UNIVERSITY OF DURHAM

DEPARTMENT OF COMPUTER SCIENCE

Master Thesis in Computer Science

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. sometime, 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

CONTENTS

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 became 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] [B91$_A$] [B81] [LS80]

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 [B93$_A$], who estimates that of $100 billion dollars annual expenditure of companies on software, at least 70% will be spent on maintaining their systems, while the other 30% will be spent on new development Consequently, companies are increasingly directing their efforts to get more from the existing systems but also to ensure that these systems are much more maintainable than it was originally demanded The existing information systems are today's "*assets*" to be protected [P95] [B91$_C$], 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] [ANSI83] [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*" [$V93_A$].

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*" [$MD91_{16}$] 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[1]. 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 [$MD91_{20}$]. 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*" [$B91_C$]. 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 [$MD91_{41}$].

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

1 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, a the decision about the evolution of a *legacy information system* should take into account not only the economic constrains - 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 [BMW96$_A$] [BMW96$_B$] 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 then procedural, imperative description*"

1 1 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*[1] *reengineering* [H90]. It is "*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 mode understandable and more maintainable [MS87] [MD91$_{20}$]. 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].

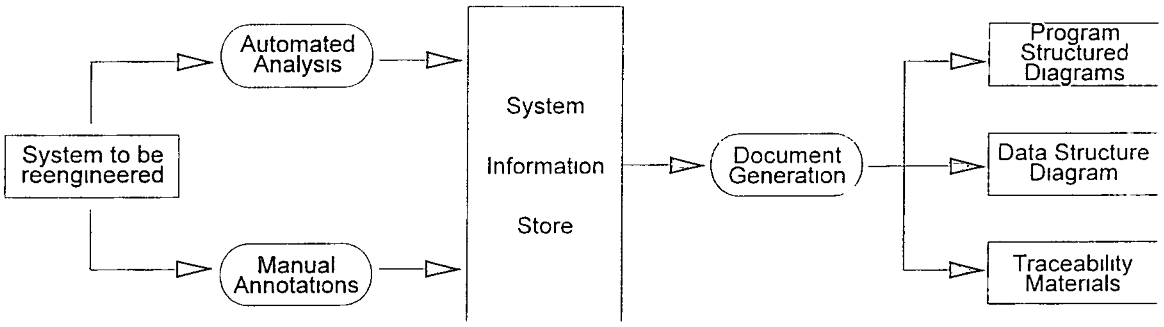

Figure 2 1 - The Reverse Engineering Process

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$_A$] the term has been used to denote a clustering construct generally providing [C89$_A$]

[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] [$S96_A$] [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 Dictorate 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 *de facto* standard of system analysis and design in the UK [EM93].

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

1. 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. 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 *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 *system network diagram* is produced The resulting *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 *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[3]. 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 GOTOs 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-rquest@prg oxford ac uk> with your name, address and e-mail address to join the mailing list

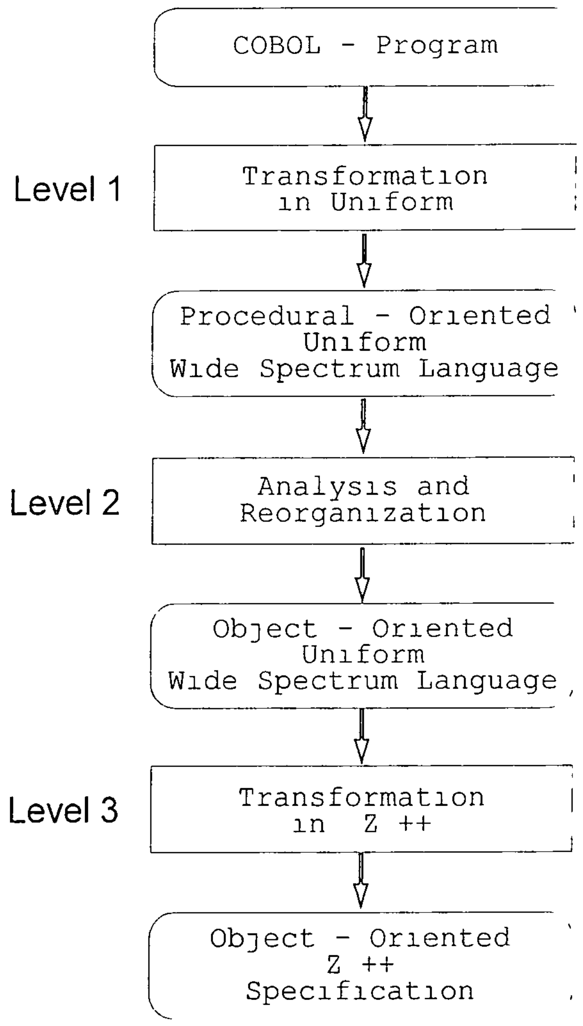
COBOL - Program
Level 1
Transformation in Uniform
Procedural - Oriented Uniform Wide Spectrum Language
Level 2
Analysis and Reorganization
Object - Oriented Uniform Wide Spectrum Language
Level 3
Transformation in Z ++
Object - Oriented Z ++ Specification

Figure 2 2 - REDO reengineering process

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 [HL91$_B$] 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.

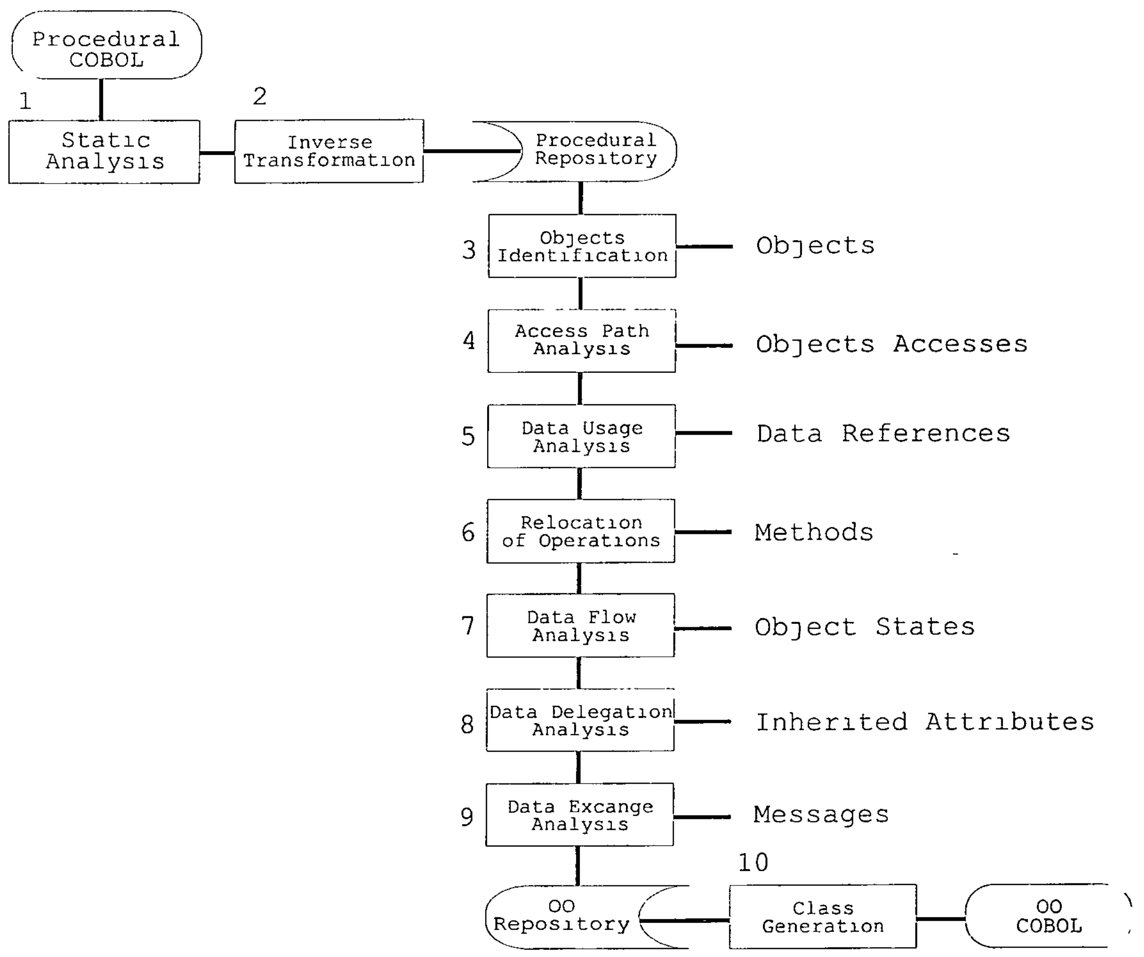

Figure 2 3 - REORG reverse engineering process

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-STORAGE SECTION. Local data are declared in the PRIVATE-STORAGE 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 wrapper[4] 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 that 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 [S96$_A$] 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*" [S96$_B$].

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,

[4] The term has being coniated by Wally Dietrich in the 1989 Many authors consider his intervention [DGN89] in the *Conference on Object Oriented Programming System, Languages and Application* as "*an original source on the object wrappers*"

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 [B87$_B$] 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

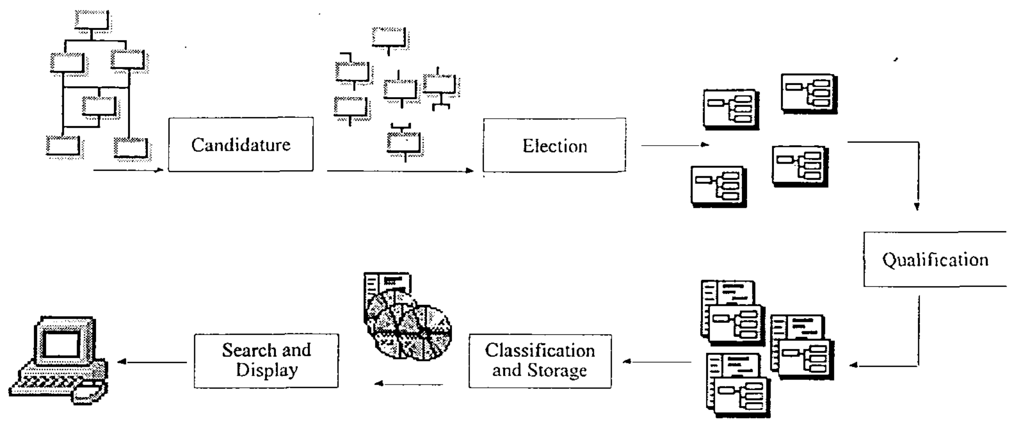

Figure 2.4 - The Reuse Reengineering (RE^2) Paradigm identifies in legacy source code *a set of software components each suitable to implement an abstraction of an entity in the application domain. Each component is candidate to be transformed in a reusable module. After that each candidate has been 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 RE^2 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.

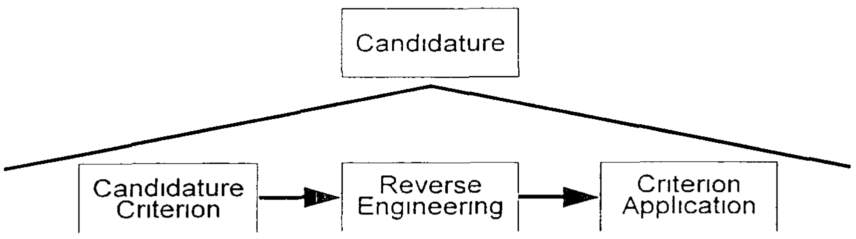

Figure 2 5 - The distinction of the activities in the Candidature phase leads to subdivide it in the sub-phases of "Candidature Criterion", "Reverse Engineering" *and* "Criterion Application"

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 undesired 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.

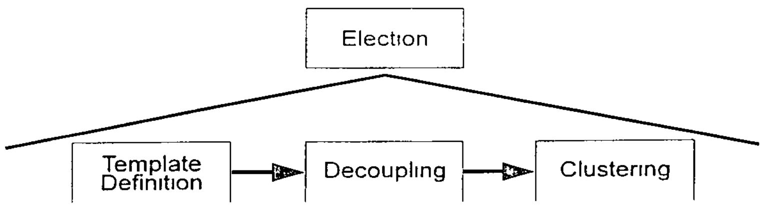

Figure 2 6 - The distinction of the activities into the election phase leads to the subdivision of the phase in the sub-phases "Template Definition", "Decoupling", "Clustering"

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.

