Extraction of objects from legacy systems: an example using cobol legacy systems

Salurso, Maria Anna

How to cite:
Salurso, Maria Anna (1998) *Extraction of objects from legacy systems: an example using cobol legacy systems*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/4673/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.
Extraction of Objects from Legacy Systems:

An Example Using COBOL Legacy Systems

Supervisor
Malcolm Munro

Candidate
Maria Anna Salurso

Academic Year 1997
This thesis is dedicated to my mother Maria Antonietta and my father Ettore, for the patience in the hard task of being parents of a student.
ABSTRACT

In the last few years the interest in legacy information system has increased because of the escalating resources spent on their maintenance. On the other hand, the importance of extracting knowledge from business rules is becoming a crucial issue for modern business. Sometimes, because of inappropriate documentation, this knowledge is essentially only stored in the code. A way to improve their use and maintainability in the present environment is to migrate them into a new hardware/software platform reusing as much of their experience as possible during this process. This migration process promotes the population of a repository of reusable software components for their reuse in the development of a new system in that application domain or in the later maintenance processes.

The actual trend in the migration of a legacy information system, is to exploit the potentialities of object-oriented technology as a natural extension of earlier structured programming techniques. This is done by decomposing the program into several agent-like modules communicating via message passing, and providing to this system some object-oriented key features. The key step is the "object isolation", i.e. the isolation of groups of routines and related data items to candidates in order to implement an abstraction in the application domain.

The main idea of the object isolation method presented here is to extract information from the data flow, to cluster all the procedures on the base of their data accesses. It will examine "how" a procedure accesses the data in order to distinguish several types of accesses and to permit a better understanding of the functionality of the candidate objects. These candidate modules support the population of a repository of reusable software components that might be used as a basis of the process of evolution leading to a new object-oriented system reusing the extracted objects.
ACKNOWLEDGEMENTS

I would like to thank Malcolm Munro for his rigorous and friendly supervision, and Professor Aniello Cimitile for encouraging me to come to Durham.

Thanks also for the facilities provided by Professor Keith Bennett and all the members of the Centre for Software Maintenance. My time in the University of Durham has been both enjoyable and interesting. Particularly, thanks to Liz Burd for reading the draft of this thesis.

Needless to write here the names of all my friends in Durham. They all know how much I appreciated their friendship and their support. I wish just to nominate those of them whom I had more coffees with: thanks Sahab, Marco, Riccardo, Isabelle, Raija, Edy and Nobuko. I’ll be missing all of you.

My final thoughts are for a very special friend, Tommaso, who has been patiently waiting for me in Italy during the last year.
COPYRIGHT STATEMENT

The copyright of this thesis rests with the author. No quotation from it should be published without prior written consent and information derived from it should be acknowledged.
CONTENTS

ABSTRACT .................................................................................................................. iii

ACKNOWLEDGEMENT .............................................................................................. iv

COPYRIGHT STATEMENT ............................................................................................ v

CONTENTS .................................................................................................................. vi

1. INTRODUCTION ...................................................................................................... 1

  1.1 CRITERIA FOR SUCCESS ..................................................................................... 6
  1.2 PLAN OF THE THESIS ......................................................................................... 7

2. A REVIEW OF EXISTING TECHNIQUES .................................................................. 9

  2.1 INTRODUCTION .................................................................................................. 9
  Reverse Engineering ................................................................................................... 9
  Reengineering .............................................................................................................. 9

  2.2 REVERSE ENGINEERING METHODS .................................................................. 12
    2.2.1 RECasting ..................................................................................................... 12
    2.2.2 REdo ........................................................................................................... 14
    2.2.3 REOrganizing .............................................................................................. 18

  2.4 ALternative Approaches to Migrate Legacy Information Systems ....................... 21

  2.3 PROCESSES OF REUSE REENGINEERING ...................................................... 22
    2.3.1 The Paradigm of Reuse Reengineering (RE²) .................................................. 23
      Candidature Phase ................................................................................................. 25
      Election Phase ...................................................................................................... 26
      Qualification Phase .............................................................................................. 27

  2.4 CLASSES OF CANDIDATURE CRITERIA TO DECOMPOSE LEGACY
      INFORMATION SYSTEMS ....................................................................................... 29
      Functional Abstraction ......................................................................................... 29
      Data Abstraction .................................................................................................. 31
      Control Abstraction ............................................................................................. 33
2. 5 Graph Theory Approaches .............................................................................. 33
2. 5.1 The Call Dependencies of Cimitile and Visaggio ................................. 34
2. 5.2 The Algorithms of Liu and Wilde ............................................................ 36
2. 5.3 The Algorithm of Dunn and Knight .......................................................... 38
2. 5.4 The Algorithm of Canfora et al. ................................................................. 39
2. 6 Summary .......................................................................................................... 43

3. Program Representations .................................................................................. 44
   Data Flow Analysis .............................................................................................. 44
   Call Graph ............................................................................................................. 45

3. 2 A New Code Representation .......................................................................... 45
   3. 2.1 Analysis of Data Access Type ................................................................. 46
   3. 2.2 The Temporal Graph ............................................................................... 49

4. An Improved Technique .................................................................................... 54
   4. 1 Overview of the Algorithm ......................................................................... 56
   4. 2 Details ........................................................................................................... 57
      4. 2.1 Drawing the Program Representations ............................................... 58
         The Bipartite Graph ...................................................................................... 58
      4. 2.2 The Computation of the Vector ΔIC( ) ................................................ 59
      4. 2.3 Data Duplication .................................................................................... 60
      4. 2.4 Data Refining ......................................................................................... 68
         Evaluation of the Data Refining Phase ....................................................... 69
      4. 2.5 Termination of the Algorithm ............................................................... 70

5. A Case Study on a COBOL Source Code ....................................................... 71
   5. 1 The Need of a Standard: The ANSI COBOL Standard ......................... 73
   5. 2 Presentation of the Case Study ................................................................... 75
   5. 3 Isolation of Procedures and Data ............................................................... 76
      5. 3.1 Isolation of Procedures ....................................................................... 76
      5. 3.2 Data Analysis ......................................................................................... 78
         Unused Analysis ........................................................................................... 79
         Data Accesses ............................................................................................ 79
      5. 3.3 The Bipartite Graph .............................................................................. 82
         Computation of Indexes IC( ) and ΔIC( ) ................................................ 84
   5. 4 Data Refining Phase .................................................................................... 84
541 Data Duplication

6. Future Works

6.1 Extension of Data Duplication Phase

6.2 Data Normalisation

7. Conclusions

7.1 Evaluation of the Criteria for Success

7.1.1 Description and evaluation of existing methods

7.1.2 Formalization of a language-independent method

7.1.3 Application of the method to a case study

References
Chapter 1

Introduction

Since the early 80’s, both in academic institutions and in a wide range of working environments, the problem of the evolution of existing information systems has become a broad interest. With time, these legacy information systems, as Dietrich calls them, have increasingly become an integral part of the fabric of many organisation, growing bigger and more complex than they were originally. The costs of keeping them operational and acceptable consumed a significant proportion - up to 70% - of the software system life cycle budget [A92] [B91a] [B81] [L880]

In addition to these escalating costs, the recession in the early 90’s led to severe cuts in the budget for the development of new systems [A94]. This is also confirmed by Bernstein [B93a], who estimates that of $100 billion dollars annual expenditure of companies on software, at least 70% will be spent on maintaining their systems, while the other 30% will be spent on new development. Consequently, companies are increasingly directing their efforts to get more from the existing systems but also to ensure that these systems are much more maintainable than it was originally demanded. The existing information systems are today’s “assets” to be protected [P95] [B91c], and exploited by extracting knowledge and business rules [CO90] that, because of inappropriate documentation, are sometimes only contained within the code [CD95]. In fact, it is becoming increasingly well known that an existing system could be a repository of ideas and could enable the identification of building blocks for development of future systems [S87], as it contains management, operational and financial information about an organisation that has been accrued over many
years.

To highlight further the actual situation, a recent study [IBM] estimates that, in average, the size of a legacy information system increases by roughly 10% each year because of normal maintenance and upgrade. This leads to a doubling in size about every seven years. Consequently, the difficulties of comprehension of the legacy code are increasing, making harder any attempt to evolve the code. These considerations, with the lack of technique to solve the legacy information systems problem and the escalating resources spent on their management, lead to “information system apoplexy” [BS95], and to a common conviction that dealing with the problem can not be further postponed.

As the area concerning legacy information systems is relatively young, there is no definition that exactly establishes when an information system is “legacy”. Moreover, in most of these systems there undoubtedly exists a set of common features: enormity (millions of LOCs sometime written in a single, monolithic block with conventional or ad-hoc languages such as Assembler, COBOL, PL/1, FORTRAN and even APL), old age, inflexibility, inconsistency or complete lack of documentation, inappropriate management of data, presence of inaccurate functions and inadequacy or lack of interaction between system components.

Not all legacy information systems corresponds to this stereotype; sometimes, an information system can be legacy even if it has been developed recently and with modern techniques, but it cannot be easily adapted to the continuous changing requirements of strategies and practices in a modern business. This lead Brone and Stonebraker [BS95] to define a legacy information system not in regard to the features above, but, quite informally, as “any information system that significantly resists modification and evolution to meet new and constantly changing business requirements”.

One of the greatest challenges facing software engineers is the management and control of these changes [BH85]. This is indicate in the time spent and in the effort required to keep software systems operational after release.

The discipline concerned with changes related to an information system after delivery is traditionally known as “software maintenance”. There exist many
different definitions of software maintenance [O90] [ANS183] [LS80], some of which highlight particulars activities carried out during maintenance processes. Cornelius et al. [CMR88] insist on a general view which consider software maintenance as "any work that is undertaken after delivery of a software system". An interesting point of view is the one of Layzell and Macaulay, defining a maintenance processes as a "need-to-adapt" activity, which entails changing the software when its operational environment or original requirements changes, or as an activity to support the users of the system [LM90]. In the 1993, the IEEE Software Maintenance Standard combined these different views, defining the software maintenance as "modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [V93A].

However, it seems that maintenance still keeps its traditional meaning of restoration in response to the gradual deterioration of parts due to an extended use [GJM91], which is simply corrective maintenance. In contrast, adaptive and perfective changes [S76] [LS80] performed on legacy information systems, does not only involve correction of malfunctioning, but also entails adapting and enhancing the system to meet the evolving need of the users [LS80] and their organisations. Consequently, many authors have advanced alternative terms as "software evolution" [A88] [GJM91] [L85], "post-delivery evolution" [MD91] and "support" [FJN89] [LM90] that are considered more inclusive and encompass most, if not all, of the activities undertaken on the existing systems to keep them operational and acceptable to the users. These alternative definitions also have a more positive image than the term "maintenance".

The legacy information system problem has undoubtedly increased the significance given to the maintenance processes. In the early days the interest in the legacy information system problem concentrated on the study of tools, technique and technologies supporting the maintenance process. By comparing the small number of publications and active researches at that time with the
situation today, it is possible to say that software maintenance was receiving much less attention than the development of new systems [MD91c]. In industry, software maintenance was "categorised as dull, un-exiting detective work" [H88].

Today, the scenario is different, but the binomial "software maintenance" / "legacy information systems problem" is even more strongly related. It has broadly diffused the belief that the existing software is the "accumulating of years of experience and refinement and, however imperfect, it is a valuable asset" [B91c]. This economic heritage has to be safeguarded by making its life longer with judicious processes of evolution, and to be exploited by reusing its components in developing ex-novo of new information systems. In the meantime, the fall in the real cost of hardware and the progress in software capability inspiring to ever more ambitious development projects, are challenging the qualified and experienced development staff to improve productivity significantly, while maintaining and improving quality [MD91c].

One method proposed for making a significant improvement in productivity and quality is software reuse [CCR90]. By reusing product, processes and personal knowledge to implement changes, productivity can be greatly increased because of the reduction in the time and effort that would have been spent on specification, design, implementation and testing the changes. The reliability and robustness of the reusable software components is greater as they have been well tested and already shown to satisfy the desired requirements. Consequently, they have fewer residual errors, and this makes software reuse attractive to software engineers interested in improving the quality of the software product [GT96].

These considerations lead to a widespread interest in software reuse. In the literature, there can be found many different definitions of software reuse [DH89] [CH91] [K87]. There is the simplistic view which defines it in term of simply

---

The terms "software evolution" and "post-delivery evolution" are similar because both highlight the tendency of software to evolve, and they are used as synonym by many authors.
reuse of code, without taking into consideration the reuse of other forms of software-related knowledge. A more comprehensive and maintenance-related view is that of Biggerstaff and Perlis, which defines software reuse as "the re-application of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development or maintenance of that other system" [BP89].

However, the complete replacement of a legacy information system excluding the reuse part of the system's components is not usually a viable option, because the risks and the costs associated with complete system replacement are very high [V80] [BS95]. Furthermore, the decision about the evolution of a legacy information system should take into account not only the economic constraints - in a survey carried out by Tamai and Torimitsu [TT92], several of the respondents who once considered replacement of their system abandoned the idea because it was too expensive - but also the residual errors in the new system. The creation of a new system does not guarantee that it will work better than the existing one.

As an alternative, the existing system has to be "evolved to an higher state, providing more sophisticated used-driven functionality, the capability of deploying cutting edges technologies and of allowing the integration of other systems in a cost-effective manner" [GT96]. A number of techniques, methods, tools and management practices are used to meet these goals.

At the current state of the art, the solution seems to be the migration of legacy information systems. This involves analysis and improved understanding of the system, followed by a traditional forward engineering process using a suitable alternative paradigm or hardware/software platform. In order to take advantage of the modern technologies, the new platform should be object oriented.

The migration towards object oriented technology also seems a promising way to guarantee reusability [BMW96a] [BMW96b] as well as adaptability either in the creation of new information systems or in their modification. A big advantage of object oriented technology is the possibility to model and simulate
real world entities with their rich semantic content making the system easier to understand and to maintain. This simulation of productive processes—through objects and operations—supports the unavoidable requirements of business change. The incremental integration of new functionality is easier because application development becomes faster and cheaper. Furthermore, object oriented technology makes it easier to collect together existing components, and to tailor them individually due to inheritance, thus promoting reuse.

In the literature there are many papers formally or informally showing the productivity of a combination of reengineering, object oriented paradigm and software reuse. Takang et al [GT96] refer to the “threesome marriage” between these three elements.

In book “Migrating to Object Technology“, Graham [G94] says “object technology is productive because of the potential to reuse existing components via their specification class libraries, whether for code or specification, are the repository of productivity. Object technology assists productivity because object oriented models are easier to debug due to their richer semantic context. It is also more productive because of the semantic richness of its model and because they are model rather than procedural, imperative description”

11 Criteria for Success

The work that will be presented in this thesis can be classified into the mainstream of the work targeted to isolate, from the existing code, software fragments implementing abstractions of entities within the application domain. The criteria for success, to be judged in the final chapter, are as follows:

- description and evaluation of the existing methods to isolate reusable object-like modules,
- formalization of a language-independent method for the identification of object-like modules from existing code,
- application of the method to a case study, in order to check the flexibility of the method to be adapted to the peculiarity of a conventional language.
1.2 Plan of the Thesis

This thesis focuses the effort of migration as a form of reengineering process from a procedural legacy information system to an object oriented platform. Particularly, the aim is the extraction of procedures and related data items having object oriented features, in order to populate a repository of reusable modules extracted from the code. This "object identification" might be the basis of a process of evolution leading to a new object oriented system reusing the extracted objects.

The remainder of this thesis is structured as follow.

The second chapter focus on frames to reverse engineer and reengineer legacy information systems in order to help any intervention aimed at the evolution of code, also allowing the extraction reusable modules from the source code. Particularly, in section 2.2, some reverse engineering techniques dealing with COBOL legacy information systems are presented. In section 2.2.4, the wrapping techniques are briefly introduced, with some related problematic. Section 2.3 is specifically dedicated to approaches dealing the legacy information system problem while populating a repository of "spare parts" to be reused in the development of a new information system. In section 2.5 the approaches to extract objects-like modules based on the graph theory are presented.

In the third chapter a program representation suitable to fit our "object isolation" method is defined. It is aimed to simplify the process of understanding the relationships of common data accesses between procedures implementing entities of the application domain. This new program representation is a variant of an inter-procedural call graph, providing information about data flow. Since it is sensitive about the temporal sequence of the invoking statements and of data accesses, it will be called "temporal graph".

In the fourth chapter an algorithm to identify candidate object-like modules from existing code is presented. In section 4.2, all the details relative to the iterative algorithm are presented. After having represented the code by a bipartite graph (section 4.2.1), the algorithm performs three phases while the...
bipartite graph is not in the form of isolated strongly connected subgraph. The data duplicating and the data refining phases are presented in sections 4.2.3 and 4.2.4, respectively. The data clustering phase follows the guidelines of the same phase of the algorithm of Canfora et al., presented in section 2.5.4. At each iterative step information is extracted from the program representations (section 4.2.1), which are updated in order to conform the changes made by the algorithm.

The aim of the fifth chapter is to show how the method can be adapted to the peculiarity of a given programming language such as COBOL, while respecting in the meantime the main ideas of the technique. By way of a case study, a simple COBOL program has being analysed, and the technique being used on it. All the necessary arrangements to adapt it to the peculiarity of COBOL are underlined throughout this section.

In the last two chapters, the future works and the conclusions of this work are presented.
Chapter 2

A Review of the Existing Techniques

2.1 Introduction

This chapter reviews existing reverse engineering and reengineering techniques for legacy information systems.

Reverse Engineering

The classical definition of reverse engineering describes it as "the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [CC90]. The reverse engineering is an important part of any software maintenance process aimed to improve understanding of the software system and its structure.

