An Aid to Similar-Characteristics-Code Clustering

by

Mohammad H. Al-Huwaidi

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June, 1997

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

An Aid to Similar-Characteristics-Code Clustering

BY

Mohammad H. Al-Huwaidi

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

June 1997

UMI Number: 1385831

UMI Microform 1385831
Copyright 1997, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by *Mohammad Hussain Al-Huwaidi* under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of College of graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Thesis Committee

[illegible]

Dr. JarAllah Al-Ghamdi (Chairman)

٩٧/٦/١١هـ

Dr. Muhammad Al-Mulhem (Co-Chairman)

[illegible] 11/6/97

Dr. Mohammed Shafique (Member)

[illegible]

Dr. Nabil Al-Zamil (Member)

[illegible]

Department Chairman
Dr. Talal Maghrabi

[illegible]

Dean, College of Graduate Studies
Dr. Abdallah M. Al-Shehri

15/6/97

Date

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

THESIS ABSTRACT

FULL NAME OF STUDENT:	Mohammad Hussain Al-Huwaidi
TITLE OF STUDY	:An Aid to Similar-Characteristics-Code Clustering
MAJOR FIELD	: Computer Science
DATE OF DEGREE	: June, 1997

Software engineers are faced with new technology and left with legacy systems, that are integral part of an organization. Legacy systems are difficult and costly to maintain and cannot just be abandoned. They are full of problems and obstacles that aggravate handling them. Industry and literature have been proposing schemes to deal with legacy systems and their problems. This thesis will propose a methodology and a tool to alleviate the problem of dealing with these notorious legacy systems. The proposed methodology is based on similar-characteristics possessed by fragments of code scattered in source code file(s). When applied, this methodology should help with source code understanding, which in turn enhances legacy systems' migration, maintenance, and parts reusability.

MASTERS OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Dhahran, Saudi Arabia

<u>Date</u>
June, 1997

خلاصة الرسالة

إسم الطالب الكامل : محمد بن حسين بن حسين آل هويدي

عنوان الدراســـــة : طريق نحو تأليف أجزاء البرامج المتشابهة الخصائص

التخصـــــــــص : علوم حاسوب

تاريخ الشهـــــــادة: ٤ صفر، ١٤١٨

لقد تغيرت تقنيات تطبيقات البرمجة خلال السنوات الأخيرة تغيرا جذريا و أصبحت هناك تقنيات جديدة نتج عنها برامج أفضل. و لكن لازال هناك الكثير من التطبيقات الموروثة التي أنتجت بالتقنيات القديمة و لا يمكن الإستغناء عن هذه التطبيقات الموروثة بسهولة. تطرح البحوث العلمية عدة إقتراحات لحل مشاكل هذه التطبيقات. و ستطرح هذه الأطروحة إحدى الحلول، مع تطبيقها، و التي سوف تساهم في تخفيف ضراوة المشكلة المتفاقمة. هذه الفكرة مبنية حول تأليف أجزاء البرامج المتشابهة الخصائص. بعد التطبيق، هذه الفكرة ستساهم في تفهم البرامج الموروثة مما يساعد كثيرا نحو معالجة المشاكل، إعادة الإستخدام، و الترحيل و غيرها.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول و المعادن

الظهران، المملكة العربية السعودية

التاريخ

٤ صفر، ١٤١٨

This thesis is dedicated to my illiterate *mother* whose care, dedication, advice, wakefulness, children (my brothers and sisters) and many other factors enabled me to complete a Masters of Science.

ACKNOWLEDGEMENT

Acknowledgment is due to King Fahd University of Petroleum and Minerals for support of this research.

I wish to express my appreciation to Professor *JarAllah Al-Ghamdi* who served as my major advisor. I also wish to thank the other members of my Thesis Committee Professor *Mohammed Al-Mulhem*, Professor *Mohammed Shafique* and Dr. *Al-Zamil, Nabil.* Also, I wish to address my gratitude to all of those who helped me complete my studies, not to forgot those who proofread my thesis, i.e. Frank Housholder.

Contents

List of Figures

Chapter 1

Introduction

There are a lot of changes and challenges in the world these days. Technology is rapidly changing. Economics are fluctuating. Competition is reaching its peak. Therefore, there are many consequences accompanying these phenomena. The need to overcome or at least deal with these changes is indispensable.