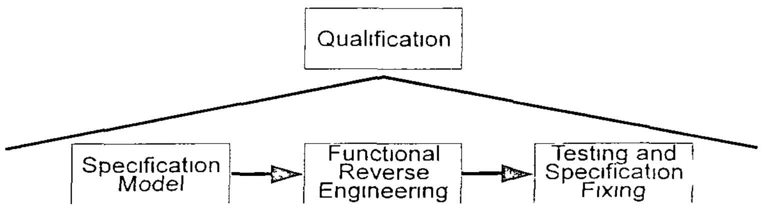
Qualification
Specification Model
Functional Reverse Engineering
Testing and Specification Fixing

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 RE^2 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^2 paradigm prominently as regards the candidature criterion within the candidature phase. A successful RE^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*[5] 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* [CD91$_A$], 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 an 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 Parnas[P72$_B$] 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], Muller 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 type[6]. 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 [$P72_A$]

links[7] 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 RE^2 project does not address control abstraction because it deals with existing software written in traditional languages that usually do not express concurrence 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*

[7] Some times those links are very weak, for example they can be only recognisible for the

system that relate directly to the new technique presented later in this thesis All the work presented propose candidature 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 $PP \times (PP-\{s\})$, 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 in 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 $t \in MET$ 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 $t \in UMET$ 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 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 $P(x_1)\cap P(x_2)\neq\varnothing$ Formally

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

$$E=\{(P(x_1), P(x_2)) \mid P(x_1)\cap P(x_2)\neq\varnothing\}$$

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

$$F = \bigcup_{P(x)\in V_c} P(x)$$

$$T = \varnothing$$

$$D = \bigcup_{P(x)\in V_c} \{x\}$$

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 $t_1 \ll t_2$, if t_1 is used to define t_2 - in this case t_2 is a super-type of t_1. Obviously, if $t_1 \ll t_2$, and $t_2 \ll t_3$, then $t_1 \ll t_3$

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 $t \ll t_1$ 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$_A$]

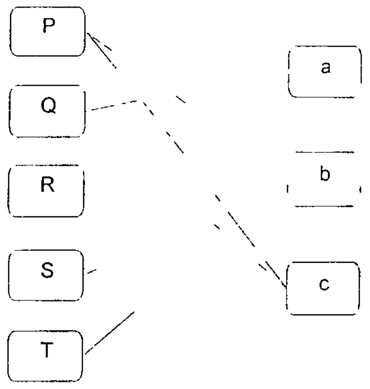

Figure 2 8 - An example of bipartite graph $G=(N, E)$, where $N\equiv(N_1 \cup N_2)$, with $N_1=\{P, Q, R, S, T\}$ and $N_2=\{a, b, c\}$ The set of edges $E=\{(P, b), (P, c), (Q, a), (Q, c), (R, a), (S, b), (T, b)\}$ contains undirected edges going from an element of one of the two subsets of N toward an element of the other subset

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 human intervention

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 items within a procedure For each node $n \in N$, the sets PreSet(n) and PostSet(n) are defined as

$\mathrm{PreSet}(n) = \{y \mid y \in N \wedge (y,n) \in E\}$,

$\mathrm{PostSet}(n) = \{y \mid y \in N \wedge (n,y) \in E\}$

Informally, the set PreSet(n_2), where n_2 is a node in N_2 (i e a data items), represents the set of all the procedure (nodes n_1 in the set N_1) referencing n_2. the set 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 $\mathrm{PreSet}(n_1)=\varnothing$, and for each $n_2 \in N_2$ the set $\mathrm{PostSet}(n_2)=\varnothing$

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[9] 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[10]. 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

$$\mathrm{IC}(P) = \frac{\sum_{A \in \mathrm{PostSet(P)}} \#\{\mathrm{P_i} \mid \mathrm{P_i} \in \mathrm{PreSet(A)} \wedge \mathrm{PostSet(P_i)} \subseteq \mathrm{PostSet}(P)\}}{\sum_{A \in \mathrm{PostSet(P)}} \#\{\mathrm{Preset}(A)\}}$$

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

$$\Delta\mathrm{IC}(P) = \mathrm{IC}(P) - \sum_{A \in \mathrm{PostSet(P)}} \frac{\#\{\mathrm{P_i} \mid \mathrm{PostSet(P_i)} = \{A\}\}}{\#\{\mathrm{Preset}(A)\}}$$

The routines having an index ΔIC(P) sufficiently high are used to generate cluster around P[11]

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] it 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 $P \in PROC$ DO
        COMPUTE INDEXES $IC(P)$ AND $\Delta IC(P)$
    END FOR
    COMPUTE THE STEP VALUE $STV$, AND THE SETS $MERGE$ AND $SLICE$
        $MERGE=\{P \mid \Delta IC(P) > STV\}$
        $SLICE=\{P \mid 0 < \Delta IC(P) \leq STV\}$
    INTERACTS WITH HUMAN EXPERTS TO DELETE FUNCTIONS FROM THE GRAPH
        AND/OR TO MOVE FUNCTION FROM THE MERGE TO SLICE AND VICE-VERSA
    FOR EACH FUNCTION $P \in MERGE$ 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$_A$].

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[13].

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

[13] These two conceps are definet similarly in *megaprogramming* [CWW]

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*" [$C89_A$]. 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[1] 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 [FO76$_A$], [FO76$_B$], [HHW76], [H86], [HR88], [HR89] resulting by some previously undiscovered program errors *Data flow analysis* is also fundamental for *program slicing* [W79] [W82] [W85] In fact, after having decomposed the program into *slices*, the analysis of *data flow* and *control flow graphs* [LR91] 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* [C89$_A$]): 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* [LPR91], providing information about data flow Since it is sensitive

[1] Depending on the aim of the data flow analysis, the region might represent a single statement or a large fragment of code

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 [FO76$_A$], [FO76$_B$]. 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
03    VAR int1, int2. INTEGER,
04
05    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 same 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 oı a varıable It ıs a varıable ıf (and only ıf) no sıngle statement dııectly refers to any of ıts elementary fıelds In other words ıf all the statements ın all the program components referrıng to the data aıe ınterested ın the entıı e data, then the data ıs a varıable

The dıstınctıon between varıables and records ıs ımportant ın the analysıs of statements accessıng the data ın wrıtıng, ın order to deteı mıne whether the access ıs eıther a creatıon or a manıpulatıon In the case of wrıtıng access to a records, for example, ıf the statement refers the entıre structure then the access ıs by creatıng If the statement refers only to some elementary of the elementary sub-fıelds, then the data operatıon ıs a manıpulatıon, because the fınal content of data partıally depends on the prevıous value stored ın the data structure

```
010900 01  NAME-CUSTOMER
011000      03  NAME
011100          07  QUALIFICATION              PIC  X(03)
011200          07  FIRST_NAME                 PIC  X(15)
011300          07  SURNAME                    PIC  X(15)
011400      03  BIRTH-DATE
011100          07  MM                         PIC  9(02)
011200          07  DD                         PIC  9(02)
011300          07  YY                         PIC  9(02)
011500      03  ADDRESS
011600          07  STREET                     PIC  X(15)
011700          07  LOCALITY
011800              11  CODE                   PIC  X(7)
011900              11  CITY                   PIC  X(15)
012000          07  COUNTRY                    PIC  X(15)
```

Fıgure 3 2 - Example of COBOL source program defınıng a record ın the DATA DIVISION *If all procedures only access ıt by refeı rıng to* NAME-CUSTOMER, *than ıt can be consıdeı ed a varıable*

Fıgure 3 2 show the defınıtıon of a COBOL record NAME-CUSTOMER defıned wıthın the DATA-DIVISION Supposıng that thıs record ıs paıt of a database of an archıve If a statement replaces the value only of the group of elementary fıelds ADDRESS, the access cannot be consıdered a creatıon of all the data ıtem NAME-CUSTOMER as the value of the whole data partıally depends on the prevıous value of the executıon of the statement, as depıctıng a data manıpulatıon In fact, wıthın an applıcatıon domaın usıng thıs record, an upgıade of thıs data ıtem ıs due to the creatıon of a sub-fıeld, and does not represent the creatıon of the whole data, but ıts manıpulatıon

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 that once.

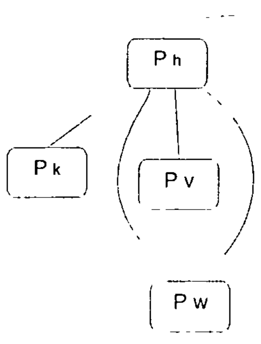

Figure 3 3 - Between P_h and P_w, there are two invoking edges *The sequence of the invoking statements is the sequence of the edges anticlockwise*