The reverse engineering process [S95], shown in figure 2.1, is usually part of the software reengineering process.

Reengineering

The process of reengineering, also called renovation or reclamation, is often associated with a business process reengineering [H90]. It is "the examination and alteration of a subject system to reconstitute it in a new form
and the subsequent implementation of the new form" [CC90] The process usually, not only recover design information from existing software, but also involves simple changes to transform the existing unstructured constructs into structured ones which are more understandable and more maintainable [MS87] [MD91]. In most cases, reengineering software re-implements the function of the existing system. But at the same time the software developer also adds new functions and/or improve overall performance [P94].

The mainstream of research in the field of reengineering of legacy information systems is enriched by the large number of studies that employ modularisation, i.e. the replacement of a large monolithic program into a functionally equivalent collection of smaller modules By now it is common opinion that modularisation is also important for downloading purposes [NS95], i.e. for the transition from monolithic, single-processor mainframe system to distributed, multiprocessor client/server (C/S) environment. An organised download, taking care of semantic and functional content of modules, achieves an efficient distribution of modules in both client and server machines.

The term "module" is used by several authors to denote different programming constructs. Originally, the term was applied to routines, but after the work of Parnas [P72], the term has been used to denote a clustering construct generally providing [C89].

---

1 The business-reengineering processes are aimed to a global reanalysis and redesign of the business process in order to reduce costs and improve quality. They cannot be considered either maintenance processes or software evolution, since they do not change requirements.
• "abstraction mechanism" offering a perspective of that clustered entity at a quite high level,
• "protection mechanism" as a control of the visibility, helping to restrict the affects of a change to a system.

If the interest in modularisation of legacy information systems is in its migration as well as its downloading, the actual trend is to decompose the system into independently compilable module units that are agent-like, communicating via message passing, and providing information hiding. Specifically, if the target system should be fully object oriented, other object oriented key features such as polymorphism and inheritance are provided, often after a targeted business reengineering process.

An efficient modularisation might be achieved by exploiting the requested object oriented features present in the earlier structured programming techniques [GK95] [KN95] [S96] [CDDF97]. In fact, even though a program is written in a conventional programming language that does not directly support object oriented programming constructs, it can contain collections of routines (functions or procedures), types and/or data items that can be isolated. In the target system, the collection of types and/or data items can store the state of an object and the collections of routines (functions or procedures) that get and/or update the state can implement the object's methods [CCM94] [JL94].

Sometime, if the programming language does not have the required object oriented features, they are simply simulated by respecting some standards of programming. An example is COBOL, in which all the data is global. Data can be accessed - and altered - at any position within the program, thus making it hard to ensure information hiding. The ANSI COBOL Committee produced a documents - afterwards enhanced by Yourdon [Y80], Microfocus et al - to propose a standard that an object oriented COBOL application should have. Among the other directives, the standard establish the guidelines of the structure of the target system guaranteeing information hiding. In order to ensure

and specification, then they do not modify the functionalities
information hiding to the system, the program should be divided up into classes corresponding to the objects processed. Each class is an abstract data type encapsulating the attributes of, and the action on, the objects enclosed. Each object is enclosed within a compilable unit. Each class communicates with an external classes only through message passing by invoking it in a CALL statement with a list of parameters. The messages are declared in a separate import/export area. Only the classes subordinate by right of inheritance can be invoked without parameters, as they can access the data of the invoking class.

2.2 Reverse Engineering Methods

There are in literature several methods to reverse engineer existing legacy information systems. They are mainly aimed at reducing the effort in migrating them from one environment to another.

In the following, three methods dealing with COBOL legacy information systems are described. The RECAST method is purely a reverse engineering technique. In the REDO method the reverse engineering techniques are used to transform COBOL programs into object oriented specifications in language “Z.” The REORG method is a reengineering method, but the modification of the source code is only performed in the last of ten steps. Thus the body of the REORG method might be considered the set of reverse engineering activities producing documentation and allowing the subsequent modification of the code.

2.2.1 RECAST

RECAST was developed as the principal product of a joint project sponsored by Information Engineering Directorate of the DTI and the Science and Engineering Research Counsel (SERC) under the Information Engineering Advanced Technology programme.

The RECAST (Reverse Engineering into CASE Technology) method is aimed
at reverse engineering existing COBOL legacy information systems into SSADM\(^2\) [CCTA] logical system specification, in order to reduce the cost of and to improve the maintenance process.

Several representations of the system at different points of view are derived through the use of a series of informal transformations. The recovered and documented system design would then provide a path into some system development methods (via the use of CASE tools) thus assisting the development of a modified or replacement system. SSADM has become the \textit{de facto} standard of system analysis and design in the UK [EM93].

The structural model for RECAST can be divided into four stages.

1. \textbf{Identification of business users' view (BUV)}.
   The outputs of this stage are the information about how the user perceive the function performed by the system. It needs direct information from the users in order to identify the business functions as well as events and enquires processed by the system. To complete the business users' view, the screens and the menus are analysed on-line and documented in SSADM notation.

2. \textbf{Identification of logical data model (LDM)}
   This stage defines the rules for analysing the data occurring in the files of the system. In order to extract a \textit{physical data scheme}, an increasing detailed analysis is performed upon the files of the system. The files potentially containing entities, the transitory files and the report files are then derived. At this stage, a document called \textit{system network diagram} is produced. The resulting \textit{physical data scheme} is detailed with the analysis of the data structure of the individual COBOL modules, COPY libraries and on data dictionary system. If the system accesses a database IDMSX (ICL's proprietary database), the rules for dealing with schemes are defined. This allows the enhancement of the \textit{physical data model} with the entities, their attributes and the mutual relationships. This step also produces a catalogue

\(^2\) SSADM (Structured System Analysis and Design Methods) [CCTA] is a system analysis and design method based on a set of complementary techniques of code representations. It is owned by the UK government agency Government Centre for Information Systems
of all the synonyms and homonyms at file, record, group and field level. This information is used to determine whether a part of the system affects the data model or simply generates reports. A phase of relational data analysis can be performed both to understand the meaning of repetitive groups and to check if the logical data model and the user's view are compatible. If the system is to be re-implemented with a different data design, then the logical data model should be abstracted.

3. Identification of the system processing (SP).
In this stage the functions - "sets of system processing which the user wish to schedule together to support their business activity" [CCTA] - with their component event/enquiry are identified. The process network diagram provides information about the relationships between files, modules and parameters. Menu hierarchies are examined in order to isolate module dependencies. The COBOL modules are examined to extract the sub-programs called. This information is detailed the process network diagram. At this stage, slicing techniques are used to abstract the processing within the sub-system. This supports both the business activities and restructures the functions into logical sub-system.

4. Identification of the menus and dialogues (MD).
In this stage, the element of on-line processing, if any, are treated. Those on-line processing are specific of the ICL TPMS (Transaction Processing Management System) transaction processor.

The RECAST procedural model generates a set of Intermediate Documents (IDs) containing all the elements of system design. This documentation is in a form that is that is appropriate for use in a CASE environment.

2.2.2 REDO

In this section, the European collaborative project ESPRIT II "REDO" (no. 2487) is presented. It covers activities from several areas such as:

- reverse engineering: redocumentation and reengineering;
• **validation**: post-hoc verification and generation of correct code from specification;
• **maintenance**: new languages and methods aimed to support.

It is aimed to transform a COBOL batch program without database accesses or special communication interfaces into a formal object oriented specification using the language "Z". The formal specification notation Z is a specification language based on mathematical set theory and logic. It has been developed at the Programming Research Group (PRG) of Oxford University for use in the specification of state-based programs, and has now matured into a valuable and widely used by industry as part of the software (and hardware) development process in both the UK and the US. Z has proved itself to be especially useful as a tool for formally verifying and demonstrating the correctness of safety critical and/or secure systems.

The REDO process, outlined in figure 2.2, involves three transformation levels, as explained below.

1. **Stage 1: Translation from COBOL to UNIFORM.**
   In the first level, the COBOL program is represented in the intermediate meta-language UNIFORM. In this phase, redundant constructs are eliminated. This program representation can be used for reengineering and reverse engineering purposes [CMW89]. It allows to produce technical documents like data flow diagrams, entity/relationship diagrams and other [BL91]. In this phase, the relation among data are analysed and translated into logical *invariants* of the program. The semantic equivalence is guaranteed by addiction of information such as the initial value of variables, being whether generated or stored.

2. **Stage 2: Higher Level Abstraction.**
   In the second level, the record types are defined as outlined objects, whereas the record fields are the objects' attribute. The DATA DIVISION is
partitioned into object classes and the **PROCEDURE DIVISION** cut up into slices based on data flow analysis. The code is split by grouping together the sequences of I/O operations on a particular file, and the intermediate statements affecting the contents of that file. At this stage, a process of restructuring eliminates all the **GOTO**s and other unstructured code constructs.

3 **Stage 3: Simplification of Abstraction and Design.**

In the third level, an object oriented specification from the intermediate representation is generated. The program slices are attached to the objects they refer to, becoming methods in a class. All statements which access the records embedded in the class, and all statements which alter or set attributes of that records are part of the generated method. In this final step, the UNIFORM syntax is converted to a Z++ notation.

---

3 From the "Z FORUM mailing list" by Jonathan Bowen (PRG - Oxford) Contact <zforum-request@prg.oxford.ac.uk> with your name, address and e-mail address to join the mailing list
At the end of the process, there is a class specification for each file and the original procedurally structured statements are now distributed among the classes where they are attached to the object of processing [L90]

This project involve, in the second stage, a well defined phase of object identification. By analysing the data flow, the variables that are logically associated with the main data structures of the program are identified. The data flow among files, indexed arrays and reports is analysed, as these data items are considered as main variables, representing objects with their attributes. After having isolated the data structures and the attached variables, the global functions that updates and modifies a data structure are candidate to be an operation upon the considered class. Then a more detailed analysis upon that
operation is performed in order to make clear the meaning of that class within the application domain

Lano et al [HL91b] proved that the transformations on the program enable to rewrite it into a restructured form

2.2.3 REORG

The REORG approach [S92] has grown out of a reengineering experiment at the Union Bank of Switzerland [S91] aimed to reduce the costs and the risks of migrating a procedurally structured COBOL system into a object oriented environment, conforming to the latest CODASYL draft. It consists of 10 steps, as outlined in figure 2.3.

1. The first step performs a static analysis of the source code in order to produce many representations of the code in form of tables for data fields, for code blocks, constants and predicates. Other connectivity tables are created in order to represent data references, control flow path, program and data interfaces.

2. In the second step, a specification repository is populated with all the program description tables. The COBOL data records are converted in data trees and data dictionary entries. In this step, user and system interfaces are analysed and transformed into message format. Tables representing program/object and program/program relationships are created from information derived from data base, file accesses and sub-program calls. A Jackson type tree is created from the control flow structure.

3. The third step identifies the object types from the data structure. Several types of objects are created: WORK objects from the local data structure, FILE objects from the data record structure, VIEW objects from the database views; INTERFACE objects from the map and record structure, PARAMETERS objects from the linkage storage structure. The output is an object catalogue linking any type of object and its attributes.

4. The fourth step examines the accesses to the database files in order to...
recognise the relationship between objects via access sequences. The output at this step is a relationship table between the objects based on their access sequences.

5. In the fifth step the data item description is completed with information on the data usage. The references to a variable are collected from the tables produced in the previous steps and the data dictionary is updated with this information.

6. In the sixth step, the procedural instructions are coupled with the data elements they set or alter. If a statement accesses to several variables, then the statement is duplicated in order to update the information relative to all the data items.

7. In the seventh step, the data flow between objects is traced in order to identify those objects from which any one object derived data either directly or indirectly.

8. In the eighth step, the inheritance relationships are examined by marking all the attributes in the form of fields in the super-ordinate object from which values are inherited by a subordinate object. After this step, there are pointers from all inherited data items to the subordinate classes which require them.

9. The aim of the ninth step is the definition of the objects' interfaces via the construction of import/export messages. A message is the list of all the attributes required by an object from another object. The attributes to be passed as parameters are placed in the message - export for the sending message, receiving for the receiving object.

10. In the final step, the new code is written in object oriented COBOL from the intermediate design language.
The output of this method is a program structured in a hierarchy of classes, one for each extracted object. Each object using or passing data inherited from the super-ordinate class declares these data in the PUBLIC-STOREAGE SECTION. Local data are declared in the PRIVATE-STOREAGE SECTION. The PROCEDURE-DIVISION is partitioned into a series of methods. There are methods allowing to operate on encapsulated objects in order to perform operation as CREATE, DELETE, SELECT, UPDATE, STORE etc. There is also a section of attributes altered or set by an event.

Unfortunately, in a large system, it happens that the method has to be performed several time - one for each program unit within the system. This may lead to the production of many variants of the same class, since the same data object can appear in different programs, and the method creates the classes encapsulating all the attributes and the operation of a particular object. After the
application of the REORG method to each program unit a process of merging on
different variants of the same class must be performed by a software engineer
using application domain knowledge.

2.2.4 Alternative Approaches to Migrate Legacy Information Systems

An alternative to moving an existing software system from the native
environment is to encapsulate it in a wrapper [W95]. A wrapper⁴ is an
intermediate component, interacting with legacy components by message passing.
It is no more than a new object oriented part of the system composed by one or
more large objects whose methods are the menu options of the old system, with
the difference that they respond to the received messages.

It is very well known that small grain objects are more reusable than the big
ones. Unfortunately, most legacy information systems usually deal with
irreducibly large-grain objects. In these cases, a special class of wrappers might
be used, the object request brokers (OBRs) are specifically aimed to deal with
such a kind of coarse grain reuse. Sneed [S96a] individuates different levels for
software encapsulation, analysing each of them in a practical approach. The
OMG's CORBA (Common Object Request Broker Architecture) is a wrapping
technique allowing access to the legacy code left on a mainframe to provide
services to the clients on the peripheral. "CORBA is becoming a world wide
standard for accessing data and objects in a distributed computer network and
for exchanging messages between objects on different computers" [S96b].

In these types of approaches, the software engineer has to deal with tricky
data management problems. Sometimes, it is necessary to duplicate data or to
share data between the legacy and the new part of the system. In order to avoid
inconsistency, Graham [G94] describes four possible strategies.

1. The tandem or handshake strategy keeps a double copy of the shared data,
one in the old part of the system, and the other within the wrapper. Of course
the effort in keeping their integrity requires frequent operation of updates and
retrieves among the shared data. For this reason it is advisable only when the
amount of shared data is reasonably small

2 In the borrowing strategy, all the data remain in the old part of the system,
and the wrapper "borrows" (copies) part of them when it needs them. In this
case some further messages from the wrapper have to handle the data updates
within the old part of the system.

3 The take-over strategy simply copies the data into the wrapper, and each data
access involves messages to and from the wrapper, thus increasing the
complexity enormously.

4 The most promising way to deal with problems of inconsistency and of
efficiency is the translation strategy. Of course, it requires a bigger effort in
translating the original design of the legacy information system to an object
oriented model. The application of this method is favourite if the legacy
information system has been developed with a technique such as stepwise
refinements around a critical data structure, because all the these structures
and the programs using them will naturally migrate to the objects of the new
system.

The data-centred translation is a refinement of the translation strategy. It
uses an approach based on the accesses types, by reverse engineering the data
model, thus allowing the creation of a CRUD (Create, Read, Update, Delete)
matrix to organise the legacy information system around the data structure.

2.3 Processes of Reuse Reengineering

For all the considerations above, in the last few years there has been an
increasing interest in the processes aimed to redefine the organisation of existing
systems (even if they are not legacy yet) in order to use powerful theories,
techniques and technologies both to design and implementing reusable software
components and to dispose repositories of "spare parts" elected from existing
software components. The main issue is the creation of a culture of reuse, considering the process of development of a new system as an activity of retrieval the appropriate software components in apposite repositories. This is basically the idea expressed by the "Full-Reuse Model" of Basili [B90].

In spite the large number of paradigms prescribing or directly promoting reuse, it is still difficult to find "catalogues of software components" that can be reassembled in the development of new systems or in the adding of new functionality to a maintained legacy information system. A pioneer approach in this direction had been made by Ada [ADA83], some software houses proposed directly in the industrial production environment Ada software components [B87b]. Few years ago [P94] there was also a project to create catalogues of software integrated circuits (software ICs) for object oriented languages.

2.3.1 The Paradigm of Reuse Reengineering (RE^2)

In a joint research project the "Dipartimento di Informatica e Sistemistica" [CMV95] [CV95] of University of Naples and the "Centre for Software Maintenance" of University of Durham [T94] [D95] defined a framework setting up all the activities concerned with the comprehension and reengineering of a legacy information system. The Reuse-Reengineering (RE^2) paradigm [CCM94] is mainly aimed to produce a set of reusable components from the existing source code in order to populate a repository of modules to be reused.

The RE^2 paradigm is articulated in five sequential phases, as displayed in figure 2.4, each of which is fully identified by the objects it produces. It identifies in the legacy information system a set of components each suitable to implement an abstraction of an entity in the application domain. The component is candidate to be transformed in a reusable software component. After each candidate has been transformed, so that its reuse is made easier, the repository is organised such in a way that the retrieval of suitable software components in the repository is made as easy as possible.
Figure 2.4 - The Reuse Reengineering (RE²) Paradigm identifies in legacy source code a set of software components each suitable to implement an abstraction of an entity in the application domain. Each component is candidate to be transformed in a reusable module. After that each candidate has been transformed such that its reuse is made possible, the repository is organised such in a way that the retrieval of suitable software components in the repository is made as easy as possible.

A RE² process can be outlined into five phases.