Unfortunately, computer software, which is already complex, has been affected very much by these activities, which has made software even more complicated [Whit93]. This has caused the burden of the correction effort to be thrown over the shoulders of software engineers to plan for the future, attack current problems, and solve the mysteries of the past.

In general, software is difficult to manage and is rather sophisticated. Brooks compared software to a monster, and stated that the complexity of software is an essential property, not an accidental one [Broo87]. Even when it reaches a high stage of development, software may be unmanageable at certain points [Booc94], [Neum95],[Tile95].

Software engineering, which is the "Development and writing of software using engineering discipline and practices, rather than the inclination of the individual programmer" [Thro90], resolved some of the software problems. Although it is not easy to tackle the problem of software development and management, some existing techniques can reduce the severity of the problem, as stated in [Broo87] that "There is no royal road, but there is a road" [Booc94], [Neum95], [Tile95], [Wegs97].

Software engineering is the discipline that organizes activities associated with software from its release to its termination. *Reverse engineering*, is a branch of software engineering. The main purpose of reverse engineering is to enhance existing software to lengthen its life, or provide tools that support the reusability of its components. Moreover, reverse engineering can be integrated within the life cycle of a software during development of a new project [Chik90], [Benn95]. More about reverse engineering is found in Appendix B.

Code clustering is a subset of reverse engineering, which is used to bundle related (usually scattered) code segments according to certain criteria, such as strength of coupling among processes. Chapter 2 will be dedicated to code clustering and its tools due to its relevance to this thesis subject.

Similar-Characteristics Code Clustering (SCCC) is a special type of code clustering that promotes grouping subprograms that posses similar characteristics, such as user interface related code [Cowa95]. The main theme of this thesis is centered around SCCC methodology, which will be discussed in details in Chapter 4.

A legacy system is a complex software system [Booc94] that requires intensive software engineering practices, which in turn may involve reverse engineering to avoid crises or obsolescence [Snee95]. In many cases, code clustering is indispensable, especially when migrating legacy systems.

Dealing with legacy systems is a significant part of the whole problem. Legacy systems are affected by software complexity, technology, economics, people, and other factors that will be discussed later. The subject of this thesis will be centered around legacy systems, their inherited problems, and how to help alleviate the severity of the problem by applying *Similar-Characteristics-Code Clustering* (SCCC) methodology (Chapter 4).

A methodology for Similar-Characteristics-Code Clustering was developed for this thesis and a tool that is based on the methodology was implemented. This tool was used to analyze four actual projects. The result of the analysis is shown in Chapter 7.

The relationship among legacy systems, software engineering, reverse engineering, and code clustering is depicted in the Figure 1.1.

