drive the program comprehension levels from code and structure to the levels of component module and architecture. Ideally, business intelligence can interpret the business requirement and business logic. The result will enrich business-oriented program comprehensibility, which, on the other hand, will insist software reverse engineering to finish software partition and abstraction.

- Business intelligence as a target item of recovery is monolithic and none-known. The best way in dealing with a complex question starting from scrunch is decomposition. The successful partition of business intelligence base into business intelligence slices will minimize the computation task, save computation effort, and accurate computation result.

- Program code, on the other hand, needs to be partitioned to match with the business intelligence slice, which will finish the recovery task by optimising the recovery result. Programming style based program partition probably is the simplest and fastest method to have business intelligence, since software engineers are differentiated and isolated to develop each program module of software functionality during the developing period.

- Concept recovery is the only way to have business intelligence concepts. To enrich the confidence of the recovery result of business intelligence concept, a defined recovery process enabled with rounding circle is important to make sure the recovered business intelligence concepts are correct. This feature could be taken advantage of in the former two processes as well.

- Since the recovered concepts plus the internal and external relationships of them in/among program modules are essential to interpreting business logic. And
there is no better way than graphical presentations to represent the recovered results and the embedded relationships, formal concept analysis is selected. It is a mathematical pre-proved method. The overall process of FCA has two steps: firstly, generating the tabular context of the recovered business intelligence concept, and finally, building concept lattice. The lattice modelling result in case study of Chapter 9 looks perfect.

- The overall objective in Auto-BIR mechanism is fast and accurate. Those lightweight techniques have been selected in the thesis fulfil the need of business intelligence recovery while saving computation time and energy.

10.5 Future Work

As mentioned in Chapter 4, the way of combining business intelligence decomposition method, traditional the linear programming-style-based program partitioning method, concept recovery method, and formal concept analysis method is promising research area. Furthermore, more programming styles can be identified and used as additional dimensions for program partitioning, more business intelligence slice can be identified and used as additional dimensions for business intelligence decomposition, concepts and relationships can be identified and used as additional dimensions for concept recovery. Thereafter, more detailed formal concept lattice could be built based on recovered business intelligence.

The future work of this Auto-BIR research can be concluded as three parts:

Firstly, Service Oriented Architecture (SOA). The computing platform is evolved into Web dynamics. The kernel idea of business intelligence recovery in SOA could be viewed to bridge business to business, and transform business data from one service to another. The study of SOA, middleware, interface and security will be the further tasks.

Further leveraging the level of automation mechanism will be the second task in future Auto-BIR research. Software reverse engineering techniques need to be further studied to maximum each ability in dealing sub-tasks in business intelligence recovery.

And finally applying the techniques invented in this research work to other field is one of the future works. For example, structured information source or semi-structured information source, such as web pages, are on the research agenda. But this is beyond the scope of this thesis.
References


References


[34] E. Burd, M. Munro and C. Wezeman, “Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity”, In Proceedings of
the 3rd IEEE Working Conference on Reverse Engineering (WCRE’96), pp. 189-196, Monterey, CA, USA, November 1996.


References


References

Young Researchers Workshop on Service Oriented Computing, Leicester, UK, April 2005.


Appendix A

Programming Style Based Program Partition

A.1 Source Code of Web UI in a mobile retailing system

```csharp
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Data.OleDb;

namespace com.tsd.ocs.gl
{
    /// <summary>
    /// Summary description for paramTrialBalance.
    /// </summary>
    public class paramTrialBalance : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.ImageButton o_btnOk;
        protected System.Web.UI.WebControls.Label o_lblCCName;
        protected System.Web.UI.WebControls.Label o_lblTab;
        protected System.Web.UI.WebControls.TextBox o_txtToDate;
        protected System.Web.UI.WebControls.TextBox o_txtFromDate;
        protected System.Web.UI.WebControls.ImageButton o_btnPreview;
        protected System.Web.UI.WebControls.Label o_lblMessage;
        protected System.Web.UI.WebControls.Label Label2;
        protected System.Web.UI.WebControls.ListBox o_lstAccounts;
        protected System.Web.UI.WebControls.CheckBox o_chkSelectAll;
        private Hashtable l_Param;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
            l_Param = new Hashtable();

            if (!Page.IsPostBack)
            {
                if (!Utilities.getActionRights("Trial Balance Report","Preview",Session["UserRights"].ToString()))
                {
                    Server.Transfer("Logout.aspx");
                }
            }
        }
    }
}
```
Utilities.LoadCOAList(o_lstAccounts);
}