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[3]

```
IF <condition> THEN <"write A"> ELSE <"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,

[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

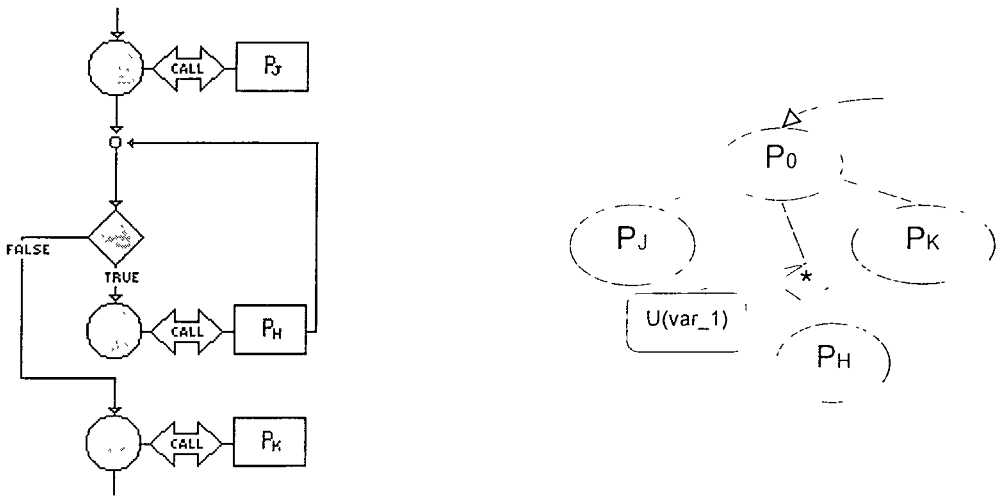

Figure 3 4 - Program representation in case of a loop statement like a WHILE The "" in figure 3 4 ii represents the fact that P_h can be repeated more than once, depending on the value of var_1 The first operation is reading the value of var_1, then only if its value satisfies a condition, procedure P_h is performed At each iteration of the loop, the value of var_1 is read*

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 *tempoial giaph* each walk going back from I_h to the starting node *s* If there is a dependence of A on itself, then the statement peiforms 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

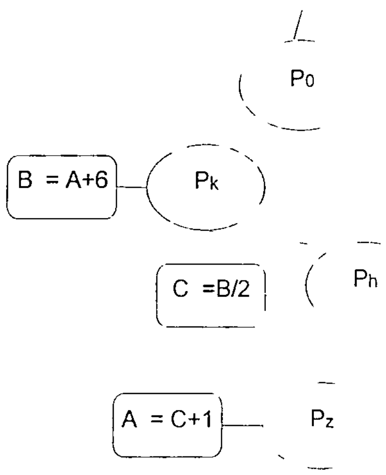

Figure 3 5 - Figure 3 5 shows the case of an indirect dependence of A on itself In this case the statement <B =A+6> induces a dependence of B on A in the pioceduie P_k i e B=f(A) In the procedure P_h the statement <C =B/2> induces a dependence of C on B and then on A i e C=f(B)=f(A) At least in the pioceduie P_z the statement <A =C+1> induces a dependence of A on itself due to the fact that C=f(A) thus depicting a data manipulation on A

In the literature there exists a number of techniques treating the restructuring processes performed in order to make procedural code modular and well structured, thus facilitating the decomposition process of the code into a set of module 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 In the following, it will be supposed that the code is modular and well structured, and that there exist some GOTOs within the code, but they are just address the control to the end of the procedure thus generating the return of the control to the invoking procedure

With these suppositions, the nodes here introduced to represent the *tempoial* *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

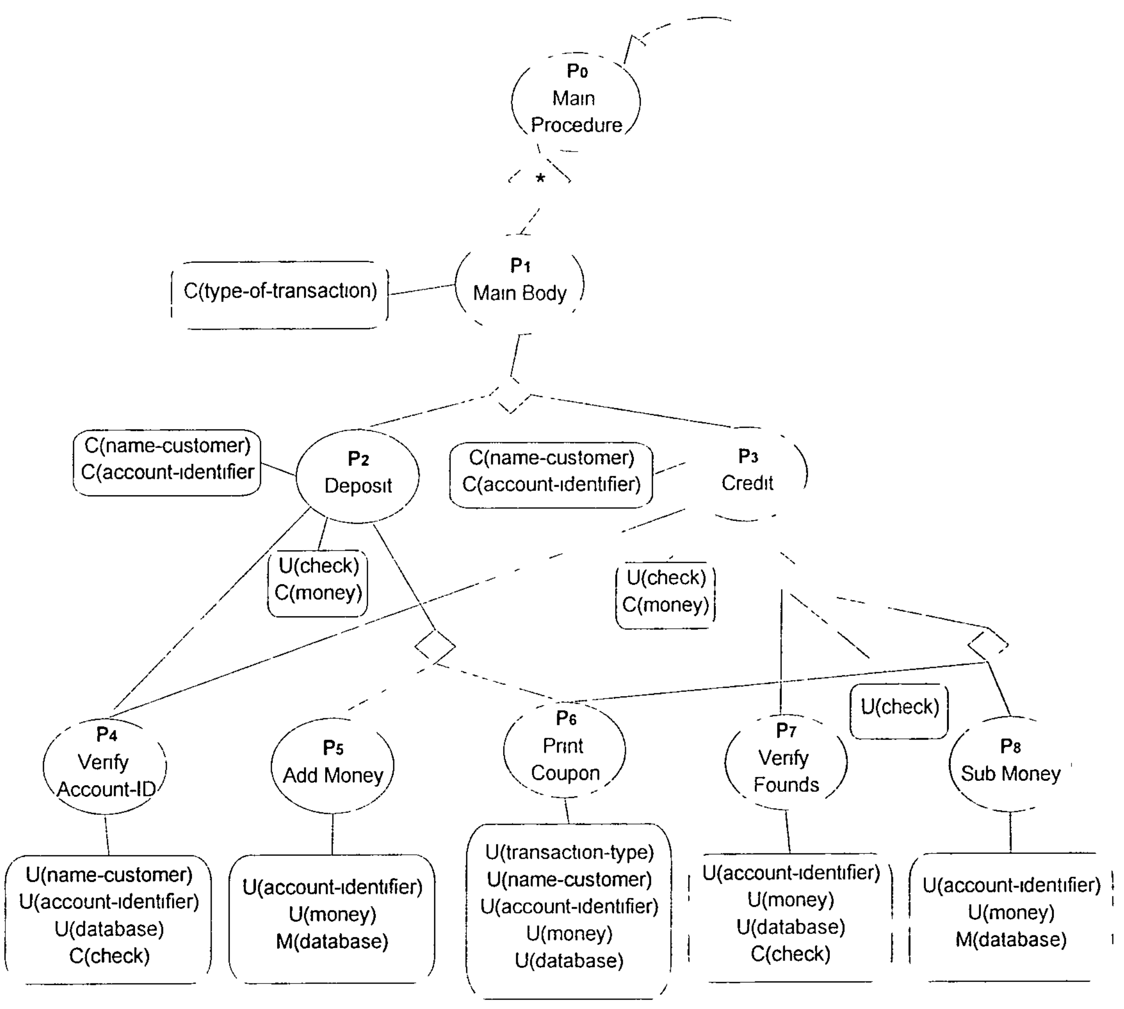

Figure 3 6 - The figure above shows the partial temporal graph *representing some procedures in a typical Bank Account Management System The data* database *represents a file whose records contain information about the customer name (*name-cust*), the account number (*account-ID*), the money present 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 an high level of abstraction Many detail 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 inducts 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 *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.

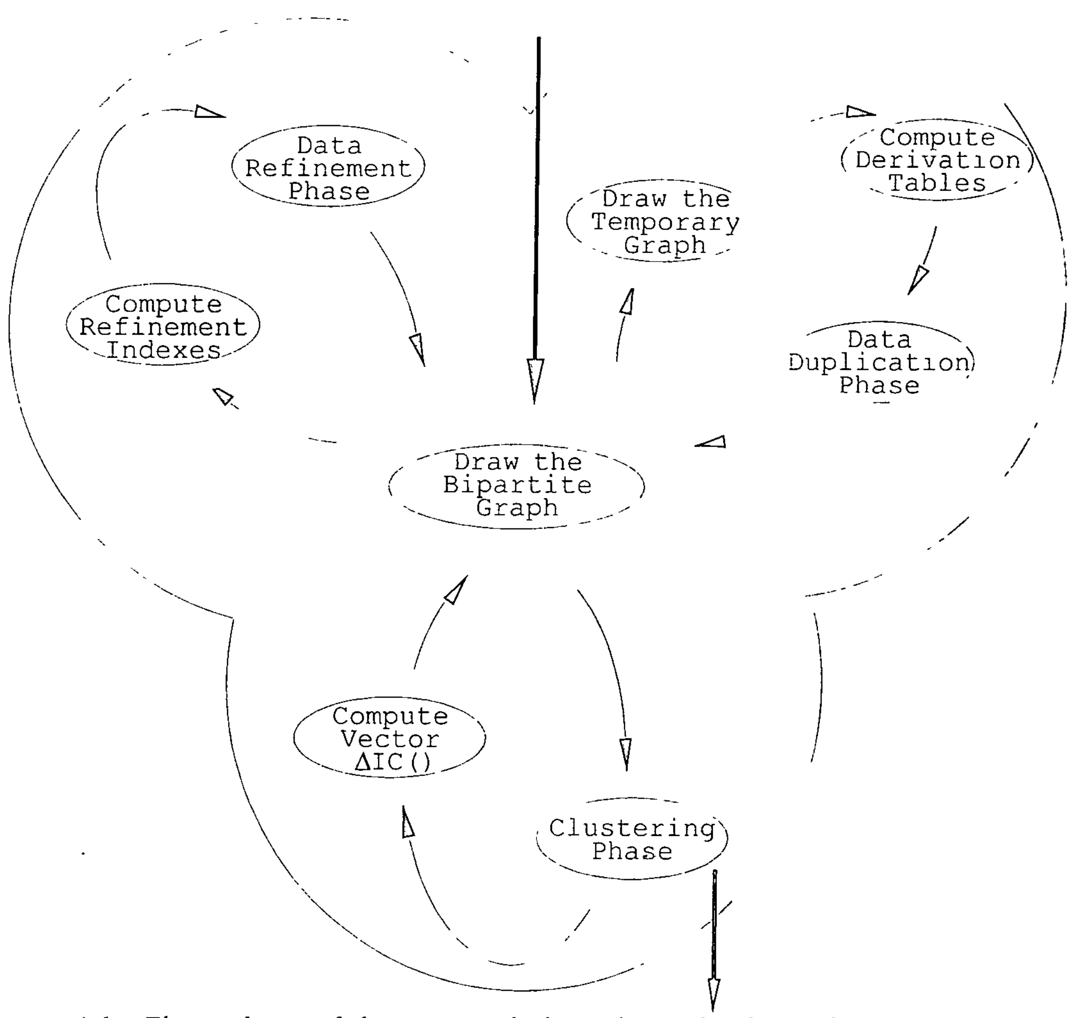

Figure 4.1 - The outlines of the improved algorithm. The three phases represented in figure modify the bipartite graph until it is decomposed in a set of disjoint sub-graphs

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 ΔIC( )*

At each iterative step, the modified algorithm examine the bipartite graph in order to compute, for each procedure P, a vector of indexes $\Delta\mathrm{IC}(P)$:

$$\Delta\mathrm{IC}(P) = \frac{\sum_{A \in \mathrm{PostSet(P)}} \#\{P_i \mid P_i \in \mathrm{PreSet(A)} \wedge \mathrm{PostSet}(P_i) \subseteq \mathrm{PostSet}(P)\}}{\sum_{A \in \mathrm{PostSet(P)}} \#\{\mathrm{Preset}(A)\}} - \sum_{A \in \mathrm{PostSet(P)}} \frac{\#\{P_i \mid \mathrm{PostSet}(P_i) = \{A\}\}}{\#\{\mathrm{Preset}(A)\}}$$

The aim of the computation of the vector ΔIC( ) 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 ΔIC(*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 ΔIC(*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 ΔIC(), 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_h before the statement PERFORM P_k and it is used in P_k before A is set again Analogously, a *output parameter* is the data item A is set within a performed procedure P_k 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.

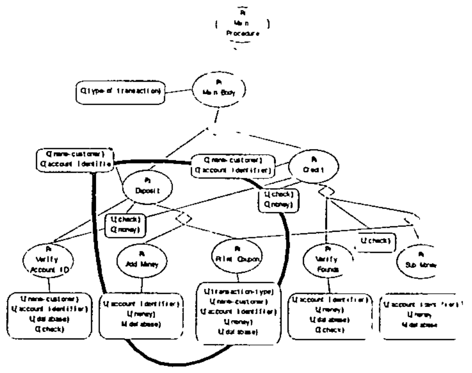

Figure 4 2 - A typical temporal graph *with the specification of a group of procedures in which a data can be considered local*

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 $I_0 \in IN$ be an *information node* creating data item A; let $P_0 \in PN$ be a *procedure node* such that $(P_0, I_0) \in E$ (I_0 of P_0). Let I_{k_1} be an *information node* of P_k using or manipulating data A. Then I_{k_1} of P_k *derives* A from I_0 if there is a path between s and I_{k_1} (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 $I_0 \in IN$ and $P_0 \in PN$ 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_1} due to the presence of *region nodes*. In definition 4.1, the condition that I_{k_1} 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_{h1} of P_h. I_{j2} only derives A from I_{j1} and I_{h1} of P_h.

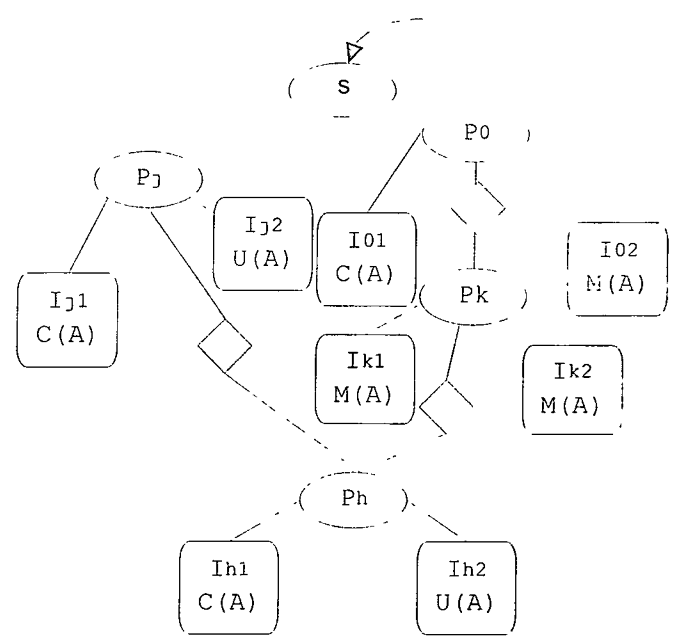

Figure 4 3

Definition 4.2 - Let *G=(s, N, E)* be a *temporal graph.* Let I_{k_1} 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_1} (included) does not contain *information nodes* creating A, then I_{k_1} 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 of P_k *directly derives* A by I_0 if in any traversal of the *temporal graph* between I_0 to I_k (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 (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_{k2} have a *directly derive* A from I_{01} and I_{h1}.

If an *information node* I_k 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_1} be an *information node* of P_k *directly deriving* data A by I_0 of P_0. Then I_{k_1} of P_k *exclusively derives* A by I_0 of P_0 if there not exists any other *information node* I_h

of P_h which I_{k1} *directly derives* A from. ☑

The word "*exclusive*" referred to the derivation is due to the fact that P_k derives A only (exclusively) by I_0 of P_0. This ensures that the value of A in P_k depends on the execution of I_0.

Note that, by definition 4.3, in figure 4.3, there are only two examples of *exclusive derivation*: I_{h2} and I_{k1} *exclusively derive* A from I_{h1} and I_{01}, respectively.

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

Information node	derivation	direct derivation	exclusive derivation
I_{01}	-	-	-
I_{02}	I_{j1}, I_{h1}, I_{01}	I_{h1}, I_{01}	-
I_{j1}	-	-	-
I_{j2}	I_{j1}, I_{h1}	I_{j1}, I_{h1}	-
I_{k1}	I_{j1}, I_{h1}, I_{01}	I_{01}	I_{01}
I_{k2}	I_{j1}, I_{h1}, I_{01}	I_{h1}, I_{01}	-
I_{h1}	-	-	-
I_{h2}	I_{j1}, I_{h1}, I_{01}	I_{h1}	I_{h1}