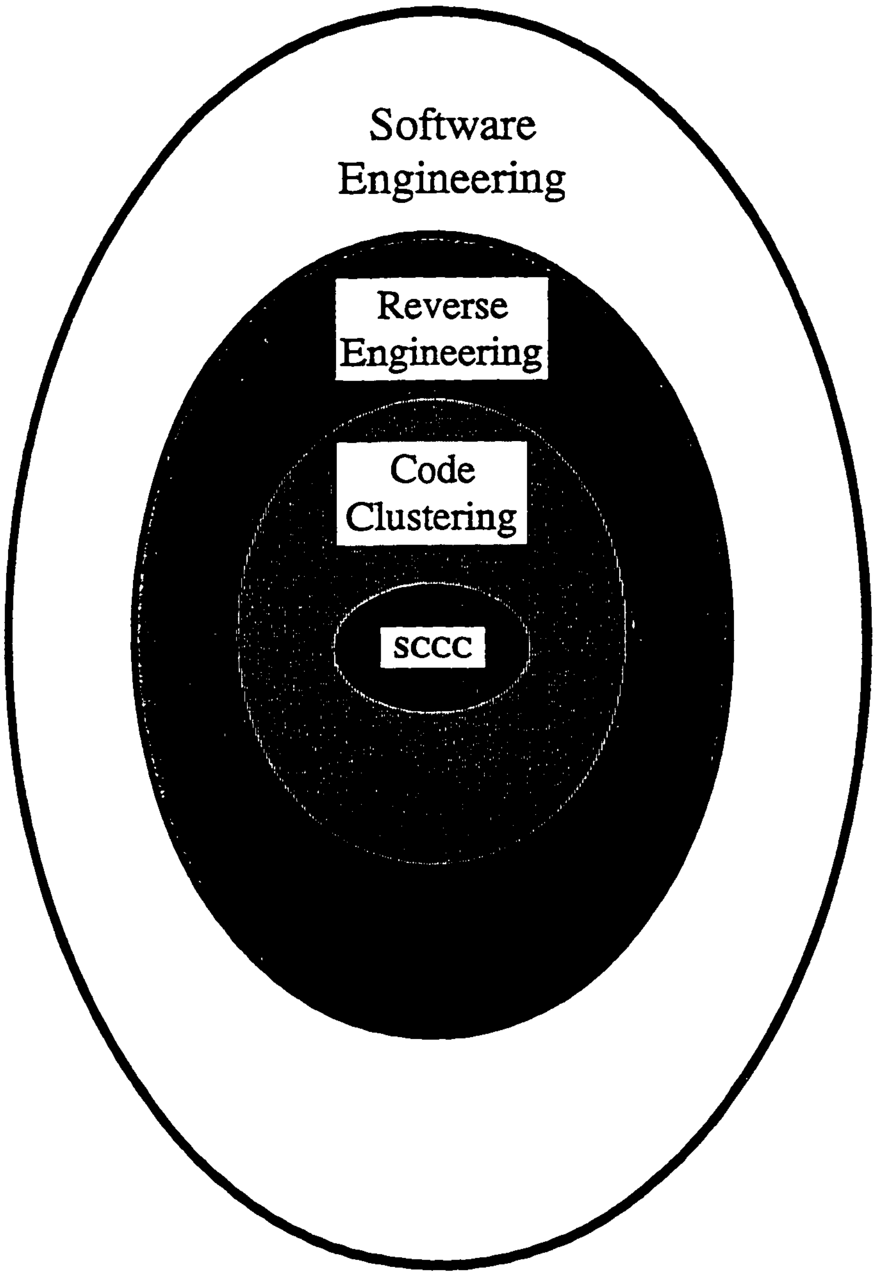


Figure 1.1: Proposal Perspectives

Chapter 2

Legacy Systems

A legacy system is usually one that is an integral part of a certain company and one that has been around for long time. Generally, it is a vital asset of a company, and without it the company may malfunction [Loud95]. A legacy system is a set of software systems that runs under set/sets of hardware to accomplish certain business needs [Ning94]. It is a large and complex software system that has consumed a significant amount of resources [Semm95]. It may consist of millions of lines of code, and it is usually developed and maintained by thousands of people [Aike94], [Ball96]. It needs ample and useful documentation, that in most cases is not present or may not be current. It contains valuable assets of critical business rules, that are embedded in the code and may not be documented anywhere else [Alop96], [Baum96], [Ricc95]. With time, the legacy system becomes outmoded and may not take advantage of newer technologies [Brod95], [Semm95]. At some point, its maintenance becomes neither practical nor economical [Ning94]. Its maintenance tends to be complex, expensive and may no longer be feasible [Brod95], [Gris95], [Semm95]. Most practitioner are opposed to rewriting a legacy system from scratch, which is neither feasible nor practical in many cases [Arno94], [Benn95], [Brod95], [Ning94], [Snee95], [Wong95].

I. Legacy System Problems

Throughout the definition of the legacy system provided earlier, a surrounding problematic atmosphere is tangible. Therefore, with time, the legacy system becomes more complex [Booc94], deficient and/or inadequate [Lyon95], [Mich95]. A legacy system becomes deficient because of factors that include:

A. Environment Instability

Environment instability is both *constantly changing business work-flow* [Adol96], [Char95], [Lyon95] and *rapidly growing technology* [Adol96], [Booc94], [Cimi96], [Davi95], [Lewi96], [Mona95], [Rama96], [Whit93]. Both of them require design modifications, people movement, functionality adds and drops, and hardware upgrades, which result into a complex migration process in order to recover the consequences. The migration process is not easy and susceptible to failure [Brod95], [Snee95].