- The **candidature** phase receives as input the existing source code and produces as output a set of software components candidate to implement the abstraction of an entity of the application domain. This phase groups all those activities to analyse the source code and all those able to identify the software components representing an abstraction of an entity of the application domain.

- The **election** phase involves the activities that refine the candidate module producing the reusable modules. It de-couples, re-engineers and generalises the set of candidate objects received as input from the candidature phase. Usually this phase produces a further selection, not all the candidate modules are elected as reusable modules because of the complexity and costs of the applied reengineering techniques.

- The **qualification** phase is aimed to "qualify" the modules in the repository by adding all the information assisting their reuse. Usually, this phase contains a documentation phase defining a template allowing to represent the functional features of the module and how it might be reused.

- The **classification and storage** phase are a set of activities supporting the retrieval of a suitable reusable module by classifying it depending on a
reference taxonomy. In this phase the repository is organised and populated with the selected modules.

- The search and display phase groups together all the activities setting up a front end user interface to interact with the repository system. The aim is to simplify the user's work in navigate through the repository system with the help of the visual languages, for example.

The RE$^2$ project is mainly concerned with the first three phases in the paradigm, and does not address the last two. These later two phases are related to the setting up of the environment to support the reuse of the modules rather than to the extraction of these modules from old systems.

For each process of reverse engineering in the RE$^2$ paradigm there are different activities aimed at different goals. For example, the ones related to the definition of the global goals of the entire reengineering process, and the ones defining the requirements of tools, methods and methodologies related with the extraction of information from the source code and to the abstraction of this information.

Particularly, the distinction of each process is due to the activities defining the templates representing the information extracted through the analysis of the modules, and the activities related to the representation of the abstraction of the extracted information.

Candidature Phase

The activities in the first phase can be subdivided into three sub-phases, as shown in figure 2.5.

![Diagram](image.png)

Figure 2.5 - The distinction of the activities in the Candidature phase leads to subdivide it in the sub-phases of "Candidature Criterion", "Reverse Engineering" and "Criterion Application".
In the first sub-phase, after having sketched the aim of the reengineering process by defining a template of the abstraction features of the objects to be searched in the existing code, the more suitable form of program representation to support the research of the defined abstraction has to be differentiate. This phase, also defines the algorithms acting on that program representation, able to identify the software components realising these abstractions. Those algorithms, called **candidature criterion**, give their name to the whole phase.

The **reverse engineering** sub-phase performs a reverse engineering process in order to extract a set of software components from code and makes up an instance of the model defined in the previous sub-phase.

The **criterion application** phase applies the candidature criterion to an instance of the model, thus producing the set of software components that can be candidate for reuse. Note that the proposed software components are not yet reusable modules.

The candidature phase also includes some reengineering activities manipulating the level of functional abstraction of the modules. Particularly, interventions realising the decoupling of the components from the environment are typical of this phase. Typical operations of this intervention are the removal of any reference to global variables and, if there is sharing of code, the activation and using of external software components and the mechanism of code protection from undesir'ed accesses. As these processes produce modules at an higher level of abstraction, they can be classified as **generalisation** process.

Before the election phase a **concept assignment** process [BMW94] is performed in order to select the subset of modules matching with the entities of the application domain.

**Election Phase**

The activities of the election phase can be organised into three sub-phases, as shown in figure 2.6.
The **template definition** sub-phase draws a general template of a module in order to reengineer the reuse candidate module. Each candidate module should match as much as possible with the template. It is based on key features such as information hiding and all the other object-oriented features of the used programming language. In general, the template gives the resources visibility that can be exported from the module, but should have a protection mechanism against the access to the non-exportable resources.

In the **decoupling** sub-phase, reverse engineering operations decouples the software components from the external environment, i.e., to split the connection with the old system's components that do not belong to the same reuse-candidate module. Only at this stage, the **clustering** sub-phase organises the clustering of the software components depending on the defined template, i.e., for producing the reusable module.

The election phase also includes some **generalisation** processes aimed to increase the generality of functionality implemented by the reuse-candidate module by transforming the type of this functionality to a type that the user can instance when the module is reused.

At this stage, an analysis is performed concerning the effort and the costs of the reengineering process aimed to decouple and to cluster the candidate sets. This analysis, with the **concept assignment** process executed in the candidature phase, might be the basis of the validation of the candidature criterion [CFM93].

**Qualification Phase**

The activities into the qualification phase can be subdivided into the
subphases shown in figure 2.7.

![Figure 2.7 - The distinction of the activities into the qualification phase leads to the subdivision of the phase in the sub-phases "Specification Model", "Functional Reverse Engineering", "Testing and Specification Fixing"]

The specification model sub-phase defines a representation formalism to represent the functionality of each module, and its possible use. The second sub-phase realises a functional reverse engineering process to extract the specification coherently with the formalism defined. The complexity of the entire qualification phase mainly depends on the complexity of the reverse engineering techniques needed to define the functional and interface specification formalism. In the final subphase, a functional testing is performed, and the specifications are fixed within the defined template. The documents produced in the candidature phase and the documentation of the legacy information system, if existing, can help in the qualification phase to reverse engineering the reusable modules to produce their specification.

The RE² paradigm includes in it an unique model both of the production of new systems and the maintenance and the evolution of existing systems. Each maintenance or evolution intervention is mainly the retrieval in the repository of those components that (whether directly or with a small effort) can substitute the components to be maintained or can add to the functionality required by evolution. Developing a new system is reduced to retrieving software components suitable to the requirements of the new system within the repository and to their subsequent assembling.
2.4 Classes of Candidature Criteria to Decompose Legacy Information Systems

A great amount of work has been carried out around the RE\textsuperscript{2} paradigm prominently as regards the candidature criterion within the candidature phase. A successful RE\textsuperscript{2} process has to look for abstractions implemented in the legacy information system. The abstractions can be of different natures depending on the focusing of the system on algorithms, or on data structure, or on control structure [CCM94]. Obviously, the candidature criterion must be tailored according to the type of abstraction one is looking for as the type of abstraction to be singled out deeply affects the reverse engineering process needed to produce the model to apply the criterion and the definition of the candidature criterion in itself.

Functional Abstraction

The high-level languages present primitives (procedures or functions) implementing functional abstraction, i.e., procedure-like software components focusing on the algorithms. A notable example of search of functional abstraction is that from Page-Jones [P80]. It takes as input a program written in a procedure oriented language and uses the information-cluster by determining which routines require the use of common data and then refines the routines around that data.

The search for functional abstraction can be conducted on those components at different abstraction levels. It is clear that the operation of isolating reusable modules within code that was not designed for the reuse—often this discipline is called "Software Scavenging"—presents difficulties concerning the quality of the isolated objects. Sometimes some operations manipulating the abstraction level of the software components implementing functional abstractions help to achieve higher quality candidate modules. The operations are isolation, aggregation, and generalisation.
The operation of **isolation** consists of the decomposition of a software component implementing more than one abstraction into several module each implementing only one abstraction, i.e. one algorithm performing only one function, thus making possible to reuse it. The search for functional abstraction can lead to either vertical or horizontal isolation. An example of vertical isolation is that of program slicing [W82] [W84], as each one of the isolated pieces of code is a set of statements that lie on the same dynamic path of the components. On the other hand, approaches such as the *primes* [FW86] [FK87] search for functional abstraction horizontally, i.e. each one of the pieces of code isolated is a block of the component's text.

The isolation by *primes* assumes that the program is structured. In this case, a reverse engineering process in the candidature phase should produce the *nesting tree* [CD91A], i.e. a tree showing the nesting relationships among the *primes* (the sub-trees). A *nesting tree* can easily represent a structured procedure-like component. The isolation can be performed on the *primes* by searching the *primes* each of which represent a function. In order to recognise a function in a *nesting tree*, an analysis on the data flow to and from the sub-tree is performed.

The operation of **aggregation** groups and links the components implementing different subpart of an abstraction at a higher level. It guarantees that the candidate modules are low-coupling and implement abstraction at the highest level is possible, thus providing high functional cohesion [CY79] among the modules candidate to be a reusable module. An analysis on calls and the inter-procedural data flow is required, thus showing the types of the relationships among the components.

In literature, most of the approaches develop a search of functional abstraction by aggregating the components by representing the program as a directed graph with the decision as nodes and the branches as edges. Using this representation then a large number of classical graph theory works can be used.

---

5. Patna [P72e] defined an information-cluster as a set of routines that have exclusive right of access to a particular data item or set of data items.
The first work focusing on control flow analysis was by Waters [W88], Mulleı et al [M90] and Bush [B85]. They split complex graphs into sub-graphs by finding the points of minimum interconnection. Colbrook [C90] has proposed another approach focusing on data flow analysis, and this has been enhanced by Lano et al [BHL93] and Kozaczynski [EKN91]. Many works refer to the creation and manipulation of a *structure chart* derived from a call graph [CDM90] [H77] [CCD91] [CC92]. The most elementary example is the search within a call graph of notable sub-graph, as strongly connected sub-graph, trees, one-in/one-out sub-graph. Other reverse engineering processes transform a call graph into a tree, by collapsing some highly connected sub-graph into a single node [BCD92].

In the section 2.5 several example of these approaches will be presented.

The operation of *generalisation* is to make the software components at a higher abstraction level, thus increasing their reusability. A classical example of generalisation is the parameterisation of some values in order to allow the user to instance a module before reusing it. An example of this low level parameterisation is the parameterisation of the length of an array in a module using it. Higher level form of generalisation generalises the type of information that a function handles in order to have such a type of generalisation, the procedure-like component implementing the generic function is recorded as a skeleton. The designer has to instance it to the required type in order to reuse the component.

**Data Abstraction**

The abstractions essentially referring to data structures and data types are classified as *data abstraction*. The candidature criteria to search for these abstractions can focus both on the data structures or types and on their generic versions, thus leading to four directions for developing candidature criteria searching for data abstraction:

- The *Data Structure Candidature* searches those sets of data items and procedure-like components implementing a data structure. Some authors call it an object. The data items belonging to types built into the language,
implement the internal state of the object that can only be accessed by calling the procedure-like components.

- The *Generic Data Structure Candidature* searches those sets of data items and procedure-like components implementing an object to be possibly generalised. This is the case of a structured object whose access operation do not depend on the type of the components and, thus it is possible to choose the type of the components in a finite set of types.

- The *Abstract Data Type Candidature* searches in the software system a set of software components (data items, user-defined data types, procedure-like components, etc.) implementing an abstract data type⁶. Some authors call it a class [C89]. An instance of the class is a set of parameters representing a data on which the services of the abstract data types are allowed. An abstract data type must allow a designer to declare several objects and access them by calling the procedure-like components. Sneed [S94] claims that the decomposition technique by abstract data types are the most difficult of all remodularisation approaches and that "it is practically impossible without tools".

- The *Abstract Data Structure Candidature* searches in the software system for a set of software components that implements an abstract data type that can be possibly generalised.

For traditional languages the reverse engineering activities in the candidature phase produces reuse candidate modules implementing an object or a class. A further process of generalisation can obtain a general object or an abstract data type. The reverse engineering process to generalise a class from an object extracted from existing code cannot be fully automated [CCM94], as the knowledge of an domain expert software engineer is required to recognise the

---

⁶ With abstract data types as defined by Dahl and Hoare [DH72], a programmer can consider a type as the set of all the operation that are applicable to variable of that type. Early work on the use of abstract data type approach in designing modular program in forward engineering was done by Parnas [P72].
links between the entities in the application domain with the code components

A very interesting survey on existing type theories is given by Danforth and Tomlinson [DT88]. The authors explore the way in which these theories are able to represent the objects and their interaction.

Control Abstraction

The abstractions referring to politics are classified as control abstraction. It meets the needs for co-ordinating concurrent processes and implementing the techniques to manage shared resources. The RE³ project does not address control abstraction because it deals with existing software written in traditional languages that usually do not express concurrency explicitly, but manage it through calls to the services of a system kernel [NS87].

A good overview on the control abstraction methods actually in use is due to Poulin and Tracz [PT94].

2.5 Graph Theory Approaches

Usually, much of the effort in identifying objects in traditional languages promote reuse by defining a candidature criterion whose reverse engineering activities search for data abstraction within the legacy code. The derivation of a module implementing an abstract data type usually works similarly if it is not just the same technique that, after having extracted the objects applies a further process of generalisation in order to obtain the abstract data type.

Both of these techniques makes use of reverse engineering approaches based on representing the program as a graph, thus gaining from a great amount of knowledge regarding the existing classical theory on graph.

The aim of this section is not to provide an exhaustive overview on all the existing methods proposed as candidature criteria for the reuse reengineering method but only those approaches to extract objects from a legacy information.
system that relate directly to the new technique presented later in this thesis. All
the work presented propose candidate criteria searching to isolate meaningful
modules by using different features of graphs.

2.5.1 The Call Dependencies of Cimitile and Visaggio

The technique defined by Cimitile and Visaggio [CV95] transforms a Call
Directed Graph CDG into a dominance tree [H77], and then analyses the
modified program representation in order to interpret the dominance relationships
of this graph as functional dependency relationships.

The technique uses the program's Call Directed Graph CDG = (N, E). In a
CDG, N = PP is the set of all the procedures and functions. The main program is
denoted by s, and obviously \( \{s\} \in PP \). The relation E is the Cartesian relation
PP x (PP - \{s\}), showing the presence of an activating statement within procedures.
A direct consequence of recursion between the procedures of the program is the
presence of strongly connected sub-graphs in the CDG. In this case, all the
sub-graphs containing at least one cycle involving all of its nodes can be
collapsed into a single node. As a result, the CDG turns into a Call Directed
Acyclic Graph CDAG.

According to Hetch [H77], a procedure \( p_x \) in a CDAG dominates a procedure
\( p_y \) if and only if each path from s to \( p_y \) contains \( p_x \). A procedure \( p_x \) directly
dominates a procedure \( p_y \) if and only if \( p_x \) dominates \( p_y \) and all the procedure
dominating \( p_y \) dominate \( p_x \), too. A procedure \( p_x \) strongly and directly
dominates a procedure \( p_y \) if and only if and only if \( p_x \) directly dominates \( p_y \) and
\( p_x \) is the only procedure calling \( p_y \).

The reflexive and transitive closure of the dominance relation on the CDAG
is the direct dominance relation, representing by a tree called the Direct
Dominance Tree DDT, whose root is the main procedure s. The Strong and
Direct Dominance Tree SDDT is obtained from the DDT by marking all the edges
representing the strong and direct dominance relationship. The set of sub-trees

---

variable name or for casual comments within the code [B89]
of a SDDT can be divided into two subsets: the subset MET of the sub-trees containing only marked edges and the subset UMET of the sub-trees containing at least an unmarked edge. The Reduction of the Strong Direct Dominance Tree RSDDT is a tree obtained from the SDDT by collapsing each sub-tree in MET into a unique node.

Four rules have been proposed to aggregate procedures into reuse-candidate modules and to identify the uses and is_compose_of relationships [JGM91] between them:

1. The set of procedures represented by a strongly connected sub-graph of a CDG is a candidate to constitute a reusable module. The programs units associated with the modules is extracted to constitute a candidate module for reuse.

2. By examining a SDDT, the set of procedures represented by the nodes of a sub-tree \( t \in \text{MET} \) is a candidate to constitute a reusable module represented by the root of \( t \).

3. The set of procedures represented by nodes of a sub-tree \( t \in \text{UMET} \) within a SDDT linked to the root of \( t \) by a marked edge is a candidate to constitute a reusable module. This module is related with a uses relation to the modules represented by the nodes in \( t \) which are linked to the root by an unmarked edge.

4. Each of the marked edges of a RSDDT is a candidate to constitute an is_compose_of relationship between the modules represented by the that the edge links, while an unmarked edge represents an uses relationship.

The dominance tree can be used as a basis of a method to search for functional abstractions in legacy information systems written in procedural languages and designed using modularity and the functional decomposition. In order to be modular, a system must be segmented in a hierarchy of code segments - corresponding to the elementary operation of the program - each with a single entry and a single exit. The modularity can be obtained by targeting restructuring intervention by the software engineer Cimitile et al. [CFM93]
confirmed the validity of the dominance criterion by experimenting it both in Pascal and in COBOL [CDDF94] environment

2.5.2 The Algorithms of Liu and Wilde

Liu and Wilde proposed two algorithms [LW90] [LOWY94] based on an analysis of global data and data types. The aim of the method is the retrieval of candidate objects \( O = (F, D, T) \), where \( F \) is the set of all the program units, \( T \) and \( D \) are the sets of the data types and data items, respectively. Each of the sets can be empty. The algorithm based on global data is dwindled into three steps as listed below:

Step 1 Definition, for each global variable \( x \) of the set \( P(x) \) of the routines directly referring \( x \)

Step 2 Supposing that each \( P(x) \) is a node in a graph, a graph \( G = (V, E) \) is then constructed in which \( V \) is the set containing the defined \( P(x) \) and an edge between two nodes \( P(x_1) \) and \( P(x_2) \) denotes that the sets \( P(x_1) \) and \( P(x_2) \) are not disjoint, i.e., \( P(x_1) \cap P(x_2) \neq \emptyset \). Formally:

\[
V = \{ P(x) \mid x \text{ is shared by at least two routines} \}
\]

\[
E = \{ (P(x_1), P(x_2)) \mid P(x_1) \cap P(x_2) \neq \emptyset \}
\]

Step 3 If strongly connected sub-graphs can be recognised in the graph as defined above, then they are regarded as candidate objects. Each of them is composed of those units and relative global variables. Formally, by denoting a strongly connected component with \( C = (V_c, E_c) \), the objects extracted from it can be represented as a tuple \( (F, T, D) \), where:

\[
F = \bigoplus_{r \in C} P(x)
\]

\[
T = \emptyset
\]

\[
D = \bigoplus_{r \in C} \{ x \}
\]

In this case, the role of the reverse engineering technique is to set up the instance of the module to apply the candidature criterion by producing the set \( P(x) \) and the above-defined graph. This criterion has been applied with
significant results to conventional programming languages such as C, Ada, COBOL and Fortran, showing that the production of both the set \( P(x) \) and the graph can be totally automated. Unfortunately, "this method in many cases can produce objects that are too big" [LW90] and there is the necessity of the software engineer's intervention in order to resolve conflicts and provide knowledge about the application domain to improve the candidate objects.