#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
    //
    // CODEGEN: This call is required by the ASP.NET Web Form Designer.
    //
    InitializeComponent();
    base.OnInit(e);
}

/// <summary>
/// Required method for Designer support - do not modify the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.o_chkSelectAll.CheckedChanged += new System.EventHandler(this.o_chkSelectAll_CheckedChanged);
    this.o_btnPreview.Click += new System.Web.UI.ImageClickEventHandler(this.o_btnPreview_Click);
    this.o_btnOk.Click += new System.Web.UI.ImageClickEventHandler(this.o_btnOk_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
#endregion

private void o_btnOk_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    Response.Redirect("Main.aspx",true);
}

private void o_btnPreview_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    o_lblMessage.Text="";
    bool bCheck=false;
    string strMessage="";
    strMessage+=" Following fields are required: ";
    if(o_txtFromDate.Text.Trim().Equals("")
    {
        strMessage += "[From Date] 
    bCheck=true;
    }
    if(o_txtToDate.Text.Trim().Equals("")
    {
        strMessage += "[To Date] 
    bCheck=true;
    }
    if(o_lstAccounts.SelectedIndex ==-1)
    {

strMessage="Please Select atleast one Account";
bCheck=true;

if(!(o_txtFromDate.Text.Trim().Equals("")))
{
    try
    {
        DateTime.Parse(o_txtFromDate.Text);
    }
    catch(Exception Ex)
    {
        strMessage = "Please enter a valid FromDate (mm-dd-yyyy) 
        +"r\n";
        Server.ClearError();
bCheck=true;
    }
}

if(!(o_txtToDate.Text.Trim().Equals("")))
{
    try
    {
        DateTime.Parse(o_txtToDate.Text);
    }
    catch(Exception Ex)
    {
        strMessage = "Please enter a valid ToDate (mm-dd-yyyy) "
        +"r\n";
        Server.ClearError();
bCheck=true;
    }
}

if(bCheck)
{
o_lblMessage.Text=strMessage;
return;
}

l_Param.Add("FromDate",o_txtFromDate.Text);
l_Param.Add("ToDate",o_txtToDate.Text);
///<l_Param.Add("lstAccounts".getSelectedItems(o_lstAccounts));

if(getSelectedItems(o_lstAccounts).Count==0)
{
l_Param.Add("FromAccount",o_lstAccounts.Items[0].Value );
}

l_Param.Add("ToAccount",o_lstAccounts.Items[o_lstAccounts.Items.Count-1 ].Value);

Session["reportParams"]=l_Param;

private void o_chkSelectAll_CheckedChanged(object sender, System.EventArgs e)
{
    if(o_chkSelectAll.Checked)
    {
        foreach (ListItem lstItem in o_lstAccounts.Items)
        {
            l_Param.Add("Account",lstItem.Value);
            o_lstAccounts.Items.RemoveAt(lstItem.Index);
        }
    }
}
```csharp
Appendix A Example Code of Case Study

```lstItem.Selected = true;
    }
}
else
{
    foreach (ListItem lstItem in o_lstAccounts.Items)
    {
        lstItem.Selected = false;
    }
}

private ListItemCollection getSelectedItems(ListBox p_lstBox)
{
    ListItemCollection l_lstCol = new ListItemCollection();
    if(!o_chkSelectAll.Checked)
    {
        foreach (ListItem lstItem in p_lstBox.Items)
        {
            if(lstItem.Selected == true)
            { l_lstCol.Add(lstItem); }
        }
    }
    return l_lstCol;
}

namespace com.tsd.ocs.gl
{
    /// <summary>
    /// Summary description for userManagementList.
    /// </summary>
    public class userManagementList : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.DataGrid o_dgSearch;
        protected System.Web.UI.WebControls.ImageButton o_btnDelete;
        protected System.Web.UI.WebControls.ImageButton o_btnAdd;
        protected System.Web.UI.WebControls.ImageButton o_btnSearch;
        protected System.Web.UI.WebControls.Label Label3;
        protected System.Web.UI.WebControls.Label o_lblCCDescription;
        protected System.Web.UI.WebControls.Label o_lblCCName;
        protected System.Web.UI.WebControls.TextBox o_txtLoginName;
        protected System.Web.UI.WebControls.TextBox o_txtLoginID;
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.WebControls.DropDownList o_cmbLoginStatus;
        protected System.Web.UI.WebControls.DropDownList o_cmbRole;
        protected System.Web.UI.WebControls.Label l_lblMessage;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here