A good current example of a simple environment instability, which is costing companies millions of dollars, is the two-digit year represented, or even stored, for the DATE data type. This problem surfaced while approaching the 21st century, where a two-digit year is not, any more, enough to represent the DATE data type. The consequences of implementing such a new need is quite sever [Mart97]. Assuming everything else was perfect, the user interfaces, database representation, and data manipulation have to be modified accurately and transparently without disturbing the work-flow of the end user!

Another example is that during early days of computers the hardware prices were cherished; meanwhile, they had very limited resources and confined computational power [Booc94]. Back then, the need to write efficient code and fully utilize these hardware systems was fundamental. The software at that time was less expensive and unsophisticated. Yet, nowadays, the table has been turned around. Hardware is cheaper and more efficient. Software is more involved and much more expensive.

B. *Poor, Inadequate, and/or Archaic user interface*

Due to their nature and rapidly changing technology, legacy system user interfaces become obsolete and require modification to comply with current environment and business needs [Baum95], [Cox 96], [Edge95], [Lyon95], [Mark95], [Wins95]. This case is perceptible during the migration process of a mainframe based legacy system, that most probably uses ISPF system as a user interface, to a client-server environment, that most likely uses a graphical windowing system (e.g. X-Motif or Windows NT).

C. *Cumbersome Data Manipulation*

In a legacy system, data manipulation is neither easy nor straight forward. In many old legacy systems, some of the data are handled through file management systems [Brod95]. Some other data could be scattered among hierarchical, network, and/or relational databases. Strangely enough, there are some data that are embedded within the code body. Handling and focusing around such data is really cumbersome. Besides, data manipulation code is not centered and, in most cases, scattered, which could further aggravate the situation [Adol96], [Edge95].

D. *Expensive and Impractical Maintenance*

Legacy system maintenance is costly, complex, and dreadful process that sustains exponential relationship with the size of any given system [Ball96], [Bank93], [Benn95], [Booc94], [Brod95], [Broo87], [Broo95], [Chik90], [Cons95], [Dods96], [Edge95], [Gris95], [Kasp94], [Lern94], [Lyon95], [Makc95], [Mark95], [Meye88], [Mona95], [Prem94], [Raz 93], [Semm95], [Sher96], [Star94], [Wins95]. Maintenance cost has been rising dramatically [Lern94]. More about maintenance will be discussed later.

E. *Continuous and Expensive Resources Requirement*

Due to technology and business needs, more resources requirement for a given legacy system rises. Some examples of new resources requirement are memory, more powerful workstations, communication links [Edge95], and many other resources. An example of some business needs that require expensive and high-tech resources is the interactive 3D earth modeling of subsurface in order to define oil reservoir boarder/shape. In order to be accomplished, this need will add to the hardware cost and software complexity. Technology, also, will influence vendors to rush (incomplete) features to their legacy systems, which will, later on, require a lot of maintenance and complication overheads [Brod95].

F. *Improper Documentation*

Documentation is one of the biggest obstacles faced in the industry of software development and maintenance, especially when dealing with a complex legacy system. This subject has been exhausted throughout literature and can be sought in many resources such as [Broo95], [Ingl94], [Ning96], [Sage95]. Usually, the documentation of a software system is either extravagant or insufficient. Even fair documentation lacks the components that are required for maintenance or reverse engineering [Canf92]. In many instances, prevalent technical data is absent, inaccurate, or outdated. Ingle and many others are nominating reverse engineering to ameliorate documentation in order to achieve improved productivity [Adol96], [Baum96], [Benn95], [Broo95], [Canf92], [Dods96], [Ingl94], [Mark95], [Ning94], [Patr95], [Ricc95].