In order to obtain a candidate of a size more suitable to fit the aims of the reuse reengineering techniques, slicing techniques can be used to search for the set of slices \( SP(x) \) by using each global variable \( x \) to define the slicing criteria for \( SP(x) \).

The second algorithm from Liu and Wilde [LW90] is aimed to candidate a module implementing the abstraction of an abstract data types. This algorithm deals with ordered relationships among the user-defined types. The user-defined type \( t_1 \) is assumed to be a sub-type of \( t_2 \), denoted by \( t_1 \preceq t_2 \), if \( t_1 \) is used to define \( t_2 \) - in this case \( t_2 \) is a super-type of \( t_1 \). Obviously, if \( t_1 \preceq t_2 \), and \( t_2 \preceq t_3 \), then \( t_1 \preceq t_3 \).

Once the set of the user-defined types has been ordered, the method exploits the classical work on graph theory by representing the program as a bipartite graph\(^8\) as a couple \( G=(N, E) \), where the set of nodes \( N \) is partitioned in two subsets, \( N_1 \) and \( N_2 \) denoting the procedure-like components and the user-defined types, respectively, and the set of edges \( E \) contains edges from a procedure-like component \( c \) to a type \( t \), thus allowing to represent the relationships among user-defined types by showing how types are used to declare formal parameters of the procedure-like components.

This graph is then simplified by eliminating the edges \((c, t)\) for which an user-defined type \( t_1 \) exists such that \( t \preceq t_1 \) and \((c, t_1)\) is an edge in the graph. Each one of the connected sub-graphs, possibly recognised in the above graph, defines a candidate to create a reusable module implementing a class.

Figure 2.8 shows an example of a bipartite graph.

\(^8\) They belong to the family of interconnection graph as defined by Calliss [C89,\( A\).]
Chapter 2  
A Review of Existing Techniques

2.5.3 The Algorithm of Dunn and Knight

The algorithm presented by Dunn and Knight [DK93] exploits expert systems to isolate the reusable modules. The system is the interaction of three functional elements, a C parser that generates the abstract syntax tree from C source code, a Prolog interpreter identifying the candidate components for reuse and an interactive interface allowing the communication within the system. The expert system uses a knowledge base containing knowledge about the application domain and the design of the software that can be examined, knowledge about the target domain, metric definition and reengineering knowledge.

The heart of this system is the phase performed by the Prolog interpreter. By the analysis of the call-graph, it searches the reusable components among those invoked more than once. Other reusable modules are candidate among the strongly connected components by analysing various kinds of coupling, when

- there is **data coupling** when the program components share formal parameters, or, more generally, simple data,
- there is **common coupling** when the program components share global data,
- there is **external coupling** when the program components share external data,
- there is **common coupling** when the program components share data used for control.
The analysis of these forms of coupling between the program components leads to the candidate for the reuse being those components not connected. The components loosely bound present high degree of reusability, as they do not depend on other function or local data within the program. A critical point is to determinate how the restriction of the above coupling characteristic can be relaxed such that the sets of program components with varying coupling degree can be identified as a candidate for reuse.

Another method to isolate software components as a candidate for possible reuse identifies those routines and function and the global data items that can be grouped to form an abstract data type. To this aim the method uses a bipartite graph, where the set of nodes \( N \) is partitioned in two subsets, \( N_1 \) and \( N_2 \), representing routines to global variables and the set of edges \( E \) contains edges specifying the "uses" relations of the global data within the routines.

The algorithm performs a depth-first traverse of the graph looking for strongly connected components; each component is regarded as a candidate object.

An evaluation of the use of this expert system on 5 public-domain software systems written in C language was done with satisfactory results. The evaluation criteria involved:

- *practicality* (how useful a part would be in an application either in the same application domain or in other);
- *reusability* (how much effort is necessary to reengineer a part in order for it to be reasonable to be candidate for the reuse);
- *understandability* (how difficult it is to comprehend what a reusable candidate does).

### 2.5.4 The Algorithm of Canfora et al

The algorithm presented by Canfora et al [CCM96] improves on the previous ones of Liu and Wilde [LW90] and Dunn and Knight [DK93], and enables the identification of objects within a *legacy information system* with less
As in the previous methods, the method proposed by Canfora et al represents the program as a bipartite graph where the two sets of nodes $N_1$ and $N_2$ represent procedures and global data, respectively, and each edge represents the reference of a data item within a procedure. For each node $n \in N$, the sets $\text{PreSet}(n)$ and $\text{PostSet}(n)$ are defined as

$$\text{PreSet}(n) = \{ y | y \in N_{\text{N}} \wedge (y,n) \in E \},$$

$$\text{PostSet}(n) = \{ y | y \in N_{\text{N}} \wedge (n,y) \in E \}.$$

Informally, the set $\text{PreSet}(n_2)$, where $n_2$ is a node in $N_2$ (i.e., a data item), represents the set of all the procedures (nodes $n_1$ in the set $N_1$) referencing $n_2$. The set $\text{PostSet}(n_1)$, where $n_1$ is a node in $N_1$ (i.e., a procedure), represents the set of all the data items (nodes $n_2$ in the set $N_2$) referenced by $n_1$. Note that, coherently, for each $n_1 \in N_1$ the set $\text{PreSet}(n_1) = \emptyset$, and for each $n_2 \in N_2$ the set $\text{PostSet}(n_2) = \emptyset$.

The bipartite graph representing the relationships between data and procedures within the program establishes when a sub-graph has a strong degree of connectivity, thus representing the routines and data they access likely having the behaviour of an object in the application domain. Within this method, an iterative algorithm based on some indexes measuring the variation of internal connectivity of the graph resulting in the use of $P$ to generate a new cluster is presented. At each step of the iterative algorithm, the procedure $P$ associated with an index “sufficiently high” is used to cluster $P$, all data it accesses, and all the procedures accessing a subset of that data.

Unfortunately, there usually exist procedures referencing data items of different objects, thus creating a link between the corresponding sub-graphs. The connections originated by this undesired links are of two types: coincidental and spurious. The coincidental connections are defined as the result of routines implementing more than one functionality, each of them logically belonging to

---

9 The “internal connectivity” of a subgraph is expressed by the ratio between the number of internal edges into the subgraph and the number of edges with only one vertex in the subgraph.
different objects. A procedure generating coincidental connections can be split on the basis of the groups of related data they refer to. The spurious connections are created by procedures accessing the supporting data structure of more than one object in order to implement system specific operations.

Both the coincidental and the spurious connections make difficult the identification of strongly connected sub-graphs and thus the isolation of different objects. The algorithm of Canfora et al [CCM96] partially overcomes the problem of identifying the undesired links by computing some indexes IC(P) and ΔIC(P), for each procedure P, in each iterative step. The IC(P) index defines the internal connectivity of the sub-graph generated by clustering together all the data items P accesses and all the procedures only accessing a part of these data items

\[
IC(P) = \frac{\sum_{A \in \text{PostSet}(P)} \# \{P_i | P_i \in \text{PreSet}(A) \land \text{PostSet}(P_i) \subseteq \text{PostSet}(P)\}}{\sum_{A \in \text{PostSet}(P)} \# \{\text{PreSet}(A)\}}
\]

The index ΔIC(P) measures the variation of internal connectivity of the bipartite graph resulting in the use of P to generate a new cluster

\[
\Delta IC(P) = IC(P) - \sum_{A \in \text{PostSet}(P)} \frac{\# \{P_i | \text{PostSet}(P_i) = \{A\}\}}{\# \{\text{PreSet}(A)\}}
\]

The routines having an index ΔIC(P) sufficiently high are used to generate cluster around P.\(^{10}\)

If the value of the index relative to P is lower than a value chosen as a threshold value, then P is considered to introduce a coincidental or spurious connection, and it is sliced or deleted, according to the objective of the reengineering process.

The algorithm of Canfora et al [CCM96] is given in figure 2.9 below.

\(^{10}\) As defined by Cimitile et al in [CDDF97]
WHILE THE GRAPH IS NOT IN THE FORM OF A SET OF ISOLATED SUB-GRAPHS DO
FOR EACH NODE \( P \in \text{PROC} \) DO
    COMPUTE INDEXES \( I_C(P) \) AND \( \Delta I_C(P) \)
END FOR
COMPUTE THE STEP VALUE \( STV \), AND THE SETS MERGE AND SLICE
MERGE = \{ \( P \mid \Delta I_C(P) > STV \) \}
SLICE = \{ \( P \mid 0 < \Delta I_C(P) \leq STV \) \}
INTERACTS WITH HUMAN EXPERIS TO DELETE FUNCTIONS FROM THE GRAPH
AND/OR TO MOVE FUNCTION FROM THE MERGE TO SLICE AND VICE-VERS
FOR EACH FUNCTION \( P \in \text{MERGE} \) DO
    CLUSTER THE SUB-GRAPH IDENTIFIED BY \( P \) INTO A SINGLE NODE AND UPDATE THE GRAPH
END FOR
END WHILE

Figure 2.9 - Algorithm presented in [CCM96]

The algorithm terminates when the graph is transformed into a set of strongly connected sub-graphs. Each sub-graph is composed of some data representing the attributes of an object, and thus its state, and some procedures representing the methods of that object.

The treatment of the spurious connections depends on the objective of the reengineering process. If the aim is the migration of the legacy system, then no procedures can be deleted in order to not modify the functionality of the information system. When the aim is populating a repository of reusable components, the routines accessing the supporting data structure of more than one object are simply deleted, as their slicing produces methods of low quality.\(^{12}\) The redevelopment of the information system, or the development from scratch of a new information system in that application domain is thus supported by the reuse of the software components extracted from the repository, and by the extraction of knowledge about the real world entities in that application domain. The extraction of knowledge supports the phase of designing of the new information system.

---

\(^{11}\) Canfora et al also suggest the use of a statistical filtering function to calculate a step value up to which the variation in the internal connectivity can be considered noise and the related routines have to be supposed to introduce noise connections.

\(^{12}\) Moreover, the evaluation on when slicing or deleting a procedure also depends on the system knowledge. This step of human intervention can by supported by an analysis of code and documentation.
2.6 Summary

Even if academic institutions and work environment agree that software maintenance improves both productivity and quality in the development of new software projects and in maintenance of existing system, at the current state of the art there is still an inhibiting difficulty in acquisition of the reusable components. This is the main cause that prevent a spread diffusion of reuse concept in the working environment [CCM94], [B87].

In particular, it is well known that the examination of legacy code can support the population of a repository of reusable software components in a cheaper way and in a shorter time [AF92] [DK93] [P91] than to develop them ex-novo. As for as the building up of this repository, the Software Productivity Consortium [SPC93] defines the concepts of domain engineering as a process to develop a repository of reusable components for a given application domain, and application engineering, as a process for the automatic assembling of the reusable modules on the basis of the customer requirements.\footnote{These two concepts are defined similarly in megaprogramming [CWW]}

This chapter has focused on well known methods to reverse engineer legacy information systems in order to help any intervention aimed to an evolution of the code, also allowing the extraction reusable modules from the source code. Note that the repository should be populated not only of modules as a fragment of code abstracting entities - or a method of complex entities - in the application domain, but of software components in general, including architecture components, documentation, fragments of legacy code and any domain-related information that can be reused in the developing of a new system in that application domain.
Chapter 3

Program Representations

The program representation plays a key role in the candidature phase of a reuse reengineering process. An accurate survey of the program representation forms can be found in De Lucia [D95].

In this chapter, a program representation suitable to fit our "object isolation" method is defined. Some preliminary definitions are provided.

Informally, code analysis is a generic term used to denote many programmers activities "where the primary emphasis is on examining a piece of program code" [C89A]. In literature several forms of code analysis activities are defined, each of them focusing on different program representations, mainly depending on the aim driving the analysis process.

Data Flow Analysis

The control flow graph is a program representation used to perform a code analysis both intra-procedural and inter-procedural in order to explore the usage of the entities within the code. It is based on the concept of a flow graph [ASU86] [H77] which is a directed graph \( G=(s, N, E) \) where \( N \) is the set of nodes, \( E \) is the set of edges and \( s \) is a special node such that for each node \( n \) in \( N \) there exists a path from \( s \) to \( n \).

Particularly, a control flow graph is a flow graph whose nodes in \( N \)
represent single-entry/single-exit regions of executable code. The node $s$ represents the main procedure. The edges in $E$ represent data flow between code regions.

The control flow graph helps to perform data flow analysis. It can be used to detect any anomalous variable usage resulting by some previously undiscovered program errors. Data flow analysis is also fundamental for program slicing. In fact, after having decomposed the program into slices, the analysis of data flow and control flow graphs allows the determination of the dependencies between different variables, and the removal of redundant statements from program slices.

Call Graph

In order to understand a program, it is important to view it from different levels of abstraction. While the control flow graph depicts the program's structure at the statements level (as necessary for the program slicing), the call graph provides meaningful information at an higher level. The call graph is based on a flow graph, too. Particularly, in the call graph, the set $E$ represents the call dependencies between procedures. Some forms of call graph use a labelled flow graph (generalised program graph): in this case, each edge labels records information about the actual parameters in the calling statements.

3.2 A New Code Representation

In this section, a new code representation is defined in order to simplify the process of understanding the relationships generated by a common data accesses between procedures implementing entities of the application domain.

This new program representation is a variant of an inter-procedural call graph, providing information about data flow. Since it is sensitive...
about the temporal sequence of the invoking statements and of data accesses, it will be called "temporal graph" (TG).

The code representation involves three kinds of static analysis on the code:
- an analysis of the temporal sequence of the data accesses;
- an analysis of the relationships between the procedures (in the following with this name we denote the code fragments among them the candidate methods will be elected) rise by invoking statements and their invoking relationships;
- the data accesses performed by a procedure into different fragments of code, some information about the control flow.

The temporal graph is constructed by data analysis, presented in section 3.2.1, in which the statements within the code are examined in order to check the information about the data access by analysing how every single procedure accesses data. In the section 3.2.2 the definitions concerning the temporal graph are given.

3.2.1 Analysis of Data Access Type

Different actions can be performed on a data items [F076A], [F076B]. Our aim is to distinguish through the analysis of the statements within the code, whether a procedure accesses data to consult their value (reading access), and/or to permanently modifying them (writing access). To determine how a procedure accesses a data item, the statements referring the data item within the procedure must be examined.

```
01 PROGRAM Example (Output).
02 VAR int1, int2. INTEGER.
04 BEGIN
06 int1 := 10
07 int2 := int1 + 20
08 int1 := int2 - int1
09 WriteLn (int2 - 5)
10 END. (*Example*)
```

Figure 3.1 - Example of a simple Pascal [JK85] program

Note that the statements 06 and 07 induct a data dependence of int2 on int1 as well as the statements 07 and 08 induct a dependence of int1 on itself.
Even though the statement `<VAR int1, int2: INTEGER;` on line 03 references both data items int1 and int2, it does not affect the data access analysis, as it does not directly reference the two data items. Statements such as the declaration of a variable in FORTRAN, do not correspond to any operation on the value stored in a data, they are a peculiarity of the programming language. However, in some cases, if the program is well designed, to each statement indirectly referencing data items corresponds, within the procedure, an access to the referenced data items.

The statement on line 07 reads int1 without modifying it, as its value is only needed to evaluate the expression `<int1 + 20>`. Then, the statement uses int1. The result of this expression is assigned to int2, afterwards the value of int2 is changed by the execution of the statement on line 07. Analogously, the statement on line 08 changes the value of int1.

Different situations can be distinguished in the data accesses of these two statements. Statements such as that on line 07 write the value of a data item (int2 in the example) from scratch, then they create int2. On the contrary, statements as that on line 08 write a new value in the data items (int1 in the example) depending on its previous value. In this case the statement manipulates int1. Note that a statement can perform actions on more than one data item.

In summary, the type of data accesses of a statement are:

- **data using**,
- **data creation**,
- **data manipulation**

It is important to note that the analysis of data accesses also depends on the structure of accessed data. For our purposes, in the following, the data will be divided into two categories: records and variables. The distinction between variables and records depends on how statements refer to the data. A data with an elementary structure (involving only one level) is bounded to be a variable. A data with a complex structure involving several levels can be considered either a
record or a variable. It is a variable if (and only if) no single statement directly refers to any of its elementary fields. In other words, if all the statements in all the program components referring to the data are interested in the entire data, then the data is a variable.

The distinction between variables and records is important in the analysis of statements accessing the data in writing, in order to determine whether the access is either a creation or a manipulation. In the case of writing access to a records, for example, if the statement refers the entire structure then the access is by creating. If the statement refers only to some elementary of the elementary sub-fields, then the data operation is a manipulation, because the final content of data partially depends on the previous value stored in the data structure.

### Figure 3.2 - Example of COBOL source program defining a record in the DATA DIVISION
If all procedures only access it by referring to NAME-CUSTOMER, than it can be considered a variable.