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_0 with an *information node* I_0 creating A, and all the procedures P_k *exclusively deriving* A from I_0. The procedures composing the group can be identified through the definitions above

The value of a data A in any procedures P_k (with $k > 0$) within a group relative to that data item, depends on I_0 and on all procedures in the path from P_0 to P_k 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 P_1 and the one of P_3 (figure 4 4a), *directly derive* A from I_{01} On the other hand, the *information node* of P_5 *directly derives* A from I_{02} With the further assumption that there are no other procedures invoking the procedures of figure 4 4a, the derivation is also *exclusive*

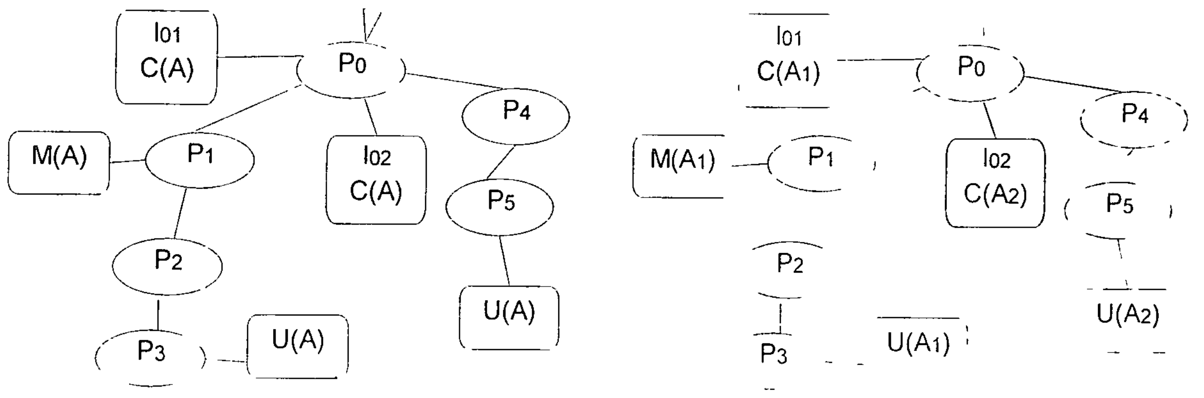

Figure 4 4 - The procedure P_0 has many information nodes *creating A in this situation, the procedure will contain the different items in which the data is spitted, i e the several portions of source code corresponding to the* information nodes I_{0i} *of P_0 contain different variables deriving by 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 A_1 and A_2 relatively to the groups of procedures $\{P_0, P_1, P_3\}$ and $\{P_0, P_4, P_5\}$ Note that procedures P_2 and P_4 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

Information node	derivation	direct derivation	exclusive derivation
I_{01}	-	-	-
I_{02}	-	-	-
I_1	I_{01}	I_{01}	I_{01}
I_3	I_{01}	I_{01}	I_{01}
I_5	I_{01}, I_{02}	I_{02}	I_{02}

Table 4 2 - In the table, the information nodes I_{hk} *are related to the procedure* P_h

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 $G_{A_1}=\{I_{01}$ of $P_0, P_1, P_3\}$ and $G_{A_2}=\{I_{02}$ 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)

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

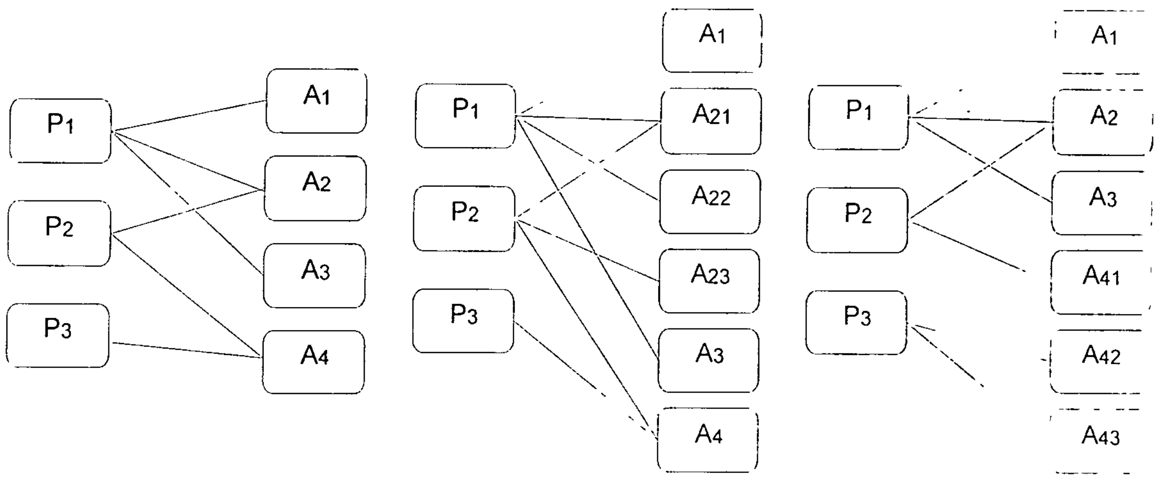

Figure 4 5 - In figure Figure 4 5a the vector of values of ΔIC for the procedures P_1, P_2 and P_3 is (0 75, 0 25, 0). By refining the data A_2 (figure Figure 4 5b) in the set of items A_{21}, A_{22} and A_{23}, the vector is (0 80, 0 30, 0) After the refining of data A_4 (figure Figure 4 5c) in the 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 $\Delta IC(P_k)$ before the refinement of A computed for any procedure P_k referring A, and the index $\Delta' IC(P_k))$ 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 $\max_{P_k \in PreSet(A)} (\Delta' IC(P_i)) - \Delta IC(P_i)$. 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 Busıness Orıented Language) programmıng language was developed in 1959 by a commıttee composed of government users and computer manufacturers. Sınce then, various COBOL committees have met to ensure that the evolutıon of the language through tıme followed an orderly fashion, thus making COBOL the most frequently used computer language directed at data processıng objectıves, extensıvely used ın admınıstratıve applıcatıons processıng of a large amount of ınput and output data

Yourdon[1] saıd "*One of the oldest, and arguably the most successful and popular of all programmıng languages, COBOL has been declared dead so many tımes that I've lost track countıng ... but COBOL lıves on*".

Table 5.1 shows the results of a recent research conducted by Caper-Jones [C92] regardıng the state of the art of software maıntenance It shows COBOL as the "dominant arena" for software maıntenance In thıs revıew, ıt ıs possıble to gaın an ınsıght ınto the proportıon of effort requıred ın the ındustrıal fıeld regardıng the maıntenance of COBOL *legacy ınformatıon systems* agaınst other conventional programmıng languages

In the recent lıterature a number of papers have specıfically descrıbed different approaches all successfully employıng reengıneerıng technıques dealıng wıth COBOL *legacy ınformatıon systems* Sneed's work ıs ımportant ın thıs area, because ıt successfully employees reengıneerıng technıques [S92] to mıgrate

[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.

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

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 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

As already seen in the sections from 2 2 1 to 2 2 3, it still appear 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*"[2]. 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[3] 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[4] 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

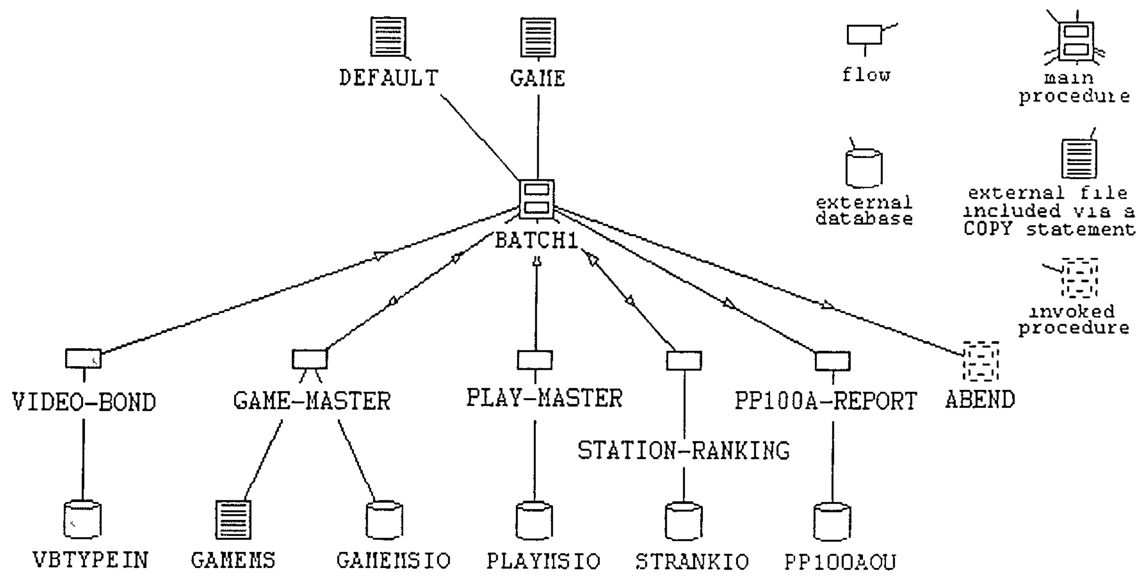

Figure 5 1

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

0000-MAINLINE	C210-EDIT-TBLE-DATA	C280-UPDATE-PLAY-DTL5
A100-HOUSEKEEPING	C210-EDIT-TBLE-DATA-EXIT	C280-UPDATE-PLAY-DTL6
A100-HOUSEKEEPING-EXIT	C220-EDIT-TABLE2-DATA	C280-UPDATE-PLAYFILE-EXIT
A100-FILE-STARTS	C220-EDIT-TABLE2-DATA-EXIT	D240-NO-STRANK-MATCH
A100-FILE-STARTS-EXIT	C230-EDIT-TABLE3-DATA	D240-NO-STRANK-MATCH-EXIT
A200-MAIN-PROCESS	C230-EDIT-TABLE3-DATA-EXIT	D250-PROCESS-PLAY-RECS
A200-BYPASS-UPDATES	C240-READ-STRANK	D250-PROCESS-PLAY-RECS-EXIT
A200-MAIN-PROCESS-EXIT	C240-READ-STRANK-EXIT	D280-REWR-PLAYFILE
A300-TERMINATION-RTN	C250-READ-PLAYFILE	D280-REWR-PLAYFILE-EXIT
A300-TERMINATION-RTN-EXIT	C250-READ-PLAY-HDR	E250-NO-PLAYMSIO-MATCH
B100-OPEN-FILES	C250-READ-PLAY-DTL1	E250-NO-PLAYMSIO-MATCH-EXIT
B100-OPEN-FILES-EXIT	C250-READ-PLAY-DTL3	F100-CHECK-VSAM-STATUS
B200-READ-VIDEO-BONDS	C250-READ-PLAY-DTL4	F100-CHECK-VSAM-STATUS-EXIT
B200-READ-VIDEO-BONDS-EXIT	C250-READ-PLAY-DTL5	U100-PRINT
B210-EDIT-DATA	C250-READ-PLAY-DTL6	U100-PRINT-EXIT
B210-EDIT-DATA-EXIT	C250-READ-PLAYFILE-EXIT	U110-PAGE-HEADER
B220-UPDATE-FILES	C260-UPDATE-GAME-MASTER	U110-PAGE-HEADER-EXIT
B220-UPDATE-FILES-EXIT	C260-UPDATE-GAME-MASTER-EXIT	U220-READ-GAME-MASTER
B230-PRINT-PLANS-REPORT	C270-UPDATE-STRANK	U220-READ-GAME-MASTER-EXIT
B230-PRINT-PLANS-REPORT-EXIT	C270-UPDATE-STRANK-EXIT	U300-SEARCH-TABLE3-LOC
B300-CLOSE-FILES	C280-UPDATE-PLAYFILE	U300-SEARCH-TABLE3-APP-EXIT
B300-CLOSE-FILES-EXIT	C280-UPDATE-PLAY-HDR	U400-READ-TABLE
C200-NO-PLANS-MATCH	C280-UPDATE-PLAY-DTL1	U400-READ-TABLE-EXIT
C200-PRINT-NOT-FOUND	C280-UPDATE-PLAY-DTL3	Z999-PGM-ABEND
C200-NO-PLANS-MATCH-EXIT	C280-UPDATE-PLAY-DTL4	Z999-PGM-ABEND-EXIT

Table 5 2

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 3 below The last paragraph of each procedure contains only the statement EXIT

0000	C200	D280
A10F	C210	E250
A10H	C220	F100
A200	* C230	U100
A300	C240	U110
B100	C250	U220
B200	C260	U300
B210	C270	U400
* B220	C280	Z999
B230	D240	
B300	D250	

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

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

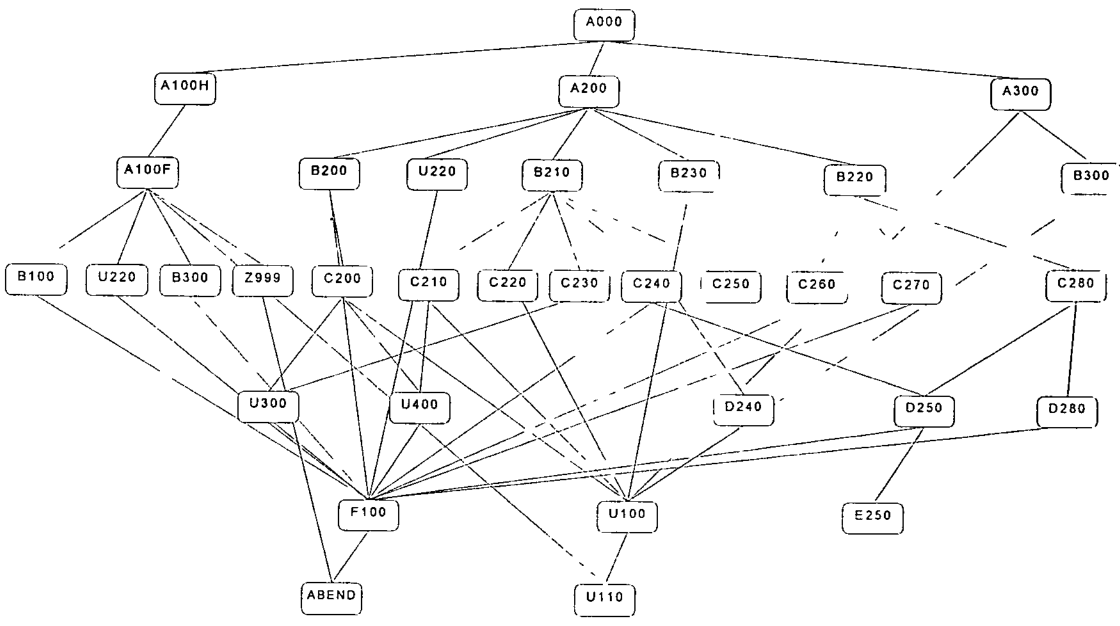

Figure 5 2

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