G. *Software Complexity*

Computer programming, as it stands, is claimed to be the most complex responsibility ever undertaken by mankind. Besides, software complexity exhibits a nonlinear behavior that has been experienced by developers since the beginning of programming

[Budd91], [McLe96]. Software complexity is described to be software resistance to maintenance, modification and understanding [Zuse90]. Notwithstanding, it was stated earlier that a legacy system is a complicated set of software. Therefore, software complexity is blown up within a legacy system environment [Brod95]. Moreover, software complexity implies legacy system complications. Software complexity is increasing [Wegs97], meanwhile, there are basic human limitations to deal with such complexity [Booc94]. Anyway, legacy system complexity is encountered in most of the cases. There are many factors that amplify the complexity of any given software system. Some of these factors are:

1. Natural complexity

   Complexity of a software system is natural (essential) property [Broo87]. This point has been briefly covered in the introduction of this thesis, yet it is well known in both academia and industry. This point can be furthered researched throughout literature, such as [Abde96], [Brod95], [Broo87], [Booc94], and [Broo95].

2. Complexity of the problem domain

Externally, the nature of the problem (i.e. nuclear reactor) the legacy software deals with, is already complex and difficult to apprehend. Mapping this external complexity to software situation causes the arbitrary complexity [Broo87]. Also, users are not clear about what they want [Phil96], and developers may interpret what the users wanted differently. This drags the two disunited groups (i.e. users and developers) into a chaotic state [Booc94], [Wegs97].

3. Documentation state

Documentation and its associated problems have been briefly discussed in a proceeding segment, i.e. *Improper Documentation.* Another fact to add, however, is the way requirements are being addressed [Whee96D]. Requirements are usually expressed in terms of large mass of text, that is usually appended with some graphs. This kind of documentation is difficult to capture and susceptible to different interpretations [Booc94]. In addition, software development requirements are, regularly, ambiguously defined [Rama96]. They go under refinement process while the software system is being developed [Mull89], or in software engineering terminology, the software development undergoes an incremental process.

4. System growth and evolution

There are some problems imbedded in legacy systems that may not be solvable. For example, a legacy system growth cannot be stopped. Furthermore, a legacy system growth, exponentially, adds to cost, complexity, faults, maintenance, and risk [Booc94]. Moreover, legacy system growth decreases efficiency, correctness, robustness, modularity, compatibility, and creditability [Mack96], [Whit93]. Besides, legacy systems evolve over time [Booc94], which causes the system to be in inconsistent dynamic state [Benn95]. Commonly, unneeded functionalities are left over, which introduces maintenance and efficiency overheads. Removing obsolete functionality is also another overhead to be taken care of.

5. People and their variant experience/environment

During the development of any given software system, people, who group together, come from different disciplinaries to accomplish a number of objectives of a given project. Most of the involved people do not have the proper software engineering or even computer science background. Also, in the industry, there are many excellent[1] coders who do not adhere to good software engineering practices. However, people do contribute to the complexity of any system and they will still do that even in the future.

Another problem caused by human is people's varying level of expertise. It was mentioned previously that a single legacy system could be developed and maintained by thousands of people [Ball96]. Throughout the legacy system life, experienced people leave with their wisdom and undocumented knowledge, and new arrivals come green. Much of the needed skills has departed and may not be recorded anywhere. The newcomers have to learn, maintain, and face the facts of the system. Even experienced people may face problems with the newer technology, that they are not familiar with. Also, they may reject it due to their negligence of the its value. Another side effect caused by people to a legacy system is bad coding

[1] Excellent in terms of attacking problems and finding fast solutions, but not necessary the right way. Industry is full of such coders who do not believe in comments/documentation or adhering to certain

and improper/outdated documentation. Regardless, good developers are scarce [Brooc94].

Even if the previous problems were minimized, an incongruent atmosphere could very much appear [McLe96]. Congruence, in this context, reflects the amount of harmony and agreement among hierarchical organization employees. This interesting subject has been thoroughly covered in "Beyond Blaming: Congruence in Large Systems Development Projects" [McLe96].

6. Security

There are many simple problems in industry, especially in distributed environment, that have been made very complicated due to security rules and constrains. A simple example of such situation is password authentication among heterogeneous systems, i.e. MVS and UNIX. Large companies, who are gradually moving toward client-server environment, have most of their legacy systems and data resident in a mainframe environment. Developers and users life would have been much

guidelines. Some excellent coders intentionally write code in a vague manner due to job security simpler if some of the security rules had been waived. In one hand, developers have to write either complex code to get through TCP/IP protocols or encrypt/decrypt passwords among these heterogeneous systems, which is rejected in many organizations anyway. In the other hand, users have to remember and keep track of many different passwords, which they usually get confused, in order to accomplish simple tasks. Users have to execute different programs in different platforms, then consolidate their result into a common place, usually using FTP. This activity cost 70-80% of users time as being claimed in the industry [GeoQ96]. Security restrictions seem to be simple at the surface, yet they do stand to be one of the most difficult stumbling blocks that developers and/or users ever face [Luca96].