Figure 3.2 show the definition of a COBOL record NAME-CUSTOMER defined within the DATA-DIVISION. Supposing that this record is part of a database of an archive. If a statement replaces the value only of the group of elementary fields ADDRESS, the access cannot be considered a creation of all the data item NAME-CUSTOMER as the value of the whole data partially depends on the previous value of the execution of the statement, as depicting a data manipulation. In fact, within an application domain using this record, an upgrade of this data item is due to the creation of a sub-field, and does not represent the creation of the whole data, but its manipulation.
3.2.2 The Temporal Graph

The aim of this language-independent program representation is to simplify the comprehension process of the relationships between two procedures created by a common access to a data item.

The program representation is based on the control flow graph $G = (s, N, E)$. The set of nodes $N$ is partitioned into three subsets of nodes: PN, RN and IN for "procedure nodes", "region nodes" and "information nodes"$^2$, respectively. In the following, the procedure nodes are represented by an oval, the region nodes as rhomboid and the information nodes as a square with rounded corners.

A procedure node (in PN) represents a single-entry / single-exit region of executable code (for example, a section in COBOL). The special node $s$ represents the program's entry. Note that, by definition of control flow graph, each procedure node has at least an invoking edge. Each procedure node has a unique entry point, and each edge is connected to the entry point of the procedure it invokes. The invoking edge towards $s$ has a different shape.

From the definition of control flow graph, for each procedure $P_h$ in $PN \subseteq N$, it is assumed there exists at least one path from $s$ to $P_h$. Differently from the control flow graph, for each node $P_h$ and $P_k$ in PN, there is an "invoking edge" $e$ in $E$ between $P_h$ and $P_k$ for each statement in $P_h$ invoking $P_k$. Figure 3.3 shows part of a temporal graph where procedure $P_h$ invokes $P_w$ more than once.

---

![Figure 3.3](image)

*Figure 3.3 - Between $P_h$ and $P_w$, there are two invoking edges. The sequence of the invoking statements is the sequence of the edges anticlockwise.*
An information node (in IN) with an invoking edge from \( P_h \) summarises information about the accesses of \( P_h \) on global data in a portion of code of procedure \( P_h \). Each statement involving a data item \( A \) in that portion of code is examined in order to set the corresponding access as \( "C(A)" \), \( "M(A)" \) or \( "U(A)" \) (creation, manipulation or use, respectively). Note that from each procedure node there can be several links to information nodes, but not consecutively.

The technique of walking the temporal graph involves visiting each node before its sons, following all the edges, starting from the left of the entry edge (the starting edge for \( s \)) and continuing to the right. If the visited node is a region node, then only one of the two set of edges starting from it is followed. Each traverse of the graph represents a different sequence of actions. A "path" between two nodes \( P \) and \( Q \) within the temporal graph is a walk following the rules above starting from \( P \) and ending in \( Q \).

Between two consecutive links to procedures and/or region nodes only one information node can be placed. Due to the limitations of static analysis it is possible to have an access performed under a condition statement. For example, in the statement\(^2\)

\[
\text{IF } <\text{condition}> \text{ THEN } <\text{write } A> \text{ ELSE } <\text{read } A>
\]

the access to data \( A \) is considered as a manipulation, and the corresponding action is set as \( "M(A)" \).

When a procedure executes an invoking statement inside an IF statement then from the procedure node, there is an edge to a region node in RN. From each region node one or two sets of edges can start, each set associated with a "T" (true) or "F" (false) value. Each set can contain edges to procedure, information and/or region nodes. The region nodes are labelled with "*" if the invoking statement is iterated (i.e., if the corresponding statement is in a loop, \(^3\)).

---

\(^2\) We will omit the terms "procedure", "information" and "region" when this does not lead to confusion.

\(^3\) The "write \( A \)" and "read \( A \)" is for statement accessing \( A \) by writing (creating or manipulating) it or by reading (using) it, respectively.
such as a WHILE, a REPEAT...UNTIL, etc) depending on the value of var_1. Note that, if the loop depends on a variable, as in the situation shown in figure 3.4 in which the loop depends on the value of var_1, the operation on the variable is in the information node linked to the entry of the region node representing the loop. The operation is repeated in each iteration of the loop.

As far as concerns the information about the data accesses, if a fragment of code relative to an information node contains only statements performing “data use” or “data manipulation” statements relative to a data A, then it “uses” or “manipulates” A, respectively, and the corresponding access to A is set to U(A), or M(A). The “data manipulation” is also performed if there are two different statements within that information node, the first of which reads A and the second one writes a new value in A.

The “data creation” accesses of an information node I_h is more difficult to determine. If a statement like <A:=6> is the first statement referring A, then I_h “creates” A, whatever statement referring to A follows it within I_h. If the first statement referring A is <A:=B+6>, (i.e if A is set as a function of another datum B) then it must be checked if B depends on A in the previous information nodes in any path between s and I_h, in order to avoid the possibility of indirect
data dependence of $A$ on itself. This is done by following in the temporal graph each walk going back from $i_k$ to the starting node $s$. If there is a dependence of $A$ on itself, then the statement performs a "data manipulation", and then the information node manipulates $A$ as well.

Figure 3.5 shows the case of an indirect dependence of $A$ on itself. The information node $P_z$ accesses $A$ by manipulating it.
graph, are sufficient to represent all the statements of the code, because it is well known that all control constructs can be represented as a combination of the IF THEN ELSE constructs in a code without GOTOS

Figure 3.6 - The figure above shows the partial temporal graph representing some procedures in a typical Bank Account Management System. The data database represents a file whose records contain information about the customer name (name-cust), the account number (account-ID), the money present into the account (cash), and a list of all the operations performed recently (list-operation), used for example to fill a coupon with the bank statement.

The program representation described above can be used at several level of abstraction. Figure 3.6 shows a temporal graph relative to a simple bank account management at a high level of abstraction. Many details are hidden in order to make easy the process of comprehension of the data and control flow together.

By using the technique of walking described above, the representation in figure 3.6 is very easy to read and to understand.
Chapter 4

An Improved Technique

Since usually legacy information systems have been written with an ad-hoc approach, there are undeniable problems in recognising which data structures and routines have to be grouped into a module candidate to implement the abstraction of an entity of the application domain. The different examples described in the section 2.2 demonstrate the difficulties in implementing an automatic tool able to extract meaningful objects from existing legacy code without considerable application domain knowledge, to recognise the implemented abstractions possibly associated with candidate modules, to purge them from all the components altering these abstractions, and finally, to match them to entities within the application domain.

In this section, a method to identify candidate object-like modules from existing code is presented. It benefits from the mainstream of similar researches described in the sections 2.5.2, 2.5.3 and 2.5.4. Similarly, it uses a bipartite graph (figure 2.8) to represent the relationships between procedures and commonly accessed data and to perform an easier isolation of notable sub-graphs, each of them candidates to implement an entity of the application domain.

Even though this method can be merely considered only an improvement of the technique defined by Canfora et al. (paragraph 2.5.4), the improvement it induces upon the previous algorithm cannot be considered trivial. This improvement is aimed to discriminate noisy connections linking candidate modules implementing different entities in the application domain. The new algorithm adds to the clustering phase two more phases: the data duplicating and
Chapter 4

**Data Refining Phases**

Basically, these phases focus both on the accuracy level of data structures and on the analysis of the relationships between two procedures accessing a common data item. This is to achieve a better comprehension of the state of each potential object by treating the system of data items implementing the object's state, also stored into these data structures. This should also achieve a more accurate object-like module and to improve the understanding process of the relationships between different potential modules due to the interaction of their methods.

Furthermore, by joining the data flow analysis to the analysis of the relationships between procedures accessing the same data, this helps to establish whether the common access of several procedures to a data item denotes that they are methods of the same object. This analysis leads to candidate modules abstracting lower level entities within the application domain, thus simplifying the process of understanding them and of matching them to domain entities. However, this analysis requires the application of domain knowledge to the iterative step, but this application is helped by the context.

Informally, the technique will refine the previous algorithm in three aspects:

- With a more detailed analysis on "how" a procedure accesses a data, the technique distinguishes not only between reading and writing accesses, but also checking if the procedure uses the value set by another procedure, thus creating a sort of dependence between the two procedures. This improves the understanding process needed to establish when two procedures accessing a data are a method of the object whose state is stored in the common accessed data.

- With a more detailed data analysis, possibly also including the refinement of structured data in a equivalent set of less structured ones. Smaller grain objects are easier to handle. This enriches the domain knowledge and reduces the effort of the subsequent concept assignment phase.

- With a lighter bipartite graph produced by including the links representing
either a procedure accessing a data item by modifying it (thus, to ensure information hiding, the procedure is a method of that object whose state is stored in the data) or a procedure using a data value without modifying it, when the invoking procedure can not send that data item as a parameter in the invoking statement.

All these issues are aimed to reduce the number of links that have more likelihood being noisy, thus improving the comprehension of the legacy code and allowing an easier isolation of candidate objects.

4.1 Overview of the Algorithm

Similarly to the algorithm of Canfora et al., the aim of the new algorithm is to isolate strongly connected sub-graphs within the bipartite graph representing the interrelationships between data structures (potentially implementing the state of candidate objects) and procedures (potentially implementing methods acting on the objects' state). On the other hand, the two added phases, with the different way to draw the bipartite graph, lead to a modified algorithm.

The outlines of the algorithm are sketched in figure 4.1. Note that the three phases are represented: data clustering, data duplication and data refining. Their sequence is not made up by the algorithm, as it depends on the features of the system it is dealing with. The algorithm receives in input the whole system and the three phases are performed while the bipartite graph does not assumes the form of a set of disjoint sub-graph, each of them is candidate to implement an object into the application domain.

After each phase, the program representations have to be redrawn, as any of the phases change the structure of the legacy code. All the phases are fully detailed below.
4.2 Details

The process starts by isolating data structures and procedures from which to select those that will be composing an object-like module (data structure and procedures acting on these) candidate to implement an abstraction of an entity of the application domain. The decomposition process of the code into a set of routines candidates to implement the object methods takes advantage of the modularity of the code (see par. 3.2.2). In the following, these isolated routines will be called “procedures”. Each data items defined within the software system forms a node of the subset relative to data. Note that the data items are chosen at the highest level of abstraction, i.e. structured data with subordinate elementary
data items is a single node. After that these components are made available, the program is represented at different abstraction levels through the temporal graph and the bipartite graph.

4.2.1 Drawing the Program Representations

The temporal graph is drawn with the process that has been already described in the paragraph 3.2. Note that the temporal graph strictly depends on the granularity level of data item, and on whether the data item is a record or a variable, as it affects the type of data access. As the data granularity might vary at each iterative step of the algorithm, a constant upgrade of the temporal graph is consequential.

The Bipartite Graph

As far as concerns the bipartite graph, procedures and data items are represented as two disjoint sets of nodes. After having drawn the two sets of nodes, a static analysis concerning the access type of the procedure to the data is needed to draw the set of edges. This information can be taken from the temporal graph by examining all the information nodes relative to that procedure.

The representation of the bipartite graph is different from that used by the previous algorithm, as the new bipartite graph takes into account whether the access is a writing or a reading access. Mainly, the edges representing writing accesses between a procedure and data are always drawn, because by writing a data item, a procedure changes the state of an object. Then, in order to ensure information hiding, that procedure must be linked to that data item storing part of the object's state, as it might be an object method.

The edges representing reading accesses are drawn with some exceptions if a procedure P accesses by reading a data item A, and all the procedures invoking P access A by writing to it, then it is likely that A does not belong to P's state, but is used as parameter to P.

In summary, the modified bipartite graph contains all the links between a
procedure $P_k$ and a data $A$ such that:

1. $P_k$ accesses $A$ in writing (some *information node* of $P_k$ accesses $A$ by creating it or by manipulating it);

2. $P_k$ accesses $A$ through reading but in the set of procedures invoking it there is at least one with a reading accesses (some *information node* of $P_k$ accesses $A$ by using it).

Note that if there is a link into the bipartite graph between procedure $P_k$ and data item $A$, then $P_k$ accesses $A$. The opposite is not true: in fact, if a procedure $P_k$ accesses $A$, there can be no link between $P_k$ and $A$ in the bipartite graph.

Note also that the starting bipartite graph can be easily obtained by purging from the one in Canfora et al. all the edges representing a reading access of a procedure $P$ to a data $A$, if and only if all the procedures invoking $P$ access data $A$ in writing.

As for the temporal graph, the bipartite graph also starkly depends on the granularity level of data item, and on whether the data item is a record or a variable, as it affects the type of data access. In this case too, a constant upgrade of the temporal graph is consequence of the varying of the data granularity throughout the execution of the algorithm.

### 4.2.2 The Computation of the Vector $AIC(\cdot)$

At each iterative step, the modified algorithm examine the bipartite graph in order to compute, for each procedure $P$, a vector of indexes $AIC(P)$:

$$AIC(P) = \frac{\sum_{A \in \text{PostSet}(P)} \# \{P_i | P_i \in \text{PreSet}(A) \land \text{PostSet}(P_i) \subseteq \text{PostSet}(P)\}}{\sum_{A \in \text{PostSet}(P)} \# \{\text{Preset}(A)\}} - \sum_{A \in \text{PostSet}(P)} \frac{\# \{P_i | \text{PostSet}(P_i) = \{A\}\}}{\# \{\text{Preset}(A)\}}$$

The aim of the computation of the vector $AIC(\cdot)$ is to help decide which is the procedure more worthy to cluster around. Here and in the following, "to cluster around procedure $P$" means to cluster the sub-graph containing $P$, all the data items it accesses and all the procedures accessing a subset of these data items.
In fact, into the modified system, the vector of indexes $\Delta IC(P)$ establishes, by comparison among its elements, which is the procedure with the highest difference between the internal connectivity of the sub-graph generated and the internal connectivity of the sub-graphs merged, thus allowing the evaluation of how the clustering around $P$ changes the internal connectivity of the bipartite graph.

At each step of the iterative algorithm, the routines having an index $\Delta IC(P)$ "sufficiently high" are used to generate cluster around $P$.

During the execution of the modified algorithm, the data duplication and data refining phases can be performed at each iterative step in order to increase the value of some of the indexes within the vector $\Delta IC()$, such that there is always a procedure of the bipartite graph that can be clustered in order to obtain a candidate module. This process combines application domain knowledge to the analysis of the data structures, and takes into account the meaning of the data within the application domain.

Note that the order of the phases is rather irrelevant and they mainly depend on the peculiarity of the legacy information system.

4.2.3 Data Duplication

The aim of "data duplication" phase is to break down some 'logical links' between groups of procedures due to a common access of global data items. In fact, not all cases a common data access are intentional such that the procedures are sharing the value of this data item for their computations, thus involving that they both are methods of the same object, whose state is partially stored in the commonly accessed data item. Sometimes, that data item can be considered "local" to a group of procedure.

When a data item is local to different procedures (for example a variable $i$ usually used as counter in any loop) a trivial reengineering intervention to be performed before the modularisation is to "split" $i$ by renaming it with different names in any procedure accessing it.
The problem of managing local data has been faced by Markosian et al [BBKMN94] within a process of reengineering of a COBOL legacy information system. They have introduced the concepts of input and output parameters in order to define when a data item A is local to a procedure. An input parameter relative to a PERFORM statement is a data item A that is set within a procedure P before the statement PERFORM P and it is used in P before A is set again. Analogously, an output parameter is the data item A is set within a performed procedure P then it is used before being set following some perform of that paragraph. After having defined the input and output parameters, Markosian et al define to be “local to that procedure” all the data referenced by the procedure that are neither input nor output parameters.

For our purpose, only global data will be considered. However, the definition of “local data” will be extended to groups of procedures. Analogous to the reengineering process performed to the localise the data to a procedure, the main idea of the data duplication phase is to isolate groups of procedures in which a data can be considered local, and then split this data item into many items by renaming it with a different name within each group.

Precisely, we will deal with groups of code fragments, represented by information nodes. In fact, as it will be fully detailed in the following examples, after having isolated the groups relatively to a data item A, it is possible that a procedure belongs to more than one group, whereas, the implicated information nodes strictly belong to a single group (see figure 4.4 and table 4.2). Consequently, the output of the analysis of the data duplication phase will be a set of groups of information nodes. In each of these groups the data item will assume a different name, thus obtaining a “data duplication”.

However, the notation dealing with groups of procedures is still preferred to the one dealing with groups of “fragments of codes” (the information nodes), because it simplifies the process of drawing the bipartite graph, on which the algorithm will be applied. It will be clarified, in case of procedure belonging to several groups at the same time, which of the information nodes belongs to each group. In this way we also hope to keep coherent with the aim of “duplicating...
the data items that are local to groups of procedures”.

Informally, a data item can be considered as “local to a group of procedures” respecting the two conditions:

1. in each path within the group, the first information node referring A creates it;
2. in each path getting out from the group, the first information node referring A creates it.

Figure 4.2 - A typical temporal graph with the specification of a group of procedures in which a data can be considered local

It is clear that the central issue is the definition of such a group of procedures. Note that a group is relative to a single data item, and the groups vary depending on the data. Since throughout the execution of the algorithm the data items and their accesses might change, the groups have to be computed each time the temporal graph has been changed (i.e. at each iterative step). Consequently, the data duplication phase has to be performed each time on a different temporal graph.

All those groups are characterised by having a procedure creating the data and, the value of the data in the other procedures of the group depend on by the computation of the procedures of the same group already executed and accessing the data.

For any data A, a group of procedures in which a data item A is local has the following features:

- there is one and only one procedure owning an information node creating A
(the *information node* and the *procedure node* will be called $I_0$ and $P_0$ in the following examples);

- in each procedures $P_k$ of the group, the value of $A$ only depends on $I_0$ and on other procedures of the group which are in the path from $I_0$ to $P_k$.

For an *information node* $I_0$ creating a data item $A$, the following definitions will be used in order to identify the procedures $P_k$ in which the value of $A$ depend on $I_0$.