            if(!Page.IsPostBack)
            {
            }
        }
```
if(!(Utilities.getActionRights("User Management","View",Session["UserRights"].ToString())))
{
    Server.Transfer("Logout.aspx");
}
else
{
    Utilities.setActionRights("User Management","Add",Session["UserRights"].ToString(),o_btnAdd);
    Utilities.setActionRights("User Management","Delete",Session["UserRights"].ToString(),o_btnDelete);
}

string strSQLQuery="Select a.Adm_LoginID,a.Adm_LoginName,replace(replace(a.Adm_Active,'Y','Active'),'N','InActive') as Adm_Active,a.Adm_ID,r.rl_name from Admin a,user_roles r where a.rl_id=r.rl_id";
loadSearch(strSQLQuery);
Utilities.loadStatusCombo(o_cmbLoginStatus);
Utilities.loadCombo("Select rl_id,rl_name from user_roles where rl_active='Y'",new string[]{"rl_id","rl_name"},o_cmbRole);

private void loadSearch(string p_Query)
{
    try
    {
        Database.getHandle().DbConnection.Open();
        OleDbDataAdapter da=new OleDbDataAdapter(p_Query,Database.getHandle().DbConnection);
        DataSet ds=new DataSet("Users");
        da.Fill(ds,"dtUsers");
        ds.Tables["dtUsers"].Columns[0].ColumnName="LoginID";
        ds.Tables["dtUsers"].Columns[1].ColumnName="LoginName";
        ds.Tables["dtUsers"].Columns[2].ColumnName="Status";
        ds.Tables["dtUsers"].Columns[3].ColumnName="ID";
        ds.Tables["dtUsers"].Columns[4].ColumnName="RoleName";
        o_dgSearch.DataSource=ds.Tables["dtUsers"]; o_dgSearch.DataBind();
    }
    catch(Exception ex)
    {
        l_lblMessage.Text=ex.Message;
        Server.ClearError();
    }
    finally
    {
        Database.getHandle().DbConnection.Close();
    }
}
// CODEGEN: This call is required by the ASP.NET Web Form Designer.
//
// InitializeComponent();
// base.OnInit(e);

private void InitializeComponent()
{
    this.o_btnSearch.Click += new
System.Web.UI.ImageClickEventHandler(this.o_btnSearch_Click);
    this.o_btnAdd.Click += new
System.Web.UI.ImageClickEventHandler(this.o_btnAdd_Click);
    this.o_btnDelete.Click += new
System.Web.UI.ImageClickEventHandler(this.o_btnDelete_Click);
    this.o_dgSearch.PageIndexChanged += new
System.Web.UI.WebControls.DataGridPageChangedEventHandler(this.o_dgSearch_PageIndexChanged);
    this.Load += new System.EventHandler(this.Page_Load);
}

private void o_btnSearch_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    l_lblMessage.Text = "";
    strSQLQuery="Select
a.Adm_LoginID,a.Adm_LoginName,replace(replace(a.Adm_Active,'Y','Active'),'N','InActive') as
Adm_Active,a.Adm_ID,r.rl_name from Admin a,user_roles r " +
" Where 1=1 and a.rl_id=r.rl_id";

    if(!o_txtLoginName.Text.Trim().Equals(""))
        strSQLQuery+= " and a.Adm_LoginName like "
    +o_txtLoginName.Text.ToString() + "%";

    if(!(o_txtLoginID.Text.Trim().Equals(""))
        strSQLQuery+= " and lower(a.Adm_LoginID) like "
    +o_txtLoginID.Text.ToString().ToLower() + "%";

    if(!(o_cmbLoginStatus.SelectedIndex==0))
        strSQLQuery+= " and a.Adm_Active = "
    +o_cmbLoginStatus.SelectedValue.ToString() + "";

    if(!(o_cmbRole.SelectedIndex==0))
        strSQLQuery+= " and a.rl_id = "
    +o_cmbRole.SelectedValue.ToString() + "";

    loadSearch(strSQLQuery);
}

private void o_dgSearch_PageIndexChanged(object source, System.Web.UI.WebControls.DataGridPageChangedEventArgs e)
{
    l_lblMessage.Text = "";
    o_dgSearch.CurrentPageIndex = e.NewPageIndex;
}
string strSQLQuery="Select
a.Adm_LoginID,a.Adm_LoginName,replace(replace(a.Adm_Active,'Y','Active'),'N','InActive') as
Adm_Active,a.Adm_ID,r.rl_name from Admin a,user_roles r where a.rl_id=r.rl_id";
loadSearch(strSQLQuery);
}

private void o_btnDelete_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    l_lblMessage.Text = "";
    int intDelCount=0;
    try
    {
        Database.getHandle().DbConnection.Open();
        OleDbCommand dbCommand;
        string strSQLQuery;
        for(int i=0;i<o_dgSearch.Items.Count;i++)
        {
            if(((CheckBox)o_dgSearch.Items[i].Cells[0].Controls[1]).Checked)
            {
                strSQLQuery = "Delete From Admin where
lower(Adm_LoginID) =""+
((HyperLink)(o_dgSearch.Items[i].Cells[1].Controls[0])).Text.Trim().ToLower().ToString() +"";
                dbCommand = new OleDbCommand(strSQLQuery,
Database.getHandle().DbConnection);
                dbCommand.ExecuteNonQuery();
                intDelCount++;
            }
        }
        if(intDelCount==0)
            l_lblMessage.Text= "Please Select a Record to Delete";
        else
            l_lblMessage.Text= intDelCount + " Records Deleted
Successfully";
    }
    catch(Exception ex)
    {
        l_lblMessage.Text=ex.Message;
        Server.ClearError();
    }
    finally
    {
        Database.getHandle().DbConnection.Close();
    }
}
loadSearch("Select
a.Adm_LoginID,a.Adm_LoginName,replace(replace(a.Adm_Active,'Y','Active'),'N','InActive') as
Adm_Active,a.Adm_ID,r.rl_name from Admin a,user_roles r where a.rl_id=r.rl_id");
private void o_btnAdd_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    Response.Redirect("userManagementView.aspx?Mode=0");
}

A.2 Programming Style Sampling Function - comments

<table>
<thead>
<tr>
<th>Function</th>
<th>SSC1</th>
<th>SSC2</th>
<th>SMC1</th>
<th>SMC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>_StartDatabase</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>_ResetDatabase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_LoadIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_StartIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_ResetIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Close</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Search</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Waiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FinishWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ResetWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FinishedWaiting</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FindPrompt</td>
<td>0</td>
<td>15</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Initialise</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Restarting</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SendString</td>
<td>0</td>