7. Efficiency

Efficiency adds to complexity [Benn95], [Mack96]. It is known that the most efficient code could be written in assembly language. Nevertheless, industry is moving away from lower to higher level of abstraction for the sake of simplicity and other attributes [Faya96], even if efficiency is

reasons.

scarified. This can be seen in fourth generation languages and object-oriented paradigms that require more resources and time to execute. After all, with the current advanced technology and cheap-efficient and fast processors, human time is considered to be the most precious, unlike those days when computers were primitive and slow yet priceless.

A good contemporary example of the problem of efficiency is *C++ Constructors* and *Destructors*. *Constructors* allocate space and initialize variables during objects creation. *Destructors* do the opposite when freeing up objects. *Destructors* are good clean-up tools; however, destruction is an expensive and time consuming operation. Efficiency seeking developers will bypass *Destructors*, which may cause a lot of confusion and memory leaks, which in turn leads to deficiency due to unattainable memory [Whee96J].

8. Programming environment/language(s)

Programming environment may add very much to the complexity. A good example of this is the DoD environment in terms of hardware and different high/assembly languages to drive these sets of hardware [Aike94], [Mack95]. Moreover, it is believed that a single programming language cannot support all the business needs [Moor94], which forces organization to use multiple languages. Some examples of such a case will be covered in later discussion(s). Likewise, it has been claimed that the structured programming falls apart when the application size exceeds 100,000 lines of code. Along with, structured design does not properly scale up with extremely complex systems [Bart94], [Booc94]. It is known that the legacy system size tremendously overruns this number. Moreover, most legacy systems lack the structured architecture [Ning94]. This leaves software engineers attack both the environment, that does not fit, and the legacy system itself. Furthermore, it is well known among software engineers community that the C programming language influences its users to abuse the language, which introduces ambiguity and documentation problems.

There are also some environmental problems that look vincible at the surface, yet they cause a lot of annoyance and incongruity. An example of such problems is the terminology inconsistency, as stated that "... the company used different terminology for the same item; payroll was called "pay" in one program, "salary" in another, and "wages" in a third ..." [Pfle96].

9. Software flexibility

Software offers the ultimate flexibility. Correspondingly, there are no set of common standards or rules that can be implemented. Most developers are inclined to do things their own way, especially if they are under pressure. Flexibility is required (i.e. this is why technology is lending itself toward software-device-drivers instead of hard-coded chips). However, flexibility creates uncertainty and reduces predictability[Booc94], like throwing a ball of soft clay not knowing the shape it will take whenever it lands.

H. Lack of Proper Process

Software processes, especially proper ones, are essential part of a software life cycle, however they are not easy to define or implement. One process may fit certain software project but it may not be proper for another [Boeh88], [Boeh96], [Booc94], [Ridd89], [Tull89]. Barry

Boehm is well known to thoroughly cover this subject in many articles throughout literature, two of which are [Boeh88] and [Boeh96]. Generally, a complex legacy system requires more developers/maintainers than a regular software. The communications of such large group tend to be complicated and overhead to the software process [Booc94], [Lede92], [Luca96], [Whee96D].

I. Competition

Currently, software industry is in a competition climax. Most software houses that support some of their own legacy systems are contesting with other rival companies. Most of the competition centers around addition of new features and enhancement of user interfaces, yet the basics of software engineering principles are violated. Competing companies build these new features over originally ill-designed legacy systems. Reengineering a legacy system is time consuming; besides, the success of the reengineering process is not guaranteed [Adol96], and this may leave a particular company behind which makes it loses market shares and business opportunities [Brod95], [Luca96]. This is why most vendors come to the conclusion to built over the already existent legacy system. Another reason these companies do not bother much is that the end-user of their legacy system is not aware of the design or the maintenance of the system, which is in fact transparent to him/her, as long as it delivers a certain functionality the user hopes for. Besides, end-users are not that sophisticated much to know the difference. In many occasions, their evaluation of a piece of software is concentrated around appearance and functionality availability, not to say these are not important features. This is analogous to a car driver who cares most about cosmetics of the automobile and totally forgets efficiency, design, tolerance and many other important factors. The competition repercussion can be felt during software exhibits and/or support.