Informally, the definition 4.1 defines when the value of a fixed data item $A$ in a procedure derives the execution of a portion of code of another procedure.

**Definition 4.1** - Let $G=(s, N, E)$ be a *temporal graph*. Let $I_0 \in \text{IN}$ be an *information node* creating data item $A$; let $P_0 \in \text{PN}$ be a *procedure node* such that $(P_0, I_0) \in E$ ($I_0$ of $P_0$). Let $I_k$ be an *information node* of $P_k$ using or manipulating data $A$. Then $I_k$ of $P_k$ derives $A$ from $I_0$ if there is a path between $s$ and $I_k$ (excluded) containing $I_0$.

For sake of simplicity, in the following the notation $I_0$ of $P_0$ will be used to denote that there exists $I_0 \in \text{IN}$ and $P_0 \in \text{PN}$ such that $(P_0, I_0) \in E$.

Note that, from the definition of *temporal graph*, there might be many paths from $s$ to $I_k$, due to the presence of *region nodes*. In definition 4.1, the condition that $I_k$ is excluded from the considered path, means that any *information node* creating $A$ does not derive $A$ from any other *information node*.

A procedure can *derive* a data item from many *information nodes* at the same time. In the example of figure 4.3, both $I_{k1}$ and $I_{k2}$ of $P_k$ derive data item $A$ from $I_{j1}$ of $P_j$, from $I_{01}$ of $P_0$ and from $I_{k1}$ of $P_h$. $I_{j1}$ only derives $A$ from $I_{j1}$ and $I_{k1}$ of $P_h$. 

63
Definition 4.2 - Let $G=(s, N, E)$ be a temporal graph. Let $I_{k_i}$ be an information node of $P_k$ deriving data $A$ from $I_0$ of $P_0$. If every information node in any path from $I_0$ (excluded) to $I_{k_i}$ (included) does not contain information nodes creating $A$, then $I_{k_i}$ of $P_k$ directly derives data $A$ by $I_0$. Procedure $P_0$ directly derives $A$ from $I_0$.

Note that, by definition 4.2, any information node $I_{k_i}$ of $P_k$ directly derives $A$ by $I_0$ if in any traversal of the temporal graph between $I_0$ to $I_{k_i}$ (inclusive), the node $I_0$ is the only one creating $A$, i.e. in any traversal of the temporal graph between $s$ and $I_{k_i}$ (inclusive) passing from $I_0$, the node $I_0$ is the last (in temporary order) one creating $A$.

Also in this case, a procedure can directly derive a data item from many information nodes at the same time. In the example of figure 4.3, both $I_{02}$ and $I_{k_2}$ have a directly derive $A$ from $I_{01}$ and $I_{h_1}$.

If an information node $I_{k_i}$ directly derives $A$ from an unique node $I_0$, then the derivation is also exclusive.

Definition 4.3 - Let $G=(s, N, E)$ be a temporal graph. Let $I_{k_i}$ be an information node of $P_k$ directly deriving data $A$ by $I_0$ of $P_0$. Then $I_{k_i}$ of $P_k$ exclusively derives $A$ by $I_0$ of $P_0$ if there not exists any other information node $I_h$. 

Figure 4.3
of P, which I,, directly derives A from.

The word "exclusive" referred to the derivation is due to the fact that P, derives A only (exclusively) by I, of P,. This ensures that the value of A in P, depends on the execution of I,.

Note that, by definition 4.3, in figure 4.3, there are only two examples of exclusive derivation: I, and I, exclusively derive A from I, and I, respectively.

In summary, the derivation relationships between the information nodes in figure 4.3, are represented in the table 4.1 below.

<table>
<thead>
<tr>
<th>Information node</th>
<th>derivation</th>
<th>direct derivation</th>
<th>exclusive derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I,</td>
<td>I, Ih, I,</td>
<td>I, I,</td>
<td></td>
</tr>
<tr>
<td>I,</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I,</td>
<td>I, I, Ih, I</td>
<td>I, I,</td>
<td>I, I,</td>
</tr>
<tr>
<td>I,</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I,</td>
<td>I, I, I, I</td>
<td>I, I,</td>
<td>I, I,</td>
</tr>
<tr>
<td>I,</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I,</td>
<td>I, I, I, I</td>
<td>I, I,</td>
<td>I, I,</td>
</tr>
</tbody>
</table>

Table 4.1

These definitions define a dependence having many similarities with the data dependence of Ottenstein et al. [FOW84] [OO84].

In order to proceed to the data duplication phase, all of the groups of procedures relative to data must be identified. For a data A, a group of procedures is composed by the procedure P, with an information node I, creating A, and all the procedures P, exclusively deriving A from I, The procedures composing the group can be identified through the definitions above.

The value of a data A in any procedures P, (with k > 0) within a group relative to that data item, depends on I, and on all procedures in the path from P, to P, manipulating A (i.e. all procedures that have modified A).

Through a static analysis of the code such groups of procedures are
identified, and a data duplication phase "splits" the data into many items, one for each group. This process is carried out for each data.

In the example shown in figure 4.4, procedures the information node of P₁ and the one of P₃ (figure 4.4a), directly derive A from I₀₁. On the other hand, the information node of P₅ directly derives A from I₀₂. With the further assumption that there are no other procedures invoking the procedures of figure 4.4a, the derivation is also exclusive.

Figure 4.4 - The procedure P₀ has many information nodes creating A. In this situation, the procedure will contain the different items in which the data is split, i.e., the several portions of source code corresponding to the information nodes I₀₁ of P₀ contain different variables deriving by the splitting of A.

In summary, the derivation relationships between the information nodes in figure 4.4, are represented in the table 4.2 below.

Two groups are derived from this example from a simple observation of table 4.2, it is easy to split up data A into two items A₁ and A₂ relatively to the groups of procedures {P₀, P₁, P₃} and {P₀, P₄, P₅}. Note that procedures P₂ and P₄ have not been added into any groups because they do not reference A, thus the fact to be belonging to a group does not affect their code.
In this example, procedure $P_0$ contains more than one information node creating $A$, then $P_0$ belongs to more than one group. In this case, $P_0$ will contain references to all the items in which $A$ will be split. Figure 4.4b shows the data $A$ duplicated in two items $A_1$ and $A_2$. The portion of code relative to $I_{01}$ and $I_{02}$ will contain reference to $A_1$ and $A_2$ respectively. As procedure $P_0$ belongs to both the groups, the groups should be written as $G_{A_1} = \{I_{01} \text{ of } P_0, P_1, P_3\}$ and $G_{A_2} = \{I_{02} \text{ of } P_0, P_5\}$, thus showing that the information nodes of $P_0$ will have references to different target data items.

As already briefly introduced in the presentation of the data duplication phase, earlier in this paragraph, this might just be considered a notation problem. However, it could be easily overcame by composing the groups on the basis of information nodes, then obtaining the two groups $\{I_{01}, I_1, I_3\}$ and $\{I_{02}, I_5\}$, instead of having the not disjoint groups of procedures $\{P_0, P_1, P_3\}$ and $\{P_0, P_5\}$. However, it is still preferred the notation dealing with groups of procedures instead with groups of "fragments of codes" in order to keep the notation coherent with its aim to draw the bipartite graph on which the algorithm will be applied. In fact, the notation dealing with groups of procedures allows a more immediate process of drawing the bipartite graph, than the notation dealing with groups of fragments of codes.

After having established the groups of procedure relatively to a data item $A$, the code relative to each information node is upgraded by changing the data item's name and all the related statements (data item's declaration, comments, documentation, etc.)

<table>
<thead>
<tr>
<th>Information node</th>
<th>derivation</th>
<th>direct derivation</th>
<th>exclusive derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_{01}$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$I_{02}$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$I_1$</td>
<td>$I_{01}$</td>
<td>$I_{01}$</td>
<td>$I_{01}$</td>
</tr>
<tr>
<td>$I_3$</td>
<td>$I_{01}$</td>
<td>$I_{01}$</td>
<td>$I_{01}$</td>
</tr>
<tr>
<td>$I_5$</td>
<td>$I_{01}, I_{02}$</td>
<td>$I_{02}$</td>
<td>$I_{02}$</td>
</tr>
</tbody>
</table>

Table 4.2 - In the table, the information nodes $I_{kn}$ are related to the procedure $P_k$. 
After this phase, the links starting from these procedures toward the commonly accessed data form a group of procedures whose computation depends on a common data item. In this case, all the procedures of that group can be considered as methods of the same object whose state is also stored in the common accessed data.

4.2.4 Data Refining

The "data refining" phase refines the granularity level of some of data items having a large number of accesses, according to their meaning into the application domain and to the benefits brought to the execution of the algorithm. This phase allows, at each step of the algorithm, the subdivision of a data accessed by many procedures into different data items. With the support of the knowledge of an human expert, the data is analysed to check if all the information belongs to the same object of the application domain, and if the procedures share the same information about the same object. If these conditions do not hold, then the data can be refined into a set of data items, each of them containing the information needed for a smaller number of procedures.

Due to the change of the data structure, both the data representations have to be updated. This re-analysis must take care of new data accesses. In fact, a procedure that before the refining was accessing the data that has been refined, might access only part of the target "sub-data". Furthermore, the kind of access can change, since a record can produce some variables as result of the refinements.

For example, the record NAME-CUSTOMER in figure 3.2 might be refined in the data items corresponding to level 03 of the data structure definition. According to the definition of record and variable in paragraph 5.3.2, if all procedures access only the new data items NAME, BIRTH-DATE and ADDRESS (without accessing directly their elementary fields), then they all are variables. In case a statement sets NAME by creating it, then the statement manipulates the record NAME-CUSTOMER, before the refinement, on the other hand, after the refinement the statement creates the variable NAME. This must be taken into...
account in establishing the data access of that procedure before and after the refinement phase.

Evaluation of the Data Refining Phase

In order to evaluate the benefits brought to the execution of the algorithm, some indexes can be defined to help the human expert decide which data are more worthy of refining. The human intervention should help also in this process.

![Diagram showing data access](image)

Figure 4.5 - In figure 4.5a the vector of values of ΔI(C for the procedures $P_1$, $P_2$, and $P_3$ is $(0.75, 0.25, 0)$. By refining the data $A_1$ (figure 4.5b) in the set of items $A_{31}$, $A_{22}$, and $A_{31}$, the vector is $(0.80, 0.30, 0)$. After the refining of data $A_1$ (figure 4.5c) in the set of items $A_{41}$, $A_{42}$, and $A_{43}$, the vector is $(0.75, 0.60, 1.00)$.

The decision criteria for the data refinement phase depend on the index $\Delta I(C(P_k))$ before the refinement of $A$ computed for any procedure $P_k$ referring $A$, and the index $\Delta' I(C(P_k))$ after the data refinement. Other aspects affecting the decision criteria are the peculiarity of the particular legacy information system and the reasons driving the reengineering process, thus the decision can be taken with regards to this aspect.

From figure Figure 4.5, it can be observed that the duplication of $A_4$ causes the isolation of a strongly connected sub-graph. In this case, the data more worthy of refinement is the data item $A$ which maximises the value $\max_{P_k \in \text{PreSet}(A)} (\Delta' I(C(P_k))) - \Delta I(C(P_k))$. Of course, the refining of $A_4$ will be preferred to that of $A_2$ if it assumes a meaning in the application domain.
4.2.5 Termination of the Algorithm

The algorithm terminates when the graph is transformed into a set of strongly connected sub-graphs. Each sub-graph is composed of some data representing the attributes of an object, and thus its state, and some procedures representing the methods of that object.

The proof that the algorithm will finish in a finite number of steps follows the same reasons of the algorithm in [CCM96], because the phases added change the value of the indexes fixed by the previous algorithm only by increasing them. Otherwise, none of those phases can be performed and the algorithm will carry on in the standard way defined by Canfora et al., that is guaranteed to finish in a finite number of steps.
Chapter 5

A Case Study on a COBOL Source Code

The COBOL (Common Business Oriented Language) programming language was developed in 1959 by a committee composed of government users and computer manufacturers. Since then, various COBOL committees have met to ensure that the evolution of the language through time followed an orderly fashion, thus making COBOL the most frequently used computer language directed at data processing objectives, extensively used in administrative applications processing of a large amount of input and output data.

Yourdon¹ said "One of the oldest, and arguably the most successful and popular of all programming languages, COBOL has been declared dead so many times that I've lost track counting ... but COBOL lives on".

Table 5.1 shows the results of a recent research conducted by Caper-Jones [C92] regarding the state of the art of software maintenance. It shows COBOL as the "dominant arena" for software maintenance. In this review, it is possible to gain an insight into the proportion of effort required in the industrial field regarding the maintenance of COBOL legacy information systems against other conventional programming languages.

In the recent literature a number of papers have specifically described different approaches all successfully employing reengineering techniques dealing with COBOL legacy information systems. Sneed's work is important in this area, because it successfully employees reengineering techniques [S92] to migrate...

¹ Extract from <http://www.yourdon.com/ap/9609INTRO.HTML>
existing COBOL applications, and to extract object oriented specification [S91] and object oriented design documentation [NS95] from existing COBOL applications running on mainframe.

<table>
<thead>
<tr>
<th>Language of the software being maintained</th>
<th>estimated number</th>
<th>portion of the total</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL</td>
<td>461,500</td>
<td>45%</td>
</tr>
<tr>
<td>Other procedural languages *</td>
<td>100,000</td>
<td>10%</td>
</tr>
<tr>
<td>C</td>
<td>95,000</td>
<td>9%</td>
</tr>
<tr>
<td>Database Languages</td>
<td>86,250</td>
<td>8%</td>
</tr>
<tr>
<td>Program Generators</td>
<td>76,000</td>
<td>7%</td>
</tr>
<tr>
<td>Assembler, all hardware</td>
<td>60,000</td>
<td>6%</td>
</tr>
<tr>
<td>Fortran</td>
<td>40,250</td>
<td>4%</td>
</tr>
<tr>
<td>Object Oriented Languages</td>
<td>29,750</td>
<td>3%</td>
</tr>
<tr>
<td>Basic</td>
<td>22,750</td>
<td>2%</td>
</tr>
<tr>
<td>Ada</td>
<td>22,500</td>
<td>2%</td>
</tr>
<tr>
<td>Special Purpose Applications</td>
<td>17,500</td>
<td>2%</td>
</tr>
<tr>
<td>PL/1</td>
<td>7,000</td>
<td>1%</td>
</tr>
<tr>
<td>Pascal</td>
<td>4,750</td>
<td>0%</td>
</tr>
<tr>
<td>LISP</td>
<td>2,500</td>
<td>0%</td>
</tr>
<tr>
<td>APL</td>
<td>1,500</td>
<td>0%</td>
</tr>
<tr>
<td>Total full time equivalent</td>
<td>1,027,250</td>
<td>100%</td>
</tr>
</tbody>
</table>

*Table 5.1 - Number of the full-time maintainers programmers in the USA in 1991, by languages read [C92]. Note that portion column does not add to 100% because of the rounding.

* Not elsewhere classified

The aim of this chapter is not to give an exhaustive explanation about the method above, using a real example, but to show how to adapt the method to the peculiarity of a given programming language such as COBOL, while respecting in the meantime the main ideas of the technique. By way of a case study, a simple COBOL program has been analyzed, and the technique being used on it. All the necessary arrangements to adapt it to the peculiarity of COBOL are underlined throughout this section.

As already seen in the sections from 2.2.1 to 2.2.3, it still appears impossible to implement fully automatic techniques to reverse engineer a typical COBOL program. The semantics, the meaning of the specifications are irreversibly lost in
the process of converting the specifications to design and then to code. Comprehension activities are required both at the program and at the architectural level. In particular, an integration of top-down and bottom-up understanding strategies [VMV94], [VMV95] can successfully identify software components and map them onto meaningful objects. The final step of each method of object identification is a concept assignment process performed by a human inspector in order to validate the candidate objects.

5.1 The Need of a Standard: the ANSI COBOL Standard

Due to international trade agreements, the global marketplace is becoming a reality. In this situation, both the private and public sectors, have understood that "standards, if adopted throughout the world, create a large market instead of many fragmented markets". The companies in every industry and of every size are realising that a business keeps its competitive edge in the face of national and global market changes only by using a strategic standardisation.

In the computing field, the need to create standards has been recognised as important from the beginning. This is particularly true for COBOL, a language specifically designed for commercial applications, usually operating on a large volume of data. It was created in 1959 by the CODASYL Committee in a meeting convened by Department of Defense (DOD), particularly dissatisfied by the lack of standards.

Due to the availability for many platforms (from desktop Intel machines to huge IBM ESA mainframe systems) and to flexibility (leaving a COBOL program to be compiled and to run on a variety of different machines with very few changes to the original code), there were so many variations among COBOL compilers produced by different computer manufacturers that it was decided that the American National Standards Institute (ANSI) would oversee COBOL.

2 Gary Tooker, manager of Motorola Inc and Vice Chairman and CEO

3 CODASYL is an abbreviation for the Conference on Data Systems Languages. The CODASYL committee included representatives from academia, users groups and computer manufacturers.
standards, to permit to COBOL to survive.

In 1985, in the attempt to create a series of international quality standard for the COBOL LISs reengineering processes, the ANSI produced a document in which are listed all the features that a good COBOL object oriented COBOL application should have.

The source system must be decomposed in several modules, each of them containing a single object interacting with the other ones by message passing. Each object correspond to a class, i.e., to an abstract data type encapsulating the state of the object trough its attributes, and to a set of actions (or methods) modifying this state. To insure information hidings, the classes should have both private and public storage: the public one is accessed only by subordinate classes, whereas private one is protected by any access.