<td>23</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>GetText</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TopLevel</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RollingTrans</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DoRolling</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TransHelp</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WhatRoll</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>DoRolling</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DoInteract</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RollingHelp</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BuildTextTable</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FullMenue</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AutoRolling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SetRollingCategory</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SetPrintCase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Load</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>LoadDemo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LoadState</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>New</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LoadTransData</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LoadTransTest</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GetTransLists</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InsertBefore</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InsertAfter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InsertBelow</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InsertBufferBefore</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InsertBufferAfter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CopyToText</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CommentFrom</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Delete</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Change</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ChangTo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UndeleteChange</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### A.3 Programming Style Sampling Function - naming

<table>
<thead>
<tr>
<th>Function</th>
<th>SVN1</th>
<th>SVN2</th>
<th>SVN3</th>
<th>SFN1</th>
<th>SFN2</th>
<th>SFN3</th>
</tr>
</thead>
<tbody>
<tr>
<td>_StartDatabase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>_ResetDatabase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>_LoadIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>_StartIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>_ResetIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Close</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Search</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Waiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FinishWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ResetWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FinishedWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FindPrompt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Initialise</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Restarting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SendString</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>GetText</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TopLevel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RollingTrans</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DoRolling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TransHelp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>WhatRoll</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DoRolling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DoInteract</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
A.4 Programming Style Sampling Function - indentation

<table>
<thead>
<tr>
<th>Function</th>
<th>SU1</th>
<th>SU2</th>
<th>SU3</th>
<th>SU4</th>
<th>SU5</th>
<th>SI1</th>
<th>SI2</th>
<th>SI3</th>
<th>SI4</th>
<th>SI5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_StartDatabase</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>_ResetDatabase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_LoadIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Command</td>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_StartIndex</td>
<td>0 0 0 3 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_ResetIndex</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>0 0 0 0 6 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Waiting</td>
<td>0 8 0 0 0 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FinishWaiting</td>
<td>0 0 0 0 0 2 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResetWaiting</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FinishedWaiting</td>
<td>0 0 0 0 0 4 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindPrompt</td>
<td>0 0 0 0 0 16 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialise</td>
<td>0 2 0 0 0 0 0 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restarting</td>
<td>0 2 0 0 0 0 0 0 2 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SendString</td>
<td>0 2 0 0 0 28 0 85 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetText</td>
<td>0 6 0 0 0 0 0 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TopLevel</td>
<td>0 26 0 0 0 5 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RollingTrans</td>
<td>0 0 0 0 0 2 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoRolling</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TransHelp</td>
<td>0 16 0 0 0 3 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WhatRoll</td>
<td>0 4 0 0 0 20 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoRolling</td>
<td>0 3 0 0 0 16 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoInteract</td>
<td>0 8 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RollingHelp</td>
<td>0 3 0 0 0 4 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BuildTextTable</td>
<td>0 6 0 0 0 0 0 2 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FullMenu</td>
<td>0 31 0 0 0 0 0 0 2 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoRolling</td>
<td>0 24 0 0 0 0 0 0 2 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetRollingCategory</td>
<td>0 3 0 0 0 0 0 0 2 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetPrintCategory</td>