```
006600 01  PMR                                                  00710066
006700     03  PMR-1                                            00730067
006800         07  PMR-1-1                                      00740068
006900             11  PMR-1-1-1           PIC X(26)            00750069
007000             11  PMR-1-1-2           PIC X(04)            00760070
007100         07  PMR-1-2                 PIC X(09)            00770071
007200     03  PMR-2                       PIC X(105)           00780072
```

Figure 5 3 - Example of COBOL source program defining a data item in the DATA DIVISION *As the procedures C250 accesses* PMR *by referring to the elementary filed* PMR-1, *then it can be considered a record*

An important peculiarity of the COBOL programming language is that it is possible to access external files only through the *record area* associated with it in the ENVIRONMENT DIVISION of the FILE SECTION. The relations about the *record area* and the corresponding external files are presented in the figure 5 4 All the statements accessing the external files also access the corresponding record area, that is defined in the WORKING-STORAGE SECTION

FLOW NAME	EXTERNAL FILE NAME		RECORD AREA NAME
VIDEO-BONDS	VBTYPEIN	(COBOL external file)	VIDEO-BOND-RECORD
GAME-MASTER	GAMEMSIO	(COBOL external file)	GAME-MASTER-RECORD
PLAY-MASTER	PLAYMSIO	(COBOL external file)	PLAY-MASTER-RECORD
STATION-RANKING	STRANKIO	(COBOL external file)	RANK1-STAT-RANK-RECORD RANK2-STAT-RANK-RECORD
PP100A-REPORT	PP100AOU	(COBOL external file)	PP100A-LINE

Figure 5 4

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 <(RE)WRITE 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 affected by the execution of these statements, thus they should be considered as anomalous. In our example, as usual, all the OPEN and CLOSE statements are included 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.

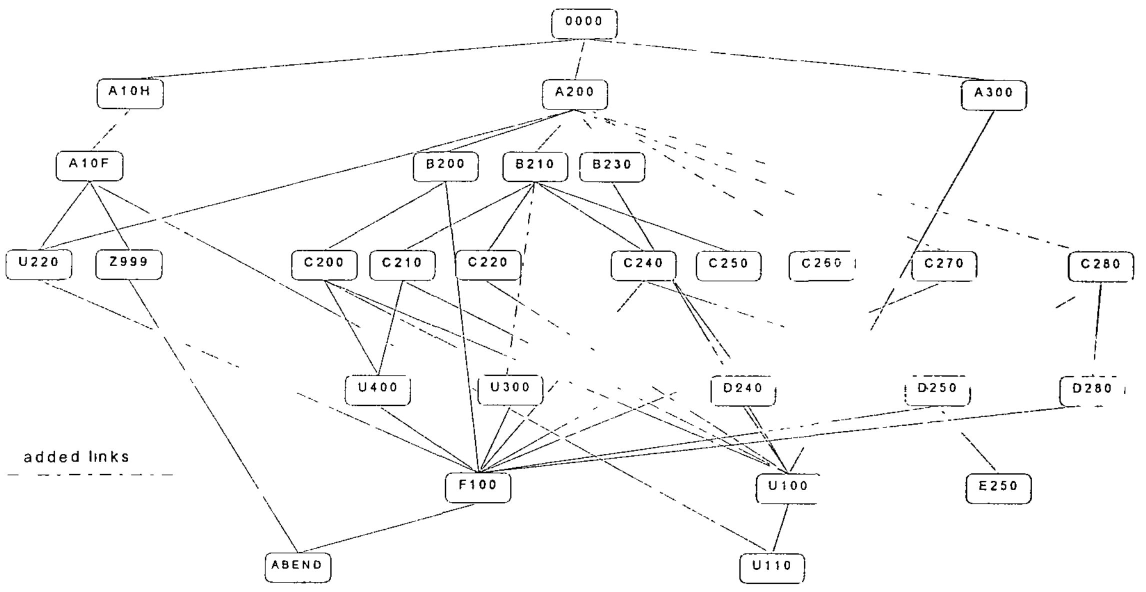

Figure 5 5

The bipartite graph is represented in figure 5.6. The dashed edges represent those *using accesses* to be removed.

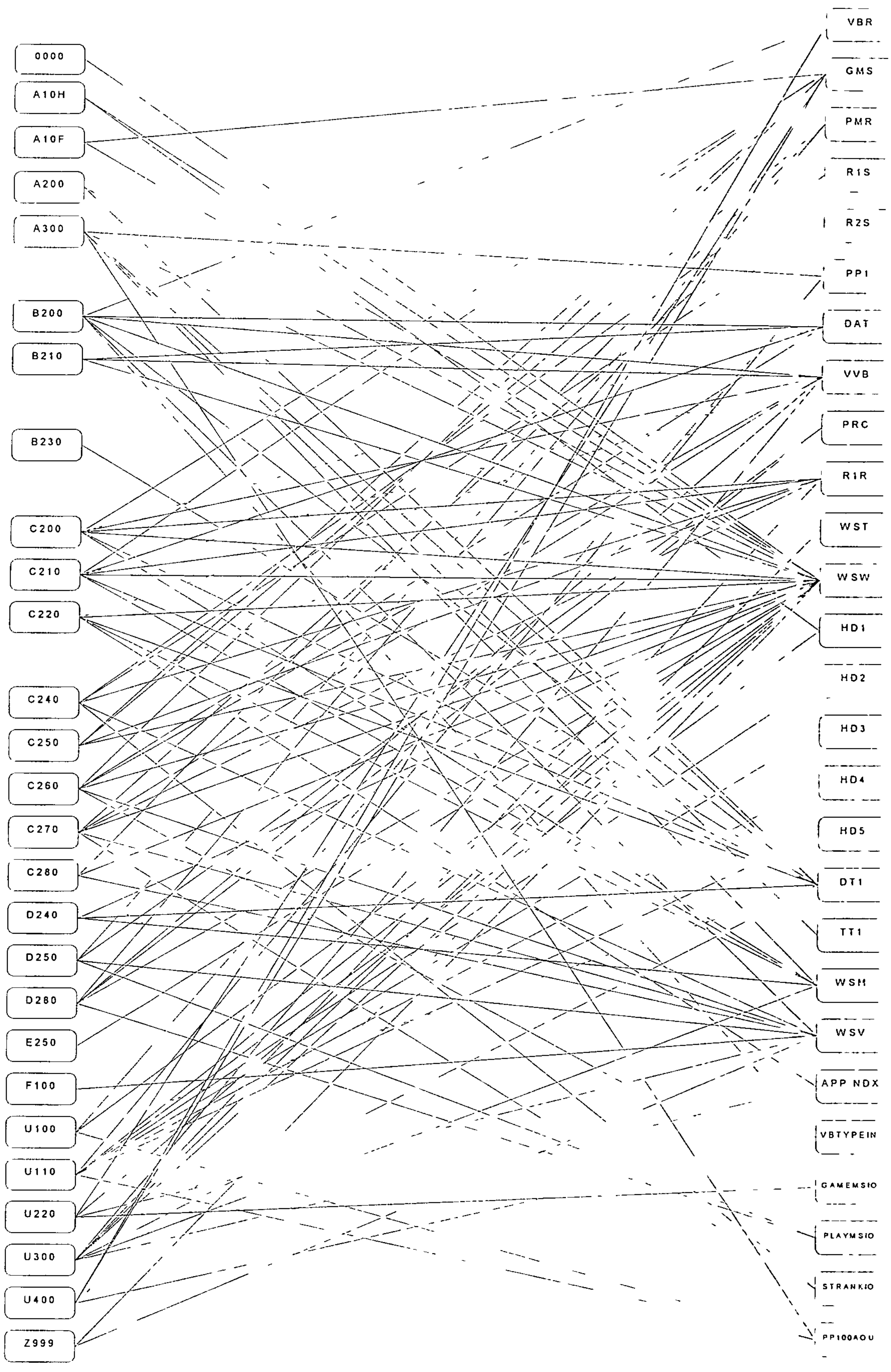
0000
A10H
A10F
A200
A300
B200
B210
B230
C200
C210
C220
C240
C250
C260
C270
C280
D240
D250
D280
E250
F100
U100
U110
U220
U300
U400
Z999
VBR
GMS
PMR
R1S
R2S
PP1
DAT
VVB
PRC
R1R
WST
WSW
HD1
HD2
HD3
HD4
HD5
DT1
TT1
WSH
WSV
APP NDX
VBTYPEIN
GAMEMSIO
PLAYMSIO
STRANKIO
PP100AOU

Figure 5 6 -

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

Procedure Name	IC()	ΔIC()
0000	0 0909091	0 0000000
A10H	0 1666667	0 0757576
A10F	0 1379310	0 0470219
A200	0 1612903	-0 0407299
A300	0 3863636	0 1843434
B200	0 3272727	0 0141414
B210	0 1515152	0 0606061
B230	0 1111111	0 0000000
C200	0 2807018	0 0786816
C210	0 2962963	0 0942761
C220	0 3076923	0 1056721
C240	0 3000000	0 1888889
C250	0 2051282	0 1142191
C260	0 3921569	0 1901367
C270	0 2291667	0 0271465
C280	0 2285714	0 0265512
D240	0 2162162	0 0141960
D250	0 2941176	0 1830065
D280	0 1891892	0 0982801
E250	0 0909091	0 0000000
F100	0 1111111	0 0000000
U100	0 1785714	0 0876623
U110	0 4411765	0 3502674
U220	0 2750000	0 0729798
U300	0 3584906	0 1564704
U400	0 2727273	0 2727273
Z999	0 1612903	-0 0407299

Table 5 4

By examining these indexes it 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, DT1 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.

Procedure	IC()	ΔIC()	IC()	ΔIC()	IC()	ΔIC()
Name	with WSW refined		with DT1 refined		with WSV refined	
0000	0.2500000	0.0000000	0.0909091	0.0000000	0.0909091	0.0000000
A10H	0.8333333	0.8333333	0.1666667	0.0757576	0.1666667	0.0757576
A10F	0.3076923	0.0576923	0.1379310	0.0470219	0.1379310	0.0470219
A200	0.3000000	-0.0611111	0.2250000	0.1340909	0.1612903	-0.0407299
A300	0.3571429	0.2460318	0.4339623	0.3430532	0.3863636	0.1843434
B200	0.2571429	0.0349207	0.3750000	0.1729798	0.3090909	0.1070707
B210	0.2307692	0.2307692	0.1515152	0.0606061	0.1515152	0.0606061
B230	0.1111111	0.0000000	0.1666667	0.1666667	0.1111111	0.0000000
C200	0.2790698	0.1679587	0.2258065	0.1348974	0.2807018	0.0786816
C210	0.2058824	0.0947713	0.2407407	0.1498316	0.2962963	0.0942761
C220	0.2500000	0.1388889	0.2307692	0.1398601	0.3076923	0.1056721
C240	0.3000000	0.1888889	0.3000000	0.1888889	0.2500000	0.2500000
C250	0.3333333	0.0000000	0.2051282	0.1142191	0.2051282	0.1142191
C260	0.2258065	0.1146954	0.3921569	0.1901367	0.3725490	0.2816399
C270	0.2500000	0.1388889	0.2291667	0.0271465	0.2083333	0.1174242
C280	0.4166667	-0.0277777	0.2285714	0.0265512	0.2000000	0.1090909
D240	0.2173913	0.1062802	0.1351351	0.0442260	0.2162162	0.0141960
D250	0.2941176	0.1830065	0.2941176	0.1830065	0.2352941	0.2352941
D280	0.2941176	0.2941176	0.1891892	0.0982801	0.1891892	0.0982801
E250	0.3333333	0.0000000	0.0909091	0.0000000	0.0909091	0.0000000
F100	0.1111111	0.0000000	0.1111111	0.0000000	0.2000000	0.2000000
U100	0.4000000	0.4000000	0.1785714	0.0876623	0.1785714	0.0876623
U110	0.5625000	0.5625000	0.4411765	0.3502674	0.4411765	0.3502674
U220	0.2727273	-0.0883838	0.2750000	0.0729798	0.2500000	0.1590909
U300	0.2424242	0.1313131	0.3018868	0.2109777	0.3584906	0.1564704
U400	0.3333333	0.3333333	0.2727273	0.2727273	0.2727273	0.2727273
Z999	0.3333333	0.2222222	0.1612903	-0.0407299	0.1290323	0.0381232

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 2nd column of table 5 5 and ΔIC( ) vector shown in the 3rd column of table 5 5 The vectors of indexes IC( ) and ΔIC( ) with the refinement of the data item DT1 are shown in the 4th and in the 5th column of table 5 5, respectively Finally, in the 6th and in the 7th column, the vectors of indexes relative to the refinement of DT1 are shown Note that after the refinement of WSW, the index ΔIC(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

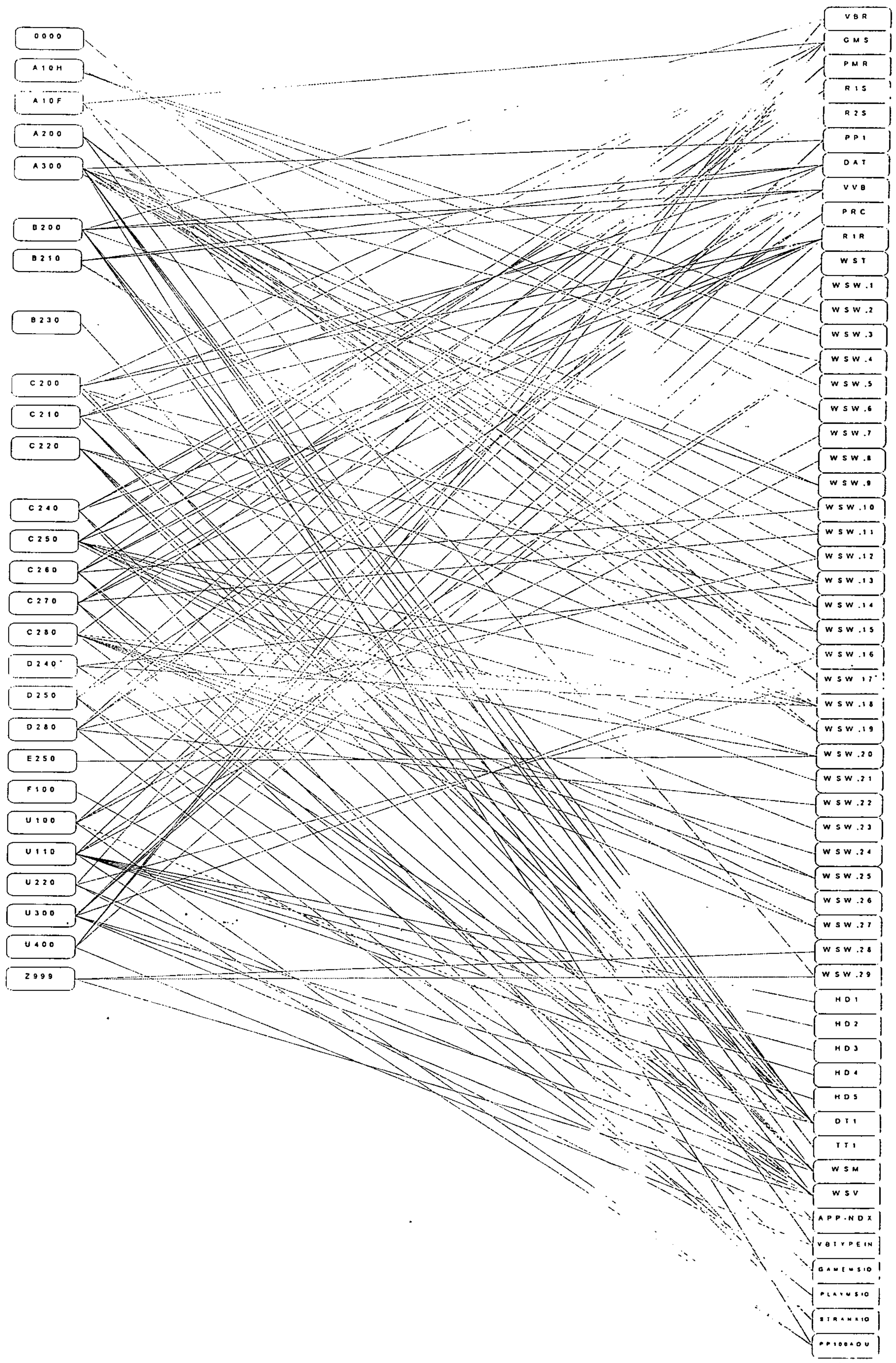
0000
A10H
A10F
A200
A300
B200
B210
B230
C200
C210
C220
C240
C250
C260
C270
C280
D240
D250
D280
E250
F100
U100
U110
U220
U300
U400
Z999
VBR
GMS
PMR
R1S
R2S
PP1
DAT
VVB
PRC
R1R
WST
WSW.1
WSW.2
WSW.3
WSW.4
WSW.5
WSW.6
WSW.7
WSW.8
WSW.9
WSW.10
WSW.11
WSW.12
WSW.13
WSW.14
WSW.15
WSW.16
WSW.17
WSW.18
WSW.19
WSW.20
WSW.21
WSW.22
WSW.23
WSW.24
WSW.25
WSW.26
WSW.27
WSW.28
WSW.29
HD1
HD2
HD3
HD4
HD5
DT1
TT1
WSM
WSV
APP-NDX
VBTYPEIN
GAMEMSIO
PLAYMSIO
STRANKIO
PP100AOU

Figure 5.7

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 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,
- if two procedures P_h and P_k belong to the same group, then all the procedures in the path of the *temporal graph* between P_h and P_k 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 P_k *directly derives* data A by I_0 if in any traversal of the *temporal graph* between I_0 and the first *information node* I_{k1} of P_k (inclusive), the node I_0 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