There are finer ingredients of impediments that consolidate (intersect with) one or more of the obstacles mentioned previously. Some of these ingredients can affect more than one major obstacle, e.g. budget, time, manpower are some common factors among most of these obstacles. It is worth noting that the impact of these obstacles is quite sever in many cases. Consulting the referenced resources can help relate the impact of these obstacles over legacy systems.

Academia and industry are aware of these obstacles and they are dealing with these obstacles effects as will be seen in the coming "Solutions" section. Then, a methodology to reduce the effect of these obstacles will be discussed in later chapters.

II. Solutions to Legacy System Problems

Literature and industry proposed a number of solutions to tackle problems of legacy systems. The panacea to these problems is not there yet, however this should not prevent searching for better ways to improve what is currently available nor come up with a new process that revolutionizes industry [Kasp94]. Some of the available solutions will be discussed shortly.

It has been noted that some of these solutions reduce the austerity of associated problems and may extend the life of a legacy system or improve reusing it or its components. However, some of these solutions have their own short-comings. Following is a brief summary of these solutions and their associated problems:

A. *Maintaining the legacy system*

1. Definition

Maintenance can be divided into three parts: *adaptive*, *perfective (enhancement)*, and *corrective* [Snee95]. Maintenance consists mainly of modifications that fix design defects, add incremental functionality, or adapt changes in the used environment or configuration [Broo95]. Around 70% of a project cost is consumed by maintenance [Meye88]; whereas, Brooks states that 40% or more of the budget is used by maintenance [Broo95].

2. Advantages

Maintenance is an indispensable process within any legacy system. Without proper maintenance, any system could fall apart. Maintenance should keep the legacy system in shape and minimize many of its faults. Sometimes, maintenance is used to improve performance or other attributes [Bank93].

3. Disadvantages

a) Maintenance is difficult and expensive [Ball96], [Dods96], [Edge95], [Kasp94], [Mark95], [Prem94].

b) Maintenance cost has been rising dramatically [Lern94].

c) Maintenance is highly impeded by improper or lack of

(1) documentation [Mark95],

(2) documentation tools [Star94, Wins95],

(3) manpower [Lyon95],

(4) and adequate processes [Sher96].

d) Maintenance degrades with time [Snee95].

Due to its complexity and cost, maintenance is protested by many practitioners who suggest the use of reengineering or other tools (such as inspection [Acke89]) [Cand96], [Cons95], [Lern94], [Mona95].

B. *Redeveloping the legacy system*

1. Definition

Redevelopment is the act of rewriting a new system from scratch without tacking into account the existing legacy system [Adol96], [Lem94].

2. Advantages

Redevelopment from scratch provides the advantage of learning from the past development(s). Redevelopment, most probably, will produce a more efficient and cleaner code. With redevelopment, dreadful maintenance could be enhanced. Also, redevelopment can take into account newer technologies, which can dramatically improve the status of the current system.

3. Disadvantages

The redevelopment activity is observably protested by many practitioners who believe that:

a) Redevelopment is too costly and impractical process in most cases [Arno94], [Brod95], [Mack95], [Prem94], [Tile95].

b) Redevelopment may react to a current need, meanwhile, not taking into account future needs, which may need another redevelopment [Brod95], [Lewi97].

c) Redevelopment requires a complete shutdown of an already working system, which may not be affordable in many cases [Brod95].

d) Redevelopment does not guarantee a perfect system. Some new problems, due to technology, for example, could very much appear [Brod95].

e) Failing to successfully complete a redevelopment project could cost a lot of assets [Brod95].