<td>0 13 0 0 0 0 0 6 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadDemo</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadState</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadTransData</td>
<td>0 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadTransTest</td>
<td>0 7 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetTransLists</td>
<td>0 7 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertBefore</td>
<td>0 19 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertAfter</td>
<td>0 5 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertBelow</td>
<td>0 7 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertBufferBefore</td>
<td>0 13 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertBufferAfter</td>
<td>0 12 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CopyToText</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CommentFrom</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChangTo</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndeleteChange</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undo</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redo</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RedoDemo</td>
<td>0 6 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stop</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redo</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fix</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetCallGraphInfo</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Prog</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Item</td>
<td>0 4 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_New</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Plot</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Muse</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deduce</td>
<td>0 3 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MakeFold</td>
<td>0 5 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeleteFold</td>
<td>0 7 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowComment</td>
<td>0 6 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetComment</td>
<td>0 2 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Home</td>
<td>0 2 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix A Example Code of Case Study

A.5 Filtered Programming Style Sampling Function

<table>
<thead>
<tr>
<th></th>
<th>SSC</th>
<th>SMC</th>
<th>SVN</th>
<th>SFN</th>
<th>SUI</th>
<th>SNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>_StartDatabase</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_ResetDatabase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_LoadIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_StartIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_ResetIndex</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Close</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Search</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Waiting</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FinishWaiting</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ResetWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FinishedWaiting</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FindPrompt</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Initialise</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Restarting</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>SendString</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>GetText</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>TopLevel</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RollingTrans</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DoRolling</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TransHelp</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>WhatRoll</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DoRolling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DoInteract</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>RollingHelp</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BuildTextTable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>FullMenu</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>AutoRolling</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>SetRollingCategory</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>SetPrintCase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Load</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LoadDemo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>LoadState</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>New</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>LoadTransData</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>LoadTransTest</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>GetTransLists</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>InsertBefore</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>InsertAfter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>InsertBelow</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>InsertBufferBefore</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>InsertBufferAfter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>CopyToText</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>CommentFrom_</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Delete</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Change</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ChangTo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
### A.6 Cutting Source Program into Program Sections

<table>
<thead>
<tr>
<th>Function</th>
<th>SSC</th>
<th>SMC</th>
<th>SVN</th>
<th>SFN</th>
<th>SUI</th>
<th>SNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>_StartDatabase</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_ResetDatabase</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_LoadIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_StartIndex</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_ResetIndex</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Close

Search

<table>
<thead>
<tr>
<th>Function</th>
<th>SSC</th>
<th>SMC</th>
<th>SVN</th>
<th>SFN</th>
<th>SUI</th>
<th>SNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FinishWaiting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

ResetWaiting

<table>
<thead>
<tr>
<th>Function</th>
<th>SSC</th>
<th>SMC</th>