```
054300 0000.                                                   05910543
054400                                                         0544
054500        PERFORM A10H THRU                                05920545
054600                A10H-EXIT.                               05930546
054700                                                         0547
054800        PERFORM A200 THRU                                05950548
054900                A200-EXIT                                05960549
055000                       UNTIL GMS.                        05970550
055100                                                         05980551
055200        PERFORM A300 THRU                                05990552
055300                A300-EXIT.                               06000553
055400                                                         06010554
055500        STOP RUN.                                        06020555
```

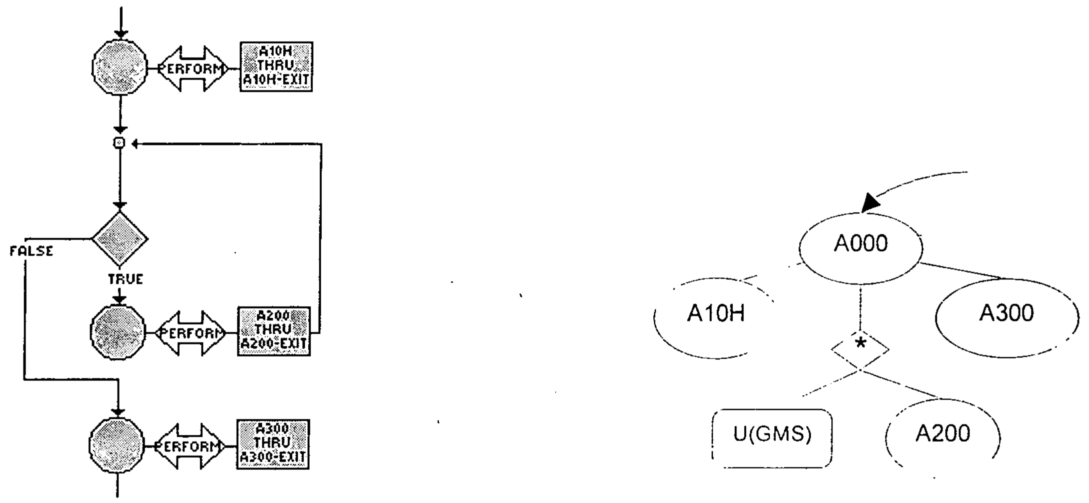

Figure 5.8 - The program representation in case of a UNTIL statement. These statements are typical of the COBOL programming style. The "" in figure 5.8.iii represents the fact that A200 can be repeated more than one time.*

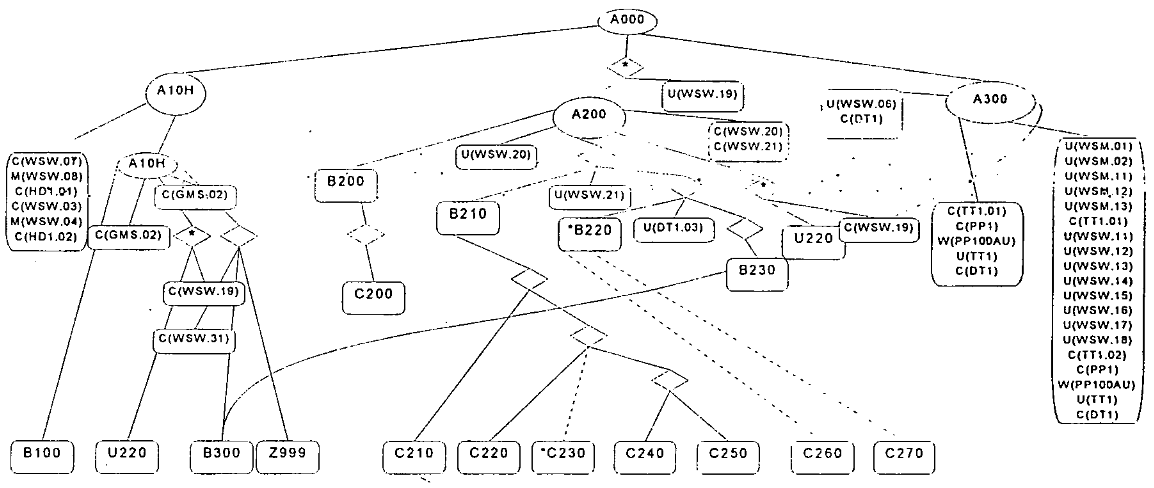

Figure 5.9 - The TPG shown in figure 5.9 is not relative to the initial stage of the technique. It is possible to note the references to the refined data. For sake of clearness, the procedure B220 *and* C230 *are still represented in the TPG. Remember that they were excluded from the bipartite graph, 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 P_0 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 A_1 and A_2, as shown in figure 5.10b, because of the identification of two different groups relative to A. The portion of code relative to I_{01} and I_{02} will contain references to A_1 and A_2 respectively. The procedure P_0 will belong to the two groups.

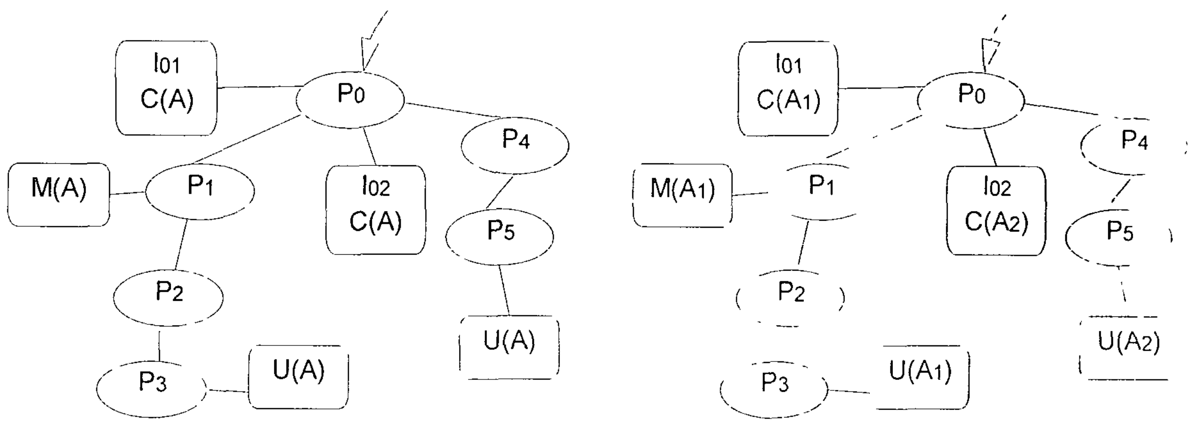

Figure 5 10 - The procedure P_0 has many information nodes *creating A In this case, the procedures P_1 and P_5 directly derive A from I_{01} and I_{02}, respectively Procedure P_3 derives A from I_{01}, but this derivation is not direct With the further assumption that there are no other procedures 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 *I_{0i} of P_0 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.

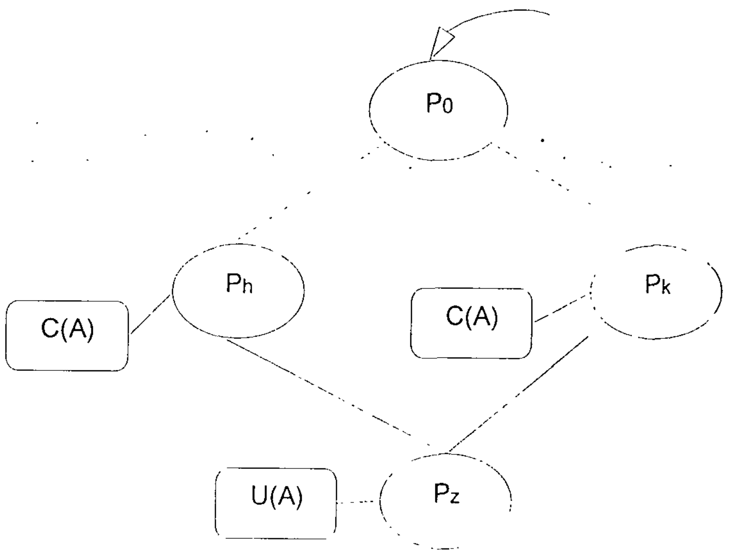

Figure 6.1 - Two procedures creating the same data but logically related. Further knowledge from the application domain can help to group more meaningful candidate modules, thus reducing the effort of the concept assignment *phase.*

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.

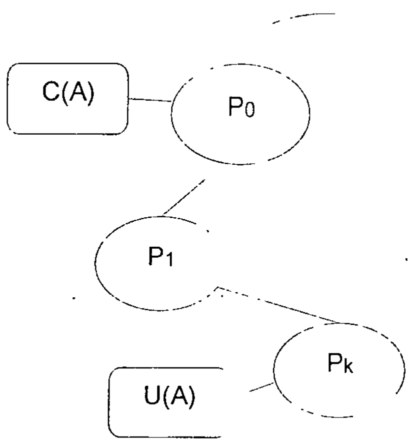

Figure 6 2

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 being 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

[A88] L. J. ARTHUS, "*Software Evolution· the Software Maintenance Challenge*", John Wiley and Sons, New York, 1988.

[A92] G. ALKHATIB, "*The Maintenance Problem of Application Software An Empirical Analysis*", Journal of Software Maintenance Research and Practices, **4**(2), pp. 83-104, 1992.

[A94] R. S. ARNOLD, "*Software Reengineering· A Quick History*", Communication of the ACM, **37**(5), pp. 13-14, 1994.

[ADA83] -, "*Reference Manual for the ADA Programming Language*", US Department of Defence, MIL STD1815A, 1983.

[AF92] R. S. ARNOLD, W. B. FRAKES, "*Software Reuse and Reengineering*", CASE Trends, **4**(2), pp. 44-48, 1992.

[ANSI83] -, "*IEEE Standard Glossary of Software Engineering Terminology*", ANSI / IEEE, Technical Report 729, 1983.

[AP82] R. S. ARNOLD, D. A. Parker, "*The Dimensions of Healthy Maintenance*", Proc. of the 6th International Conference on Software Engineering, pp. 10-27, 1982.

[AP95] J. D. AHRENS, N. S. PRYWES, "*Transition to a Legacy- and Reuse-Based Software Life Cycle*", Computer, **28**(10), pp. 27-36, 1995.

[ASU86] A. V. AHO, R. SETHI, J. D. ULLMANN, "*Compiler. Principles, Techniques and Tools*", Reading, MA: Addison Wesley, 1986

[B81] B. W. BOEHM, "*Software Engineering Economics*", Prentice Hall, Inc., New Jersey, 1981.

[B85] E. BUSH, "*Automatic Restructuring of COBOL*", Conference on Software Maintenance, IEEE Comp. Soc. Press, 1985.

[B87$_A$] F. P. BOEHM, "*Improving Software Productivity*", IEEE Computer, **4**, pp. 43-57, 1987.

[B87$_B$] G. BOOCH, "*Software Components with ADA*", Benjamin Cummings, Menlo Park, CA, 1987.

[B87$_C$] F. P. BROOKS, "*No Silver Bullets - Essence and Accidents of Software Engineering*", IEEE Computer, **20**(4), pp. 10-20, 1987.

[B89] T. J. BIGGERSTAFF, "*Design Recovery for Maintenance and Reuse*", IEEE Computer, **22**(7), pp. 36-49, 1989.

[B90] V. R. BASILI, "*Viewing Maintenance as Reuse-Oriented Software Development*", IEEE Computer, **23**(1), pp. 19-25, 1990.

[B91$_C$] K. BENNETT, "*The Software Maintenance of Large Software Systems· Management, Methods and Tools*", Elsevier Science Publisher, London, 1991.

[B91$_A$] B. I. BLUM, "*The Software Process for Medical Application*", T. Timmers and B. I. Blum editors, Software Engineering in Medical Informatic, pp. 3-25, Elsevier Science Publisher B.V., 1991.

[B91$_B$] P. BROWN, "*Integrated Hypertext and Program Understanding Tools*", IBM System Journal, **30**(3), pp. 363-392, 1991.

[B93$_A$] L. BERNSTEIN, "*Tidbits*", ACM SIGSOFT - Software Engineering Notes, IEEE Comp. Soc. Press, **18**(3), pp. A-55, 1993.

[B93$_B$] J. BOWEN, "*From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation*", Journal of Software Maintenance: Research and Practice, **5**(4), pp. 205-234, 1993.

[BBKMN94] R. BRAND, S. BURSON, T. KITZMILLER, L. MARKOSIAN, P. NEWCOMB, "*Using an Enabling Technology to Reengineer Legacy Systems*", Communication of the ACM, **37**(5), pp. 58-70, 1994.

[BBL93$_A$] J. BOWEN, P. BREUER, K. LANO, "Formal Specifications in Software Maintenance: From code to Z++ and back again", Information and Software Technology, **35**(11/12), pp. 679-690, 1993.

[BBL93$_B$] J BOWEN, P. BREUER, K. LANO, "A Compendium of Formal Techniques for Software Maintenance", IEEE/BCS Software Engineering Journal, **8**(5), pp. 253-262, 1993.

[BCD92] P. BENEDUSI, A. CIMITILE, U DE CARLINI, "*Reverse Engineering Process, Design Recovery and Structure Charts*", Journal of System and Software, **19**, pp. 225-245, 1992.

[BH85] W. E. BEREGI, G. F. HOFFNAGLE, "*Automating the Software Development Process*", IBM System Journal, **24**(2), pp. 102-120, 1985.

[BHL93] P. T. BREUER, H. HAUGHTON, K. LANO, "*Reverse Engineering COBOL via formal methods*", Journal of Software Maintenance. Research and Practice, **5**(1), pp. 13-35, 1993.

[BL91] P. T. BREUER, K. LANO, "*Creating Specification from Code - Reverse Engineering Techniques*", in Journal of Software Maintenance, **3**(3), pp. 145-162, 1991.

[BM97] E. BURD, M. MUNRO, "*Enriching Program Comprehension for Software Reuse.*", International Workshop on Program Comprehension, IEEE Comp. Soc. Press, pp. 130-138, 1997.

[BMW94] T. J. BIGGERSTAFF, B. G. MITBANDER, D. WEBSTER, "*Program Understanding and the Concept Assignment Problem*", Communication of the ACM, **37**(5), pp. 72-83, 1994.

[BMW96$_A$] E. BURD, M. MUNRO, C. WEZEMAN, "*Analysing Large COBOL Programs. the Extraction of Reusable Modules*", International Conference on Software Maintenance, IEEE Comp. Soc. Press, pp. 238-243, 1996.

[BMW96$_B$] E BURD, M MUNRO, C WEZEMAN, "*Extıactıng Reusable Modules from Legacy Code Consıderıng the Issues of Module Gıanularıty*", Workıng Conference on Reverse Engıneerıng, IEEE Comp Soc Press, pp 189-196, 1996

[BP89] T J BIGGERSTAFF, A J PERLIS, "*Software Reusabılıty Concepts and Models*", ACM Press, Addıson-Wesley, New York, 1989

[BS95] M L BRODIE, M STONEBRAKER, "*Mıgratıng Legacy System*", Morgan Kaufmann Publıshers, 1995

[C89$_A$] F W CALLISS, "*Inter-Module Code Analysıs for Softwaıe Maıntenance*", Ph D Thesıs, Unıversıty of Durham, School of Engıneerıng and Applıed Scıences, Computer Scıence, 1989

[C89$_B$] T A CORBI, "*Program Understandıng Challenge foı the 90's*", IBM System Journal, **28**(2), pp 294-306, 1989

[C92] T CAPERS-JONES, "*Gerıatrıc Care for Ageıng Softwaıe*", Knowledge Based 1, Software Productıvıty Research Inc , Burlıngton, 1992

[CC90] E J CHIKOFSKY, J H CROSS Jr , "*Reverse Engıneeıng and Desıgn Recovery a Taxonomy*", IEEE Software, **7**(1), pp 13-17, 1990