Messages define everything an object can do, i.e., its interface. The classes can invoke directly methods of an external class, i.e., not subordinate by inheritance right. In order to do this a method of a class is defined as an entry point with a definition of the parameters it receives. For the features of COBOL every method has access to data of the invoking class. Thus classes subordinate by inheritance rights can be invoked by a message without parameters; on the contrary, the methods of an external class can be invoked only by messages declared in a separate import/export area and passed as parameters in a CALL statement. Every time that a method is invoked, the control can be returned back to the caller or to another class. To insure information hiding, the classes should have both private and public methods. The last ones are accessible only to the class in which they are encapsulated.

The classes should have multiple inheritance, i.e., they have to be able to inherit data attributes from more than one superordinate class. The same applies to methods which are inherited from superordinate class. The inherited data must be declared referring to the class from which they are inherited. The same applies to methods which are inherited from superordinate classes.
In order to ensure that each module do not exceed a certain size and complexity, some limitations are fixed: the number of attributes of a class is limited to 100 and the number of methods per class is limited to 10. The number of statement in a method must be less than 20 statements. A message should be restricted to five parameters. This limitation provides to the system a high degree of modularity.

5.2 Presentation of the Case Study

The technique has been used on a COBOL program of 1500 LOC, large enough to give interesting demonstration of the application of the algorithm presented above. The program consists of a batch program with 5 flows defined in the INPUT-OUTPUT SECTION and in the FILE SECTION.

The whole structure of the COBOL program BATCH1 is outlined in figure 5.1. The arrows are the links between the program and the external files represents the input and/or output relation. The 5 external files and the 3 files included in the program by a COPY statement are represented by different

---

4 Stored with the number FIPS-PUB-21-2 and worked out by Yourdon, Microfocus and other in [Y90]
notations.

During the data description phase, the COBOL permits to keep copies of data description in program libraries in the computer system, enabling the description to be copied into the programs by using the COBOL statement COPY. Some of the external files were defined externally, and then included within the program by using a COPY.

5.3 Isolation of Procedures and Data

In order to decompose the monolithic program into a system of interacting object-like modules, the first step is the identification of a collection of data structures implementing the state of the objects, and a set of modules candidate to implement their methods. These two sets will be represented by the two disjoint sets of nodes of the bipartite graph. The granularity level of both of them depends on several factors, such as the features of the particular legacy information system and the environment in which it actually runs, and the reasons driving the reengineering process.

5.3.1 Isolation of Procedures

In the literature there exists a number of techniques for treating the restructuring processes performed in order to make a COBOL legacy system modular and well structured, thus facilitating the decomposition process of the code into a set of modules (a set of paragraphs, of sections or isolated statements) candidates to implement a method of an object. This restructuring process will achieve the aim of having a hierarchy of code segments, each with a single entry and a single exit and with GOTO statements within a segment of code, but not outside of it.

The modularity of the code allows to isolate procedures (sections) through a simple static analysis of the code. In our example, the 75 paragraphs shown in table 5 2 have been isolated. It is possible to note that all the paragraph names start with an alphanumeric code of 4 digits.
A simple analysis has shown that all the paragraphs starting with the same alphanumeric code were part of the same procedure, thus allowing to extract 31 procedures consisting of 2 or more paragraphs each. Their names have been changed with a code of 4 digit depending on the previous name. Only procedures A100-HOUSEKEEPING and A100-FILE-STARTS, both starting with A100, make necessary to change the name into A10H and A10F, respectively. The 31 new procedures are listed in the table 5.2 below. The last paragraph of each procedure contains only the statement EXIT.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Paragraphs</th>
<th>File Name</th>
<th>Exit Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>A100-HOUSEKEEPING</td>
<td>C210</td>
<td>C280</td>
<td></td>
</tr>
<tr>
<td>A100-FILE-STARTS</td>
<td>C220</td>
<td>C280</td>
<td>D240</td>
</tr>
<tr>
<td>A100-FILE-STARTS-EXIT</td>
<td>C230</td>
<td>C280</td>
<td>D240</td>
</tr>
<tr>
<td>A200-MAIN-PROCESS</td>
<td>C240</td>
<td>C280</td>
<td>D250</td>
</tr>
<tr>
<td>A200-BYPASS-UPDATES</td>
<td>C240</td>
<td>C280</td>
<td>D250</td>
</tr>
<tr>
<td>A200-MAIN-PROCESS-EXIT</td>
<td>C240</td>
<td>C280</td>
<td>D250</td>
</tr>
<tr>
<td>A100-TERMINATION-RTN</td>
<td>C250</td>
<td>C280</td>
<td>E250</td>
</tr>
<tr>
<td>B100-OPEN-FILES</td>
<td>C250</td>
<td>C280</td>
<td>E250</td>
</tr>
<tr>
<td>B100-OPEN-FILES-EXIT</td>
<td>C250</td>
<td>C280</td>
<td>F100</td>
</tr>
<tr>
<td>B200-READ-VIDEO-BONDS</td>
<td>C250</td>
<td>C280</td>
<td>F100</td>
</tr>
<tr>
<td>B200-READ-VIDEO-BONDS-EXIT</td>
<td>C250</td>
<td>C280</td>
<td>U100</td>
</tr>
<tr>
<td>B210-EDIT-DATA</td>
<td>C250</td>
<td>C280</td>
<td>U100</td>
</tr>
<tr>
<td>B210-EDIT-DATA-EXIT</td>
<td>C250</td>
<td>C280</td>
<td>U110</td>
</tr>
<tr>
<td>B220-UPDATE-FILES</td>
<td>C250</td>
<td>C280</td>
<td>U220</td>
</tr>
<tr>
<td>B220-UPDATE-FILES-EXIT</td>
<td>C250</td>
<td>C280</td>
<td>U220</td>
</tr>
<tr>
<td>B230-PRINT-PLANS-REPORT</td>
<td>C270</td>
<td>C280</td>
<td>U220</td>
</tr>
<tr>
<td>B300-CLOSE-FILES</td>
<td>C280</td>
<td>C280</td>
<td>U300</td>
</tr>
<tr>
<td>B100-CLOSE-FILES-EXIT</td>
<td>C280</td>
<td>C280</td>
<td>U400</td>
</tr>
<tr>
<td>C200-NO-PLANS-MATCH</td>
<td>C280</td>
<td>C280</td>
<td>U400</td>
</tr>
<tr>
<td>C200-PRINT-NOT-FOUND</td>
<td>C280</td>
<td>C280</td>
<td>2999</td>
</tr>
<tr>
<td>C200-NO-PLANS-MATCH-EXIT</td>
<td>C280</td>
<td>C280</td>
<td>2999</td>
</tr>
</tbody>
</table>

**Table 5.2**
The two procedures B220 and C230 have been excluded from the following analysis, as they do not access any data. For sake of simplicity, in order to increase the size of the source code as less as possible, the procedures invoking them have been chosen to perform the functionality of B220 and C230, as they are less than the procedures invoked by the excluded ones. This information has been extracted from the PERFORM GRAPH of figure 5.2 below.

Table 5.3 - The 31 procedures listed have been extracted from table 5.2. This task has been simplified by a simple examination of the paragraphs' name.

5.3.2 Data Analysis

The data isolation process strongly depends on the peculiarity of the
programming language. As COBOL is a business-oriented language, it is designed to process a large amount of data. The structure of the language is highly structured to accomplish the business data processing; consequently the data structure is highly organised in a hierarchical structure. Through a static analysis, the data have been considered at the broadest level: COBOL records (data composed of one or more group items and/or elementary items); COBOL variables (data with elementary structure), as defined in the DATA DIVISION.

In the data list there also has been added a table index that the COBOL does not define in the DATA DIVISION.

Unused Analysis

The "unused analysis" examines all the members included in the program but not used by it. Through this analysis, the unused data items have been isolated and excluded from the data list.

The unused analysis has also shown that the DATA DIVISION references the external file GAME that is never used during the execution of the program.

Data Accesses

After having isolated procedures and data items, the code has been analysed in order to examine the type of data access to each procedure. This process will also allow drawing the bipartite graph. In summary, the three types of data accesses are recalled below:

- **data using** (in the procedure there are one or more statements executing reading accesses);
- **data creation** (in the procedure the first statement accessing the data is a writing access);
- **data manipulation** (in the procedure there is at least a statement with a reading access followed by at least a statement with a writing access).

As the data access also depends on the category of accessed data (records, for example, in a statement with writing accesses, it should be checked to see if
the statement refers the entire structure of the record or only some elementary sub-fields. In the first case, the operation is a data creation. Otherwise, the operation is a manipulation, because the final content of data partially depends on the previous value.

For our purposes, then, the COBOL data have being subdivided in two categories - variables and records - depending on how the procedures refer to them. A data item, structured in a hierarchy of elementary fields, can be considered as a record only if there is at least a procedure directly referring to one of its elementary fields. All the other data items are variables.

<table>
<thead>
<tr>
<th>FLOW NAME</th>
<th>EXTERNAL FILE NAME</th>
<th>RECORD AREA NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIDEO-BONDS</td>
<td>VBTYPEIN</td>
<td>(COBOL external file)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\VIDEO-BOND-RECORD</td>
</tr>
<tr>
<td>GAME-MASTER</td>
<td>GAMEMSIO</td>
<td>(COBOL external file)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\GAME-MASTER-RECORD</td>
</tr>
<tr>
<td>PLAY-MASTER</td>
<td>PLAYMSIO</td>
<td>(COBOL external file)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\PLAY-MASTER-RECORD</td>
</tr>
<tr>
<td>STATION-RANKING</td>
<td>STRANKIO</td>
<td>(COBOL external file)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\RANK1-STAT-RANK-RECORD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\RANK2-STAT-RANK-RECORD</td>
</tr>
<tr>
<td>PP100A-REPORT</td>
<td>PP100AOU</td>
<td>(COBOL external file)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PP100A-LINE</td>
</tr>
</tbody>
</table>

This peculiarity affects the data accesses of those statements accessing the external files. The statement <READ data-file-name> obtains, in the
corresponding record area, the copy of one or more records from the file 
data-file-name. As the content of the file is unchanged, the access to the 
file is a using access. The access to the record area is a creating access, because 
the statement accesses the whole record. Statements such as 
<REWRITE 
record-name> have the effect of a copy of the content of record-name in 
the file as a new record (or overwriting the previous record). The content of 
external file will be changed after execution of the (RE)WRITE statement, and the 
content of that record-name is undefined after the successful execution of the 
(RE)WRITE statement; thus the operation induces a manipulation access in the 
external file and in the corresponding record area. The statement 
<DELETE 
data-file-name> deletes one record from the data file data-file-name, 
without accessing the corresponding record area. This statement thus produces a 
manipulation access only in the database. It is trivial that the statements 
described below accesses the whole record.

In many COBOL legacy systems, only the statements accessing the external 
files also access the record area. In this case (unfortunately it is not possible to 
generalise it), it is possible to ignore the access to the record area in these 
statements. Consequently, the record area can be removed from the data list.

Another important peculiarity is that concerning statements as 
<START 
data-file-name>, <OPEN data-file-name> and <CLOSE 
data-file-name>. Neither the file nor the corresponding record area is 
fected by the execution of these statements, thus they should be considered as 
amalous. In our example, as usual, all the OPEN and CLOSE statements are 
cluded in spare procedures. These procedures generating coincidental 
connections can be treated in a preliminary phase.

In our example, as the procedure B300 contains only CLOSE statements, it 
is excluded from the bipartite graph shown in figure 5.6. The procedure B100 
does not contains only OPEN statements, but also a single statement manipulating 
a data item, but due to the great number of coincidental connections, it is 
excluded from the bipartite graph as well.
5.3.3 The Bipartite Graph

The perform graph of the system is shown in figure 5.5. In it, as in the following objects isolation process, the procedures not accessing any data, as B220 and C230, have being ignored. The links starting from these procedures have being substituted by links from the invoking procedures A200 and B210, respectively. The procedures B100 and B300 have being ignored as well, because they contain the statements OPEN and CLOSE, respectively.

The bipartite graph is represented in figure 5.6. The dashed edges represent those using accesses to be removed.
Figure 5.6 -
Chapter 5

Computation of Indexes IC( ) and ΔIC( )

From the bipartite graph, the indexes IC( ) and ΔIC( ) can be computed. In our example, the values of these functions at the first stage are

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>IC( )</th>
<th>ΔIC( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0.0909</td>
<td>0.0000</td>
</tr>
<tr>
<td>A10H</td>
<td>0.1667</td>
<td>0.0758</td>
</tr>
<tr>
<td>A10F</td>
<td>0.1375</td>
<td>0.0470</td>
</tr>
<tr>
<td>A200</td>
<td>0.1613</td>
<td>-0.0407</td>
</tr>
<tr>
<td>A300</td>
<td>0.3864</td>
<td>0.1843</td>
</tr>
<tr>
<td>B200</td>
<td>0.3273</td>
<td>0.0141</td>
</tr>
<tr>
<td>B210</td>
<td>0.1515</td>
<td>0.0606</td>
</tr>
<tr>
<td>B230</td>
<td>0.1111</td>
<td>0.0000</td>
</tr>
<tr>
<td>C200</td>
<td>0.2867</td>
<td>0.0787</td>
</tr>
<tr>
<td>C210</td>
<td>0.2963</td>
<td>0.0343</td>
</tr>
<tr>
<td>C220</td>
<td>0.3077</td>
<td>0.0567</td>
</tr>
<tr>
<td>C240</td>
<td>0.3000</td>
<td>0.1889</td>
</tr>
<tr>
<td>C250</td>
<td>0.2052</td>
<td>0.1142</td>
</tr>
<tr>
<td>C260</td>
<td>0.3922</td>
<td>0.1903</td>
</tr>
<tr>
<td>C270</td>
<td>0.2292</td>
<td>0.2272</td>
</tr>
<tr>
<td>C280</td>
<td>0.2286</td>
<td>0.2265</td>
</tr>
<tr>
<td>D240</td>
<td>0.2162</td>
<td>0.0142</td>
</tr>
<tr>
<td>D250</td>
<td>0.2541</td>
<td>0.1830</td>
</tr>
<tr>
<td>D280</td>
<td>0.1892</td>
<td>0.0962</td>
</tr>
<tr>
<td>E250</td>
<td>0.0909</td>
<td>0.0000</td>
</tr>
<tr>
<td>F100</td>
<td>0.1111</td>
<td>0.0000</td>
</tr>
<tr>
<td>U100</td>
<td>0.1786</td>
<td>0.0876</td>
</tr>
<tr>
<td>U110</td>
<td>0.4412</td>
<td>0.3503</td>
</tr>
<tr>
<td>U220</td>
<td>0.2750</td>
<td>0.0729</td>
</tr>
<tr>
<td>U300</td>
<td>0.3584</td>
<td>0.1564</td>
</tr>
<tr>
<td>U400</td>
<td>0.2727</td>
<td>0.2727</td>
</tr>
<tr>
<td>2999</td>
<td>0.1613</td>
<td>-0.0407</td>
</tr>
</tbody>
</table>

Table 5.4

By examining these indexes it is rather difficult to establish the procedure to be clustered, because there are many procedures whose indexes are similar in size.

5.4 Data Refining Phase

From the bipartite graph of figure 5.6, it is possible to note the great number of edges leading to data items such as WSW, DT1 and WSV. A data refining phase can help the human expert to refine the granularity level of that datum that has a large number of accesses. With the support of the knowledge of an human expert, the datum is analysed to check if all the information belongs to the same object of the application domain, and if the procedures share the same...
information about the same object. If these conditions do not hold, then the datum can be refined into several more meaningful data items, each of them containing the information needed for a smaller number of procedures.

The effects of the refinements of WSW, DTI and WSV by increasing each structured elementary sub-field to a upper level, on the indexes IC( ) and ΔIC( ) are shown in the table 5.5 The decisions on which data item is more worthy of refinement should also be supported by the knowledge of the application domain. To introduce such a domain knowledge is not the aim of this presentation. Then some refining criteria based on the benefits caused on the following application of the algorithm will be examined. In general, the decision criteria for the refining phase will depend on the vectors of indexes ΔIC( ) before and after the refinement of a data item.