<th>SVN</th>
<th>SFN</th>
<th>SUI</th>
<th>SNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FinishedWaiting</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FindPrompt</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Initialise</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Restarting</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SendString</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
### GetText

<table>
<thead>
<tr>
<th>TopLevel</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RollingTrans</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

### DoRolling

| TransHelp | 1 | 0 | 0 | 3 | 4 | 4 |
| WhatRoll | 1 | 1 | 0 | 3 | 4 | 4 |

### DoRolling

| DoInteract | 1 | 1 | 0 | 3 | 2 | 2 |
| RollingHelp | 1 | 0 | 0 | 3 | 2 | 2 |
| BuildTextTable | 1 | 0 | 0 | 3 | 2 | 0 |
| FullMenu | 1 | 0 | 0 | 3 | 2 | 2 |
| AutoRolling | 2 | 0 | 0 | 3 | 2 | 2 |
| SetRollingCategory | 0 | 0 | 0 | 3 | 2 | 2 |
| SetPrintCase | 0 | 0 | 0 | 3 | 2 | 0 |
| Load | 0 | 0 | 1 | 3 | 2 | 2 |

### LoadDemo

| LoadState | 0 | 0 | 0 | 3 | 2 | 3 |

### New

| LoadTransData | 0 | 0 | 0 | 3 | 2 | 3 |
| LoadTransTest | 2 | 1 | 0 | 3 | 2 | 3 |
| GetTransLists | 0 | 0 | 0 | 3 | 2 | 3 |

### InsertBefore

| InsertAfter | 0 | 0 | 0 | 3 | 2 | 0 |
| InsertBelow | 0 | 0 | 0 | 0 | 2 | 2 |
| InsertBufferBefore | 0 | 0 | 0 | 3 | 2 | 0 |
| InsertBufferAfter | 0 | 0 | 0 | 3 | 2 | 0 |
| CopyToText | 0 | 0 | 0 | 3 | 2 | 0 |
| CommentFrom_ | 0 | 0 | 0 | 0 | 2 | 0 |
| Delete | 0 | 0 | 0 | 3 | 2 | 0 |
| Change | 0 | 0 | 0 | 0 | 2 | 0 |
| ChangTo | 0 | 0 | 0 | 3 | 2 | 0 |
| UndeleteChange | 0 | 0 | 0 | 3 | 2 | 0 |
| Undo | 0 | 1 | 0 | 3 | 2 | 0 |
| Redo | 0 | 0 | 0 | 3 | 2 | 0 |
| RedoDemo | 0 | 0 | 0 | 3 | 2 | 0 |
| Start | 0 | 0 | 0 | 3 | 2 | 0 |
| Stop | 0 | 0 | 0 | 3 | 2 | 0 |
| Redo | 2 | 0 | 0 | 3 | 2 | 0 |
| Fix | 0 | 0 | 0 | 0 | 2 | 0 |
| GetCallGraphInfo | 0 | 0 | 0 | 0 | 2 | 0 |
| _Prog | 0 | 0 | 0 | 3 | 2 | 0 |
| _Item | 0 | 0 | 0 | 3 | 2 | 0 |
| _New | 0 | 0 | 0 | 0 | 2 | 0 |
Appendix A Example Code of Case Study

Plot 0 0 0 0 2 0
Muse 0 0 0 3 2 0
Deduce 0 0 0 0 2 0
MakeFold 0 0 0 0 2 0
DeleteFold 0 0 0 3 2 0
ShowComment 0 0 0 3 2 0
SetComment 0 0 0 3 2 2
Home 0 0 0 3 2 0
Up 0 0 0 0 2 0
Down 0 0 0 0 2 0
Left 2 0 0 3 2 0
Right 2 0 0 3 2 0
Next 2 0 0 0 2 0
Previous 2 0 0 0 2 0
ClimbToStatement 2 0 0 3 2 0
MoveToNextItem 2 0 0 3 2 2
ExtendRight 2 0 0 3 2 0
ExtendAll 2 0 0 3 2 0
UnextendAll 2 0 0 3 2 0
NextExtend 2 0 0 0 2 0
_StartDatabase 2 0 0 0 2 0
_ResetDatabase 2 0 0 3 2 0
_LoadIndex 2 0 0 3 2 0
_StartIndex 2 0 0 3 2 0
Appendix B

Business Intelligence Concept Recovery

B.1 Source Code of Data Base in a Mobile Retailing System

usingSystem;
usingSystem.Collections;
usingSystem.ComponentModel;
usingSystem.Data;
usingSystem.Drawing;
usingSystem.Web;
usingSystem.Web.SessionState;
usingSystem.Web.UI;
usingSystem.Web.UI.WebControls;
usingSystem.Data.OleDb;

namespacecom.tsd.ocs.gl
{
/// <summary>
/// Summary description for financialYearList.
/// </summary>
/// </summary>
publicclassfinancialYearList: System.Web.UI.Page
{
    protectedSystem.Web.UI.WebControls.DataGridodgSearch;
    protectedSystem.Web.UI.WebControls.Labell_lblMessage;
    protectedSystem.Web.UI.WebControls.LabeloilblCCName;
    protectedSystem.Web.UI.WebControls.TextBoxo_txtFINStartDate;
    protectedSystem.Web.UI.WebControls.LabelLabel3;
    protectedSystem.Web.UI.WebControls.DropDownListocmbFINStatus;
    protectedSystem.Web.UI.WebControls.ImageButtonobtnDeleteAll;
    protectedSystem.Web.UI.WebControls.ImageButtonobtnAdd;
    protectedSystem.Web.UI.WebControls.ImageButtonobtnSearch;
    protectedSystem.Web.UI.WebControlsImageButtonImage1;

    privatevoidPage_Load(objectsender,System.EventArgs e)
    {
        // Put user code to initialize the page here
        if(!Page.IsPostBack)
        {
            if(!(Utilities.getActionRights("Financial Year","View",Session["UserRights"]).ToString())))
            {
                Server.Transfer("Logout.aspx");
            }
            else
            {

            }
        }
    }
}
Utilities.setActionRights("Financial Year","Add",Session["UserRights"].ToString(),o_btnAdd);
Utilities.setActionRights("Financial Year","Delete",Session["UserRights"].ToString(),o_btnDeleteAll);

string strSQLQuery="Select CONVERT(CHAR(10),fin_startdate,110) as fin_startdate, CONVERT(CHAR(10),fin_enddate,110) as fin_enddate,replace(replace(fin_active,'Y','Active'),'N','InActive') as fin_active,fin_id from financial_year order by fin_startdate";
loadSearch(strSQLQuery);
Utilities.loadStatusCombo(o_cmbFINStatus);

private void loadSearch(string p_Query)
{
    try
    {
        Database.getHandle().DbConnection.Open();
        OleDbDataAdapter da=new OleDbDataAdapter(p_Query,Database.getHandle().DbConnection);
        DataSet ds=new DataSet("FinancialYear");
        da.Fill(ds, "dtFinancialYear");
        ds.Tables["dtFinancialYear"].Columns[0].ColumnName="StartDate";
        ds.Tables["dtFinancialYear"].Columns[1].ColumnName="EndDate";
        ds.Tables["dtFinancialYear"].Columns[2].ColumnName="Status";
        ds.Tables["dtFinancialYear"].Columns[3].ColumnName="ID";
        o_dgSearch.DataSource=ds.Tables["dtFinancialYear"]; o_dgSearch.DataBind();
    }
    catch(Exception ex)
    {
        l_lblMessage.Text=ex.Message;
        Server.ClearError();
    }
    finally
    {
        Database.getHandle().DbConnection.Close();
    }
}

#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
    // CODEGEN: This call is required by the ASP.NET Web Form Designer.
    InitializeComponent();
    base.OnInit(e);
}
private void InitializeComponent()
{
    this.o_btnSearch.Click += new System.Web.UI.ImageClickEventHandler(this.o_btnSearch_Click);
    this.o_btnAdd.Click += new System.Web.UI.ImageClickEventHandler(this.o_btnAdd_Click);
    this.o_btnDeleteAll.Click += new System.Web.UI.ImageClickEventHandler(this.o_btnDeleteAll_Click);
    this.o_dgSearch.PageIndexChanged += new System.Web.UI.WebControls.DataGridPageChangedEventHandler(this.o_dgSearch_PageIndexChanged);
    this.Load += new System.EventHandler(this.Page_Load);
}

private void o_dgSearch_PageIndexChanged(object source, System.Web.UI.WebControls.DataGridPageChangedEventArgs e)
{
    l_lblMessage.Text = "";
    o_dgSearch.CurrentPageIndex = e.NewPageIndex;