[CC92] G CANFORA, A CIMITILE, "*Reverse Engıneerıng and Intermodular Data Flow A Theoretıcal Approach*", Journal of Software Maıntenance Research and Practıce, **4**(1), pp 37-59, 1992

[CCD91] G CANFORA, A CIMITILE, U DE CARLINI, "*A Logıc Based Approach to Reverse Engıneerıng Tools Productıon*", Proceedıngs of Conference on Software Maıntenance, IEEE Comp Soc Press, pp 83-91, 1991

[CCM94] G CANFORA, A CIMITILE, M MUNRO, "*RE^2 Reverse Engıneerıng and Reuse Re-engıneerıng*", Journal of Software Maıntenance, **6**(5), pp 53-72, 1994

[CCM96] G Canfora, A Cimitile, M Munro, "*An Algorıthm for Identıfyıng Object ın Code*", Software Practıce and Experıence, **26**(1), pp 25-48, 1996

[CCTA] -, "*SSADM Versıon 4 Manuals*", NCC-Blackwell, Manchester, 1990

[CCR90] T N Commer Jr , J R Comer, D J Rodjak, "*Developıng Reusable Software for Mılıtary System - Why ıt ıs needed and why ıt ısn't workıng*", ACM SIGSOFT Software Engıneers Notes, **15**(3), pp 33-38, 1990

[CCV95] G Canfora, A Cimitile, G Visaggio, "*Assessıng Modulaıısatıon and Code Scavengıng Technıques*", Journal of Software Maıntenance, IEEE Comp Soc Press, **26**(1), pp 25-48, 1996

[CD91$_A$] A Cimitile, U De Carlini, "*Reveıse Engıneerıng-Algorıthms for Program Graph Productıon*" Software Pıactıce and Expeııence, **21**(5), pp 519-537, 1991

[CD91$_B$] A Cimitile, U De Carlini, "*Reveıse Engıneerıng Algorıthms foı Programs Graph Productıon*", Software - Practıce and Experıence, **21**(5), pp 519-537, 1991

[CD95] A Cimitile, U De Carlini, "*Metodologıe, Tecnıche e Stıumentı dı Reverse Engıneerıng*", Franco Angelı edıtore 1995

[CDDF94] G Canfora, A De Lucia, G A Di Lucca, A R Fasolino, "*Recoverıng the Archıtectural Desıgn for Software Compıehensıon*", Proceedıngs of the 3rd Internatıonal Workshop on Program Comprehensıon, IEEE Comp Soc Press, pp 30-38, 1994

[CDDF97] A Cimitile, A De Lucia, G A Di Lucca, A R Fasolino, "*Identıfyıng Objects ın Legacy Systems*", Internatıonal Workshop on Program Comprehensıon, IEEE Comp Soc Press, pp 138-147, 1997

[CDM90] A. CIMITILE, G. A. DI LUCCA, P. MARESCA, "*Maintenance and Intermodular Dependencies in Pascal Environment*", Proceedings of the International Conference on Software Maintenance, IEEE Comp. Soc. Press, pp. 166-173, 1990.

[CFM93] A. CIMITILE, A. R. FASOLINO, P MARESCA. "*Reuse-Reengineering and Validation via Concept Assignment*", Proceedings of the International Conference on Software Maintenance, Montreal, Quebec, Canada, IEEE Comp. Soc. Press, pp. 216-225, 1993.

[CH91] R. O. CHESTER, J. W. HOOPER, "*Software Reuse· Guidelines and Methods*", Plenum Press, New York, 1991.

[CMR88] B. J. CORNELIUS, M. MUNRO, D. J. ROBSON, "*An Approach to Software Maintenance Education*", Software Engineering Journal, **4**(4), pp. 233-240, 1988.

[CMW89] F. CALLIS, M. MUNRO, M. WARD, "*The Maintainer's Assistant*", Proceedings of the International Conference on Software Maintenance, 1989.

[CO90] E. J. CHIKOFSKY, W. M. OSBORNE, "*Fitting Pieces to the Maintenance Puzzle*", IEEE Software, **7**(1), pp. 11-12, 1990

[CV95] A. CIMITILE, G. VISAGGIO, "*Software Salvaging and the Call Dominance Tree*", The Journal of Systems and Software, **2**(1), pp. 25-48, 1995.

[CWW92] S. CERI, G. WEIDERHOLD, P. WEGNER, "*Toward Megaprogramming*", *Comm ACM,* **35**(11), pp. 89-99, 1992.

[CY79] L. L. CONSTANTINE, E. YOURDON, "*Structured Design Fundamentals of a Discipline of Computer Program and System Design*", Prentice Hall, Englewood Cliffs, New York, 1979.

[D95] A. DE LUCIA, "*Identifying Reusable Functions in Code Using Specification Driven Technique*", M. Sc. Thesis, University of Durham, 1995.

[DGN89] W DIETRICH, F. GRACER, L. NACKMAN, "*Saving a Legacy with Objects*", *in OOPSLA'90 ACM Conference on Object oriented Programming System, Languages and Application* (Meyrowitz N., editor), reading, MA: Addison Wesley, pp. 77-88, 1989.

[DK93] M. F. DUNN, J. C. KNIGHT, "*Automating the Detection of Reusable Parts in Existing Software*", Conference on Software Maintenance, Baltimore, Maryland, IEEE Comp. Soc Press, pp. 381-390, 1993.

[DH72] O. -J. DAHL, C. A. R. HOARE, "*Hierarchical Program Structures*", Structured Programming, Academic Press Inc., London, 1972.

[DH89] L. DUSINK, P. HALL, "*Introduction to Re-use*", Proceedings of the Software Re-use Workshop, pp. 1-19, 1989.

[DT88] S. DANFORTH, C. TOMLINSON, "*Type Theories and Object-Oriented Programming*", ACM Computing Survey, **20**(1), pp. 29-72, 1988

[E76] J. L. ELSHOFF, "*An Analysis on Some Commercial PL/1 Programs*", IEEE Transaction on Software Engineering, **SE-2**(2), pp. 113-120, 1976.

[EKN91] A. ENGBERTS, W. KOZACZYNSKI, J. Q. NING, "*Concept Recognition-Base Program Transformation*", Conference on Software Maintenance, pp. 73-92, 1991.

[EM93] H. M. EDWARDS, M. MUNRO, "*RECAST. Reverse Engineering from COBOL to SSADM Specification*", Proceedings 15th International Conference on Software Engineering, IEEE Comp. Soc. Press, pp. 499-508, 1993.

[FJM89] J. R. FOSTER, A. E. P. JOLLY, M. T. NORRIS, "*An Overview of Software Maintenance*", British Telecomm Technical Journal, 7(4), pp. 37-46, 1989.

[FK87] N. E. FENTON, A. A. KAPOSI, "*Metrics and Software Structures*", Information and Software Technology, **29**(6), pp. 301-320, 1987

[FW86] N. E. FENTON, W. WITTHY, "*Axiomatic Approach to Software Metrication through Program Decomposition*", The Computer Journal, **29**(4), pp. 330-339, 1986.

[FO76$_A$] L. D. FOSDICK, L. J. OSTERWEIL, "*Data Flow Analysis in Software Reliability*", Computer Survey, 8(3), pp. 305-330, 1976.

[FO76$_B$] L. D. FOSDICK, L. J OSTERWEIL, "*DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs*", Software: Practice and Experience, 6, pp. 473-486, 1976.

[FOW84] J. FERRANTE, K. J. OTTENSTEIN, J. D. WARREN, "*The Program Dependence Graph and its Use in Optimisation*", ACM Trans. Programming Languages and Systems, **9**(3), pp. 319-349, 1987.

[G94] I. M. GRAHAM, "*Migrating to Object Technology*", Addison Wesley. 1994.

[GJM91] C. GHEZZI, M. JAZAYERI, D. MANDRIOLI, "*Fundamentals of Software Engineering*", Prentice-Hall International, Inc. New Jersey, 1991

[GK95] H. GALL, R. KLOSCH, "*Finding Objects in Procedural Programs an Alternative Approach*", Working Conference on Reverse Engineering, IEEE Comp. Soc. Press, pp. 208-216, 1995.

[GN81] R. L. GLASS, R. A. NOISEUX, "*Software Maintenance Guidebook*", Prentice Hall, 1981.

[GP90] B. GRABOWSKI, N. PENNINGTON, "*Psychology of Programming*", Academic Press, London, 1990.

[GT96] P. A. GRUBB, A. A. TAKANG, "*Software Maintenance Concepts and Practice*", International Thomson Computer Press, 1996.

[H77] M. S. HECHT, "*Flow Analysis of Computer Programs*", Elsevier, North Holland, 1977.

[H86] W. E. HOWDEN, "*A Functional Approach to Program Testing and Analysis*", IEEE Transaction on Software Engineering, **SE-12**(10), pp. 997-1005, 1886.

[H88] D A. HIGGINS, "*Data Structured Maintenance the Warnier/Orr Approach*", Dorset House Publishing Co Inc, New York, 1988

[H90] M HAMMER, "*Reengineering Work Don't Automate, Obliterate*", Harvard Business Review, 1990.

[HHW76] D HEDLEY, M A HENNEL, M R WOODWARD, "*On Program Analysis*", Information Processing Letters, 5, pp 136-140, 1976

[HL91$_A$] H HAUGHTON, K LANO, "*A Specification-Based Approach to Maintenance*", Journal of Software Maintenance Research and Practice, 3(1), pp. 193-213, 1991

[HL91$_B$] H. HAUGHTON, K. LANO, "*Extracting Design and Functionality from Code*", REDO Project Document 2487-TN-PRG-1085, Oxford University, 1991

[HM84] E HOROWITZ, J B MUNSON, "*An Expansive view of Reusable Software*", IEEE Transaction on Software Engineering, **SE-10**(5), pp. 477-487, 1984

[HR88] J. HARTMANN, D J ROBSON, "*Approaches to Regression Testing*", Proceeding of the Conference on Software Maintenance, Computer Society Press, pp. 368-372, 1988,

[HR89] J. HARTMANN, D J ROBSON, "*Revalidation During the Software Maintenance Phase*", Tech Rep , School of Engineering and Applied Science, University of Durham, 1989.
Computer Science Technical Report TR 1/89.

[IBM94] E BUSS, R. DE MORI, W M GENTLEMAN, J HENSHAW, H JOHNSON, K KONTOGIANNIS, E MERLO, H A MULLER, J MYLOPULOS. S PAUL, A PRAKASH, M STANLEY, S R TILLEY, J TROSTER, K WONG, "*Investigating Reverse Engineering Technologies for the CAS Program Understanding Project*", IBM System Journal, **33**(3), pp 477-500, 1994.

[JK85] K. JENSEN, N. WIRTH, "*PASCAL User Manual and Report*", Springer-Verlag, New York, 3rd Edition, 1985