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>IC( )</th>
<th>ΔIC( )</th>
<th>IC( )</th>
<th>ΔIC( )</th>
<th>IC( )</th>
<th>ΔIC( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0.25000000</td>
<td>0.00000000</td>
<td>0.0909091</td>
<td>0.00000000</td>
<td>0.0909091</td>
<td>0.00000000</td>
</tr>
<tr>
<td>A10H</td>
<td>0.83333333</td>
<td>0.83333333</td>
<td>0.1666667</td>
<td>0.0757576</td>
<td>0.1666667</td>
<td>0.0757576</td>
</tr>
<tr>
<td>A10F</td>
<td>0.3076923</td>
<td>0.0576923</td>
<td>0.1379310</td>
<td>0.0470219</td>
<td>0.1379310</td>
<td>0.0470219</td>
</tr>
<tr>
<td>A200</td>
<td>0.3000000</td>
<td>-0.0611111</td>
<td>0.2250000</td>
<td>0.1340909</td>
<td>0.2250000</td>
<td>0.1340909</td>
</tr>
<tr>
<td>A300</td>
<td>0.3571429</td>
<td>0.2460319</td>
<td>0.4339623</td>
<td>0.3430532</td>
<td>0.3863636</td>
<td>0.1843434</td>
</tr>
<tr>
<td>B200</td>
<td>0.2571429</td>
<td>0.0349207</td>
<td>0.3750000</td>
<td>0.1729798</td>
<td>0.3090909</td>
<td>0.1070707</td>
</tr>
<tr>
<td>B210</td>
<td>0.2307692</td>
<td>0.2307692</td>
<td>0.1379310</td>
<td>0.0470219</td>
<td>0.1379310</td>
<td>0.0470219</td>
</tr>
<tr>
<td>B230</td>
<td>0.1111111</td>
<td>0.0000000</td>
<td>0.1515152</td>
<td>0.0606061</td>
<td>0.1515152</td>
<td>0.0606061</td>
</tr>
<tr>
<td>C200</td>
<td>0.2790698</td>
<td>0.1679587</td>
<td>0.2258065</td>
<td>0.1348974</td>
<td>0.2807018</td>
<td>0.0786816</td>
</tr>
<tr>
<td>C210</td>
<td>0.2058824</td>
<td>0.0947713</td>
<td>0.2407407</td>
<td>0.1983166</td>
<td>0.2962963</td>
<td>0.0942761</td>
</tr>
<tr>
<td>C220</td>
<td>0.2500000</td>
<td>0.1388889</td>
<td>0.2307692</td>
<td>0.1938601</td>
<td>0.3076923</td>
<td>0.1056721</td>
</tr>
<tr>
<td>C240</td>
<td>0.3000000</td>
<td>0.1888889</td>
<td>0.1888889</td>
<td>0.1888889</td>
<td>0.2500000</td>
<td>0.2500000</td>
</tr>
<tr>
<td>C250</td>
<td>0.3333333</td>
<td>0.0000000</td>
<td>0.2651282</td>
<td>0.1142191</td>
<td>0.2651282</td>
<td>0.1142191</td>
</tr>
<tr>
<td>C260</td>
<td>0.298065</td>
<td>0.1146954</td>
<td>0.2264704</td>
<td>0.1146954</td>
<td>0.2264704</td>
<td>0.1146954</td>
</tr>
<tr>
<td>C270</td>
<td>0.2500000</td>
<td>0.1388889</td>
<td>0.2291667</td>
<td>0.1903264</td>
<td>0.3076923</td>
<td>0.1056721</td>
</tr>
<tr>
<td>C280</td>
<td>0.4166667</td>
<td>0.0277777</td>
<td>0.2285714</td>
<td>0.0265512</td>
<td>0.2000000</td>
<td>0.1090909</td>
</tr>
<tr>
<td>D240</td>
<td>0.2173913</td>
<td>0.1062802</td>
<td>0.2285714</td>
<td>0.0265512</td>
<td>0.2000000</td>
<td>0.1090909</td>
</tr>
<tr>
<td>D250</td>
<td>0.2941176</td>
<td>0.1830065</td>
<td>0.2941176</td>
<td>0.1146954</td>
<td>0.2941176</td>
<td>0.1146954</td>
</tr>
<tr>
<td>D280</td>
<td>0.2941176</td>
<td>0.2941176</td>
<td>0.1891892</td>
<td>0.0982801</td>
<td>0.1891892</td>
<td>0.0982801</td>
</tr>
<tr>
<td>E250</td>
<td>0.3333333</td>
<td>0.0000000</td>
<td>0.2941176</td>
<td>0.1146954</td>
<td>0.2941176</td>
<td>0.1146954</td>
</tr>
<tr>
<td>F100</td>
<td>0.1111111</td>
<td>0.0000000</td>
<td>0.1111111</td>
<td>0.0000000</td>
<td>0.1111111</td>
<td>0.0000000</td>
</tr>
<tr>
<td>U100</td>
<td>0.4000000</td>
<td>0.4000000</td>
<td>0.2250000</td>
<td>0.1340909</td>
<td>0.2250000</td>
<td>0.1340909</td>
</tr>
<tr>
<td>U110</td>
<td>0.5625000</td>
<td>0.5625000</td>
<td>0.4411765</td>
<td>0.3502674</td>
<td>0.4411765</td>
<td>0.3502674</td>
</tr>
<tr>
<td>U220</td>
<td>0.2727273</td>
<td>-0.0277777</td>
<td>0.2285714</td>
<td>0.0265512</td>
<td>0.2285714</td>
<td>0.0265512</td>
</tr>
<tr>
<td>U300</td>
<td>0.2422442</td>
<td>0.1211111</td>
<td>0.2291667</td>
<td>0.1903264</td>
<td>0.2291667</td>
<td>0.1903264</td>
</tr>
<tr>
<td>U400</td>
<td>0.3333333</td>
<td>0.3333333</td>
<td>0.2727273</td>
<td>0.2727273</td>
<td>0.2727273</td>
<td>0.2727273</td>
</tr>
<tr>
<td>2999</td>
<td>0.3333333</td>
<td>0.2222222</td>
<td>0.1612903</td>
<td>-0.0407299</td>
<td>0.1290323</td>
<td>0.0381232</td>
</tr>
</tbody>
</table>

Table 5.5

From the bipartite graph shown in figure 5.6 it is possible to note that the system with WSW refined is more balanced than it was before the refinement.
In our case, by refining \( WSW \), we obtain the \( IC(\) \) vector shown in the 2\(^{nd}\) column of table 5.5 and \( \Delta IC(\) \) vector shown in the 3\(^{rd}\) column of table 5.5. The vectors of indexes \( IC(\) \) and \( \Delta IC(\) \) with the refinement of the data item \( DT1 \) are shown in the 4\(^{th}\) and in the 5\(^{th}\) column of table 5.5, respectively. Finally, in the 6\(^{th}\) and in the 7\(^{th}\) column, the vectors of indexes relative to the refinement of \( DT1 \) are shown. Note that after the refinement of \( WSW \), the index \( \Delta IC(A10H) \) is passed from \( +0.0730435 \) to \( +0.8333333 \), then isolating \( A10H \) for the clustering.

After refining and clustering, the system must be reanalysed and the bipartite graph redrawn. The modified bipartite graph is shown in figure 5.7.
Chapter 5

a case study on a COBOL source code

Figure 5.7

87
Note that the refinement of a record (a data whose elementary fields are accessed directly by the statements in the code), can turn it into a variable. Then the statements referring it have to be reviewed in order to decide the access type.

5.4.1 Data Duplication

The aim of this phase is to break off the "logical links" that the common access to a global data creates between the procedures accessing them. In many cases, common accessed data can be treated as "local" relative to different groups of procedures. When the same data is local to different single procedures (for example the variable i usually used as counter in the loops) a trivial intervention is to "split" it by renaming with a different name in any procedure accessing it.

The main idea of the data duplication phase is to isolate groups of procedures in which a data can be considered as local and then duplicate this data item into many items by renaming it. All those groups are characterised by having a procedure creating the data and, the value of the data in the other procedures of the group depends by the computation of the procedures of the same group already executed and accessing the data. More formally, for any data A, the aim is to identify groups of procedures with the following features:

- there is one and only one procedure linking an information node creating A (the information node and the procedure node will be called I₀ and P₀ in the following examples);
- in each procedures Pₖ of the group, the value of A only depends on I₀ and on other procedures of the group which are in the path from I₀ to Pₖ;
- if two procedures Pₖ and P₇ belong to the same group, then all the procedures in the path of the temporal graph between Pₖ and P₇ belong to that group.

Informally, the technique defines when the value of a fixed data item A in a procedure derives from the execution of a portion of code of another procedure. Besides, it is defined when a procedure Pₖ directly derives data A by I₀ if in any traversal of the temporal graph between I₀ and the first information node Iᵦ of Pₖ (inclusive), the node I₀ is the only one creating A. These definitions, have many
similarities with the data dependence of Ottenstein et al. [FOW84] [OO83], and will be used in order to identify the procedures $P_k$ in which the value of $A$ depend on $I_0$.

In order to proceed to the data duplication phase, all of the groups of procedures relative to data must be identified. For a data item $A$, a group of procedures is composed by the procedure $P_0$ creating it and all the procedures $P_k$ directly and exclusively deriving $A$ from $I_0$ identified through the definitions above, and all the procedures not accessing $A$ in each path leading from $I_0$ to any procedure $P_k$ of the group.

The aim of this paper is just to explain how the augmented technique can be "adapted" to a programming language as COBOL. As there are no special arrangements for this phase regarding the peculiarity of COBOL, the attention will be concentrated on the drawing of the "temporal perform graph" (TPG), combining the features of control flow graph and of the PERFORM GRAPH.

A simple example on the creation of the TPG relative to the main procedure of our example is given in figure 5.8.iii. The corresponding code and control flow graph are shown in figure 5.8.i and 5.8.ii, respectively. Due to the UNTIL statement, the procedure A200 can be performed zero or more times. Note that the analysis for drawing the TPG is still a static analysis.

The TPG is built through a data analysis in which every statement of the code is examined in order to check how every procedure accesses the data and the sequence of the PERFORM statements affecting the data.

In order to obtain a representation which is easy to handle, instead of drawing an unique TPG representing all the data, several TPGs can be drawn, each of them relative to an unique data item. Since the data duplication phase examines the data flow and the temporal sequence of the PERFORM statements only relatively to a data item per time.

A simple example of TPG relative to a small portion of code is represented in figure 5 9. In it, all the references of the procedures to the data are shown
Figure 5.8 - The program representation in case of a UNTIL statement. These statements are typical of the COBOL programming style. The "*" in figure 5.8.iii represents the fact that A200 can be repeated more than one time.

Figure 5.9 - The TPG shown in figure 5.9 is not relative to the initial stage of the technique. It is possible to note the references to the refined data. For sake of clearness, the procedure B220 and C230 are still represented in the TPG. Remember that they were excluded from the bipartite graph, as they do not access any data.

A common situation met during the duplication phases of this system is shown in figure 5.10a. Procedure P9 contains more than one information node.
creating data item A, then it belongs to more than one group. In this case, the procedure will contain references to all the items in which A will be duplicated. For example, in figure 5.10a the data item A will be duplicated into two items A₁ and A₂, as shown in figure 5.10b, because of the identification of two different groups relative to A. The portion of code relative to I₀₁ and I₀₂ will contain references to A₁ and A₂ respectively. The procedure P₀ will belong to the two groups.

Figure 5.10 - The procedure P₀ has many information nodes creating A. In this case, the procedures P₁ and P₅ directly derive A from I₀₁ and I₀₂, respectively. Procedure P₄ derives A from I₀₁, but this derivation is not direct. With the further assumption that there are no other procedures invoking the procedures in figure 5.10a, the derivation is also exclusive. In this situation, the procedure will contain the different items in which the data is duplicated, i.e. the several portions of source code corresponding to the information nodes I₀₁ of P₀ contain different variables derived by the duplication of A

Through a static analysis of the code such groups of procedures are identified, and a data duplication phase “splits” the data into many items, one for each group. This process is carried out for each data item.

After this phase, the links starting from these procedures toward the commonly accessed data form a group of procedures whose computation depends on a common data item. In this case, all the procedures of that group can be considered as methods of the same object whose state is also stored in the common accessed data.
Chapter 6

Future Works

The data analysis sketched here is also aimed at supporting the extraction of knowledge from the legacy software at a high abstraction level, thus reducing the effort of domain engineering. In order to support further the candidature phase of higher level modules, an accurate examination of the available documentation and the application of syntactic knowledge of the programming language to the source legacy code can enrich the knowledge about the legacy information system within that application domain.

The process produces a repository of reusable modules with object like features directly relevant to the application domain. Each module should be documented with specific information that can further support the process of designing an object oriented system in that application domain.

A parallel top-down process of forward engineering could be defined in order to support further the bottom-up reengineering process described here. If the aim of this reengineering process is to reverse engineer the software system, it can be combined with a forward engineering process aimed to significantly increase both the understanding of the legacy code and the application domain knowledge, both of which are needed to extract object oriented features from the existing code.

A top-down forward engineering approach targeted to build from scratch reusable modules to be reused in the development of a new software system within the application domain needs a greater amount of time than a combined approach in order to obtain the first reusable module. This is due to the longer
time needed for the domain experts to produce an object oriented design for that application domain. However, in case of a not very complex, or of a well known application domain, the top-down approach alone might be preferred.

A combined approach may identify software components in less time because it gains of the use of knowledge extracted from the legacy information system combined with the knowledge of the experts of that application domain.

### 6.1 Extension of Data Duplication Phase

In order to identify high quality objects-like modules, the addition of a forward engineering process can help to group together procedures that cannot be grouped solely on the basis of the data flow analysis. For instance, figure 6.1 shows the case of two procedures creating the same data. Let us suppose that both of them call a common procedure using that data (as for example in the case of a common subroutine), the data duplication phase cannot group them into a single group because of the presence of two information nodes creating the same data.

![Figure 6.1 - Two procedures creating the same data but logically related. Further knowledge from the application domain can help to group more meaningful candidate modules, thus reducing the effort of the concept assignment phase.](image)

Further knowledge from the application domain simplifying the
understanding of the data meaning can enrich the algorithm here presented, thus allowing the grouping of more meaningful candidate modules. This reduce the effort of the subsequent concept assignment phase.

6.2 Data Normalisation

A further phase can be formalised in order to break down the undesired links between two procedures accessing the same data. In the case shown in figure 6.2, both the procedures $P_k$ and $P_0$ access the data item $A$. In order to remove the link between $P_k$ and $A$ from the bipartite graph, it is possible to pass $A$ as parameter in all the invoking statements of the procedures between $P_k$ and $P_0$. This approach gives all the intermediate procedures access to $A$, but in the bipartite graph there are no links between them and the data item $A$. This solution has as negative aspect in that it increases list of parameters and, for the feature of the COBOL, the increase of the LINKAGE SECTION of all the intermediate procedures.

If it is supported by the knowledge extracted from the application domain, this approach can help to isolate more meaningful groups of procedures on the basis of the common accesses to the same data $A$, thus allowing the isolation of modules with object oriented features.
Chapter 7

Conclusions

Object technology is undoubtedly the most promising way of delivering systems based on reusable modules that can be adapted and changed without having to re-examine all the existing code minutely [G94]. Migrating towards an object-oriented platform is the best solution to protect the investments aimed to keep an operational information system. The target system is easier to understand and to modify and the behaviour of the agents simulating the real-world entity is easier to modify making the information system much more adaptable to the rapid nature of the commercial change, and simpler to migrate than it was before.

7.1 Evaluation of the Criteria for Success

The goals proposed at the beginning of the work have been developed through the outlining of the research and further defined at the stage of defining the method. By following the sequence of the stages in which the work aimed to formalise the method here presented, it is possible to evaluate the criteria for success proposed in the first chapter of this thesis.

The temporary sequence of these stages is basically respected through the schema of this thesis.

7.1.1 Description and evaluation of existing methods

The overview presented in the second chapter of this thesis has been
important because it depicts the state of the art about the reverse engineering and reengineering techniques targeted at intervention producing evolution of the code. A global overview of these language-independent techniques and their applications to case studies has been important in the process of a better understanding of all the problems related to the evolution of the legacy information systems, as well as in the process of formalisation of the definition of the technique here presented. Through that presentation, great attention has been given to those aspects directly suggesting some details in the formalisation of our method. Particularly, Sneed's work and the reverse engineering techniques, presented in section 2.2, directly dealing with COBOL source code have been important in the later stage of this work, as it was clear from some estimates about the large amount of COBOL legacy information systems the IT people are dealing with. It suggested checking the method to extract reusable modules from COBOL source code.

Section 2.3 has presented approaches dealing with the legacy information system problem while populating a repository of "spare parts" to be reused in the development of a new information system. By reading these works, the importance of the extraction of reusable components became clearer. In fact, as the functionality of "small" object-like modules is easy to understand and to handle, the isolation of object-like modules with an easier data structure enlarges the likelihood of reusing them rather than developing them from scratch as well as in each evolution process of the existing legacy information systems within the application domain. This has allowed us to promote reuse within a definition of a method targeted to help a process of evolution of existing legacy information systems.

7 1.2 Formalisation of a language-independent method

The approaches to extract objects-like modules based on the graph theory presented in section 2.5 have shown that graph theory may help to define an efficient program representation by adding more information about the kind of data access.
In fact, the main idea of the object isolation method presented here is to extract information from the data flow, to cluster all the procedures on the base of their data accesses. In order to distinguish several types of accesses and to permit a better understanding of the functionality of the candidate objects, graph theory allows the use of a program representation simplifying the process of understanding "how" a procedure accesses the data.

Particularly, great importance has been given to the data structures in each iterative step of the algorithm presented here. This importance came from the conviction that it would be useful to the maintenance programmer to understand the data and the function relationship and objects the original developer had in mind. Clustering and reengineering operations on the components belonging to a candidate object are necessary to transform them into an actual reusable object. Furthermore, successful maintenance requires a precise knowledge of the data items in the system, the way these items are created and modified and their relationships.

7.1.3 Application of the method to a case study

The aim of the application of the technique to a real example is not to give an exhaustive explanation about the method above, using a real example, but to show how to adapt the method to the peculiarity of a given programming language such as COBOL, while respecting in the meantime the main ideas of the technique. By way of a case study, a simple COBOL program has been analysed, and the technique being used on it. All the necessary arrangements to adapt it to the peculiarity of COBOL are underlined.

Unfortunately, an information system that is modern today, will be legacy tomorrow. It is impossible to conceive information systems that do not turn legacy "...we can use the best methods and the most modern tools in order to reduce their resistance to the necessary changes, and we can build them in a
modular way such that make easier each change but we can not foretell the future requirements of a business or the progress in the technology these two things can challenge us as we can not foretell, thus making greater the future resistance of our legacy information system to the changing” [BS95]
REFERENCES


[C92] T Capers-Jones, "Geriatric Care for Ageing Software", Knowledge Based 1, Software Productivity Research Inc , Burlington, 1992


106


Revised By A. B. Mikel and J. F. Miner.


[SPC93] SPC Services Corp, "Reuse-Driven Software Process Guidebook", SPC-92019-CMC, Version 02 00 03, November 1993


P WINSBERY, "Legacy Code - Don't Reengineer it, Wrap it", Datamation, pp. 36-41, May 1995