    string strSQLQuery = "Select CONVERT(CHAR(10),fin_startdate,110) as fin_startdate, CONVERT(CHAR(10),fin_enddate,110) as fin_enddate, replace(replace(fin_active,'Y','Active'),'N','InActive') as fin_active,fin_id from financial_year order by fin_startdate";
    loadSearch(strSQLQuery);
}

private void o_btnDeleteAll_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    l_lblMessage.Text = "";
    int intDelCount = 0;
    try
    {
        Database.getHandle().DbConnection.Open();
        OleDbCommand dbCommand;
        string strSQLQuery;
        for(int i = 0; i < o_dgSearch.Items.Count; i++)
        {
            if(((CheckBox)o_dgSearch.Items[i].Cells[0].Controls[1]).Checked) {
                strSQLQuery = "Delete From financial_year where fin_startdate =" + ((HyperLink)(o_dgSearch.Items[i].Cells[1].Controls[0])).Text.Trim().ToLower().ToString() + "'";
                dbCommand = new OleDbCommand(strSQLQuery, Database.getHandle().DbConnection);
                dbCommand.ExecuteNonQuery();
                intDelCount++;
            }
        }
    }
}
if(intDelCount==0)
    l_lblMessage.Text="Please Select a Record to Delete";
else
    l_lblMessage.Text= intDelCount + " Records Deleted Successfully";
}
catch(Exception ex)
{
    l_lblMessage.Text=ex.Message;
    Server.ClearError();
}
finally
{
    Database.getHandle().DbConnection.Close();
}

loadSearch("Select CONVERT(CHAR(10),fin_startdate,110) as fin_startdate,
CONVERT(CHAR(10),fin_enddate,110) as
fin_enddate,replace(replace(fin_active,'Y','Active'),'N','InActive') as fin_active,fin_id from financial_year
order by fin_startdate");

private void o_btnAdd_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    Response.Redirect("financialYearView.aspx?Mode=0");
}

private void o_btnSearch_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
    l_lblMessage.Text = "";
    bool bCheck=false;
    string strSQLQuery="Select CONVERT(CHAR(10),fin_startdate,110) as 
fin_startdate, CONVERT(CHAR(10),fin_enddate,110) as 
fin_enddate,replace(replace(fin_active,'Y','Active'),'N','InActive') as fin_active,fin_id from financial_year
where 1=1 ";

    if(!o_txtFINStartDate.Text.Trim().Equals(""))
    {
        try
        {
            DateTime.Parse(o_txtFINStartDate.Text.ToString());
            strSQLQuery+= " and fin_startdate ="
            +o_txtFINStartDate.Text.ToString() + "",
        }
        catch(Exception Ex)
        {
            l_lblMessage.Text="Invalid start date entered format(mm-
dd-yyyy)";
            Server.ClearError();
            bCheck=true;
        }
    }
if(!o_txtFINEndDate.Text.Trim().Equals(""))
{
    try
    {
        DateTime.Parse(o_txtFINEndDate.Text.ToString());
        strSQLQuery+= " and fin_enddate =" + o_txtFINEndDate.Text.ToString() + ""
    }
    catch(Exception Ex)
    {
        l_lblMessage.Text="Invalid End date entered format(mm-dd-yyyy)";
        Server.ClearError();
        bCheck=true;
    }
}

//Check Field Validation
if(bCheck)
{
    return;
}

if(!(o_cmbFINStatus.SelectedIndex==0))
{
    strSQLQuery+= " and fin_active = " + Utilities.toDBString(o_cmbFINStatus.SelectedValue.ToString()) + ""
}

strSQLQuery += " order by fin_startdate";
loadSearch(strSQLQuery);

B.2 Filtering Business Intelligence Concepts - variable naming

<table>
<thead>
<tr>
<th>SVN1</th>
<th>SVN2</th>
<th>SVN3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCName</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Tab</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ToDate</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>FromDate</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Message</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Account</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SelectAll</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>TrailBalance</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Report</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Preview</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ServiceTransfer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>InitialiseComponent</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Response</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Redirect</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Sender</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ToDateText</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DateTimeParse</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
B.3 Filtering Business Intelligence Concepts - parameter naming

<table>
<thead>
<tr>
<th>Parameter</th>
<th>SPN1</th>
<th>SPN2</th>
<th>SPN3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCName</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tab</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ToDate</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>FromDate</td>
<td>2</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Message</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Account</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>SelectAll</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Column</td>
<td>Rows</td>
<td>Columns</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------</td>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>TrailBalance</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Report</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Preview</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ServiceTransfer</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>InitialiseComponent</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Response</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Redirect</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sender</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ToDateText</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DateTimeParse</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CheckingChange</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ListItem</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DropDownList</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Status</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Role</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ActionRight</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>rl_id</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rl_name</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rolename</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>WebData</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Admin</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>getHandle</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add_Click</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Minute</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Text</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Data</td>