Revised By A. B. MIKEL and J. F. MINER.

[JL94] P. E. LIVADAS, T. JOHNSON, "*A New Approach to find Objects in Programs*", Journal of Software Maintenance· Research and Practice, **6**(5), pp. 249-260, 1994.

[K87] K. C. KANG, "*A Reuse-Based Software Development Methodology*", Proceeding of the Workshop in Software Reuse, 1987.

[KN95] G. KOTIK, P. NEWCOMB, "*Reengineering procedural into Object Oriented Systems*", Working Conference on Reverse Engineering, IEEE Comp. Soc. Press, pp. 237-249, 1995.

[L85] M. M. LEHMAN, "*Program Evolution*", Academic Press, London, 1985.

[L90] K. LANO, "Z++, *an Object Oriented Extension to Z*", Proceedings of the Z User Meeting, Oxford, 1990

[L93] S. LAUCHLAN, "*Case Study Reveals Future Shocks*", Computing, 1993.

[LM90] P. J. LAYZELL, L. MACAULAY, "*An Investigation into Software Maintenance: Perception and Practices*", Conference on Software Maintenance, IEEE Comp. Soc. Press, pp. 130-140, 1990.

[LOWY94] S. LIU, R. M. OGANDO, N. WILDE, S. S. YAU "*An Object Finder for Program Structure Understanding in Software Maintenance*", Journal of Software Maintenance: Research and Practice, **6**(5), pp. 261-283, 1994.

[LPR94] W. A. LANDI, H. D. PANDI, B. G. RYDER, "*Interprocedural Def-Use Association For C System with Single Level Pointers*", IEEE Transaction on Software Engineering, **SE-20**(5), pp. 385-403, 1994

[LR91] W. A. LANDI, B. G. RYDER; "*Pointer-Induced aliasing· a Problem Classıfication*", in *Proceedings of the 18th Annual ACM Symposıum on Principles of Programming Languages*, Orlando, Florida, U S A , ACM Press, pp. 93-103, 1991.

[LS80] B. P. LIENTZ, E. B. SWANSON, "*Software Maintenance Management*", Addison Wesley Publishing Company, Reading, Massachusetts, 1980.

[LST78] B. P. LIENTZ, E. B. SWANSON, G. E. TOMPKINS, "*Characteristic of Applicatıon Software Maintenance*", Communication of the ACM, **21**(6), pp. 466-471, 1978.

[LW90] S. LIU, N. WILDE, "*Identıfyıng Objects in a Conventıonal Procedural Language. An Example of Data Design Recovery*", International Conference on Software Maintenance, IEEE Comp. Soc. Press, pp. 266-271, 1990.

[MD91$_{16}$] D. A. STROKES, "*Requirements Analysis*", J. McDermid editor, Software Engıneer's Reference Book, Chapter 16, pp 16/1-16/21, Butterworth-Heinemann Ltd, Oxford, 1991.

[MD91$_{20}$] K. H. BENNET, B. CORNELIUS, M. MUNRO, D. ROBSON, "*Software Maintenance*", J. McDermid editor, Software Engineer's Reference Book, Chapter 20, pp. 20/1-20/18, Butterworth-Heinemann Ltd, Oxford, 1991.

[MD91$_{41}$] P. HALL, C. BOLDYREFF, "*Software Reuse*", J. McDermid editor, Software Engineer's Reference Book, Chapter 41, pp. 41/1-41/12, Butterworth-Heinemann Ltd, Oxford, 1991.

[MS87] J. C. MILLER, B. M. STRAUSS III, "*Implicatıon of Automatıc Restructuring of COBOL*", ACM SIGPLAN Notices, **22**(6), pp. 76-82, 1987.

[MU90] H. A. MULLER, J. S. UHL, "*Composıng Subsystem Structures Using Partite Graphs*", Conference on Software Maintenance, 1990

[NS87] K. W. NIELSEN, K. SHUMATE, "*Designing Large Real Time System with ADA*", Communication of the ACM, **30**(8), pp. 695-715, 1987.

[NS95] H. M. SNEED, E. NYARY, "*Extracting Object-Oriented Specification from Procedurally Oriented Programs*", Working Conference on Reverse Engineering, IEEE Comp. Soc. Press, pp. 217-226, 1995.

[O87] W. OSBORNE, "*Building and Sustaining Software Maintainability*", Proc. Of Conference on Software Maintenance, pp. 13-23, 1987.

[O90] W. OSBORNE, "*Software Maintenance and Computers*", IEEE Computer Society Press, pp. 2-14, 1990.

[OO84] K. J. OTTENSTEIN, L. M. OTTENSTEIN, "*The Program Dependence Graph in a Software Development Environment*", ACM Sigplan Notices, **19**(5), pp. 177-184, 1984.

[P72$_A$] D. L. PARNAS, "*On the Criteria to be Used in Decomposing Systems into Modules*", Communication of the ACM, **15**(12), **pp. 1053-1058,** 1972.

[P72$_B$] D. L. PARNAS, "*Information Distribution Aspects of Design Methodology*", Proceedings of the IFIP Congress-1971, pp. 339-344, 1972.

[P80] M. PAGE-JONES, "*The Practical Guide to Structured Systems Design*", Yourdon Press, New York, 1980.

[P82] G. PARIKH, "*Technique of Program and System Maintenance*", Winthrop Publishers, 1982.

[P86] G. PARIKH, "*Making the Immortal Language Work*", International Computer Program Business Software Review, 7(2), 1986.

[P91] R. PRIETO-DIAZ, "*Making Software Reuse Work. An Implementation Model*", ACM SIGSOFT Software Engineering Notes, **16**(3), 1991

[P94] R. S. PRESSMAN, "*Software Engineering: a Practicioner's Approach*", McGraw Hill, 1994.

[P95] D. PEARCE, "*It's a Wrap*", Consultant's Conspectus, 1995.

[PT94] J POULIN, W TRACZ "*WISR '93 6th Annual Workshop on Software Reuse Summary and Working Group Reports*", ACM SIGSOFT Software Engineering Notes, **19**(1), pp 55-71, 1994

[S76] E B SWANSON, "*The Dimensions of Maintenance*", Proc of the 2nd International Conference on Software Engineering, IEEE Comp Soc Press, pp 492-497, 1976

[S87] N F SCHNEIDEWIND, "*The State of Software Maintenance*", IEEE Transaction on Software Engineering, **SE-13**(3), pp 303-310, 1987

[S91] H M SNEED, "*Bank Application Reengineering and Conversion at the Union Bank of Switzerland*", Conference on Software Maintenance, IEEE Comp Soc Press, pp 60-70, 1991

[S92] H M SNEED, "*Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture*", Conference on Software Maintenance, IEEE Comp Soc Press, pp 105-116, 1992

[S94] H M SNEED, "*Downsizing Large Application Programs*", Journal of Software Maintenance Research and Practice, **6**(5), pp 105-116, 1994

[S95] I SOMMERVILLE, "Software Engineering", 3rd Edition, International Computer Science Series, Addison-Wesley, Workingham, 1995

[S96$_A$] H M SNEED, "*Encapsulating Legacy Software for Use in Client/Server Systems*", Working Conference on Reverse Engineering, IEEE Comp Soc Press, pp 104-119, 1996

[S96$_B$] H M SNEED, "*Object-Oriented COBOL Recycling*", Working Conference on Reverse Engineering, IEEE Comp Soc Press, **pp. 169-178,** 1996

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

[T88] W TRACZ "*Software Reuse Myths*", ACM SIGSOFT Software Engineering Notes, **13**(1), pp 17-21, 1988

[T94] M E. TORTORELLA, "*Identification of Abstract Data Types in Code*", M Sc Thesis, University of Durham, 1994

[TT92] T TAMAI, Y TORIMUTSU, "*Software Lifetime and its Evolution Process over Generations*", Proceeding of the 8th Conference of Software Maintenance, pp 63-69, 1992

[V80] E V VAN HORN, "*Software Engineering*", Academic Press, New York, 1980.

[$V93_A$] D VAN EDELSTEIN, "*Report on the IEEE STD 1219-1993 - Standard for Software Maintenance*", ACM SIGSOFT, Software Engineering Notes, IEEE Comp. Soc. Press, **18**(4), pp. 94-95, 1993.

[$V93_B$] H VAN VLIET, "*Software Engineering Principles and Practice*", John Wiley, Chichester, 1993

[V94] A. VON MAYRHAUSER, "*Maintenance and Evolution of Software Products*", Advance in Computers, **38**(1), pp 1-49, 1994

[W79] M. WEISER, "*Program Slices Formal, Psychological, and Practical Investigation of an Automatic Program Abstraction Method*", Ph D Thesis, University of Michigan, 1979

[W82] M WEISER, "*Programmers use Slices when Debugging*", Communication of the ACM, **25**(7), pp 446-452, 1982

[W84] M. WEISER, "*Program Slicing*", IEEE Transaction on Software Engineering, **SE-10**(4), pp. 352-357, 1984

[W88] C WATERS, "*Program Translation via Abstraction and Reimplementation*", Transaction on Software Engineering, IEEE Comp Soc Press, **14**(8), 1988

[W95] P WINSBERY, "*Legacy Code - Don't Reengineer it, Wrap it*", Datamation, pp. 36-41, **May** 1995

[Y75] E YOURDON, "*Techniques of Program Structure and Design*", Prentice-Hall Inc., 1975.

[Y90] E YOURDON, "*Object Oriented COBOL*", American Programmer, 3(2), 1990

[Z93$_{10}$] J BOWEN, P BREUER, "*Decompilation*", Henk van Zuylen (ed), The REDO Compendium: Reverse Engineering for Software Maintenance, Chapter 10, John Wiley & Sons, pp 131-138, 1993

[Z93$_{15}$] J BOWEN, P BREUER, K. LANO, "*Understanding Programs through Formal Methods*", Henk van Zuylen (ed), The REDO Compendium Reverse Engineering for Software Maintenance, Chapter 15, John Wiley & Sons, pp 195-223, 1993

[Z93$_{16}$] P BREUER, H. HAUGHTON K LANO, "*Reverse Engineering COBOL via Formal Methods*", Henk van Zuylen (ed), The REDO Compendium Reverse Engineering for Software Maintenance, Chapter 16, John Wiley & Sons, pp 225-248, 1993