<td>31</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>800</td>
<td>15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unlimited</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Year</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Month</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Date</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RollingText</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RollingData</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RollingMinute</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ContractPlan</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Agreement</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Address</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Manger_ID</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BankingAddress</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BankingAccount</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Balance</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Credit</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FreeMinute</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FreeData</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FreeText</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Promotion</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Upgrade</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Search</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Index</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_count</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Delete</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### B.4 Filtering Business Intelligence Concepts - method naming

<table>
<thead>
<tr>
<th>SMN1</th>
<th>SMN2</th>
<th>SMN3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCName</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tab</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ToDate</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>FromDate</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Message</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Account</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>SelectAll</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TrialBalance</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Report</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Preview</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ServiceTransfer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>InitialiseComponent</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Response</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Redirect</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sender</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ToDateText</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DateTimeParse</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CheckingChange</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ListItem</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DropDownList</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Status</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Role</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ActionRight</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>rl_id</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rl_name</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rolename</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>WebData</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Admin_</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>getHandle</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add_Click</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Minute</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Text</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Data</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>800</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>600</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unlimited</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Year</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Month</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Date</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RollingText</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RollingData</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RollingMinute</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ContractPlan</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Agreement</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Address</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Store</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Manger_ID</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BankingAddress</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BankingAccount</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Balance</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Credit</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FreeMinute</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FreeData</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>----------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>FreeText</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Promotion</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Upgrade</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Search</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Index</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_count</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Delete</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix C

List of Publications by Author