C. *Migration*

1. Definition

Migration is the process of migrating a system from one platform to another due to technological, economical, or new business requirement [Brod95]. The cost of migration can be minimized by reusing as much as possible of the code and design of the original system. Applying reusability can save a lot of time and money as well as producing better product [Card94].

2. Advantages

Rarely it is feasible to run an old legacy system over a new environment without modifying the system. Migration has the advantage of preparing a legacy system into a new environment. It provides the advantage of exploiting newer technologies. It also provides a good chance to improve the old system.

3. Disadvantages

Migration is expensive and labor intensive process, which may not necessarily succeed [Snee95]. There are many stumbling blocks during the migration process [Adhi96], [Baum96], [Edge95], [Mack95], [Mona95].

D. Wrapping

1. Definition

Wrapping is the process of building a GUI (graphical user interface) around a legacy system [Patr95]. Wrappers ought to be targeted toward a specific legacy system in order to enhance/ease access to legacy functions. However, wrappers are not general solutions for every system [Wins95].

2. Advantages

Wrappers have the advantage of preserving the legacy code and data, and interfacing with users under their environment [Brod95], [Lyon95]. Wrapping is a cheaper solution than redevelopment and/or migration.

3. Disadvantages

Wrappers are supposed to be temporary solutions because inflexible-legacy systems are aging underneath wrappers [Wins95]. Wrappers are used as transient state of migration. Wrappers have integrity and performance problems. Probably, the most difficult part, when implementing wrappers, is identifying the code that should be wrapped [Lyon95].

E. Integration

1. Definition

Fitting a legacy system with a new environment is an integration process, which is rather complex [Cimi96].

2. Advantages

Integration is proposed because legacy systems are staying and are not going away soon [Loud95], and the only way to preserve a legacy system is integrating it with the new environment.

3. Disadvantages

Integration is full of complexities and very costly, yet it is indispensable in a heterogeneous environment [Cimi96], [Davi95].

F. *New Paradigm Adaptation*

1. Definition

New paradigm adaptation is the transition from conventional software engineering practices to object-oriented software engineering [Faya96], or the transition from older to newer disciplines [Selk96a], [Selk96b]. New paradigm adaptation is, somehow, the process of changing the way one thinks [Booc94], [Budd91], [Mull89]. It is a new way of viewing the world [Budd91].

2. Advantages

New paradigm adaptation came to reduce the problematic effects of older methodologies, e.g. enhancing maintenance and reusability [Booc94]. When applied well, paradigm shift pays well [Faya96]. Indeed, with growing complexity, paradigm shift is mandatory [Booc94].

3. Disadvantages

New paradigm adaptation is a real challenge since people are inclined to get attached to their original concepts [Booc94], [Budd91], [Clin95]. Besides, it is not just a process of translation that could be easily translated or mapped (i.e. it is not one-to-one rendering [Mull89]) which indicates that automating the process is not handy yet. Furthermore, paradigm shift is not simple and it requires real qualified experts to do the transition [Faya96]. Also, it was stated that it is a difficult shift for experienced developers; meanwhile, it is much easier with novices [Budd91]. In reality, experienced people and expertise are needed to handle the shift smoothly and successfully. Recruiting people with proper expertise in the field costs.

G. *Data Warehousing*

1. Definition

Data warehousing sterilizes data, strips it out of application context, and then isolate it behind one or more relational database engine. It is an architecture that separates applications from their data yet provides seamless integration. More about data warehousing can be found in [Ecke95a], [Ecke95b], and [Hard95].

2. Advantages

This new technology has exited to confront many current problems found in legacy systems and operating environment, and provide cleaner atmosphere for future data manipulation.

3. Disadvantages

There are no disadvantages to data warehousing except the cost and complexity associated with the task.

H. Reengineering

1. Definition

Reengineering is a discipline that restructures existing code without functional changes to the system [Ingl94]. Some practitioners consider reengineering to be one of the latest trends in the information systems field [Cand96], [Luca96]. Some other practitioners define reengineering to be "the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance such as cost, quality, service, and speed" [Hamm93], [Cand96].

2. Advantages

   a) reducing maintenance cost by making the system more understandable [Snee95], [Cand96], [Lern94],

   b) better cost estimation due to availability of better metrics, such as the number of lines of code [Snee95],

   c) better testing process due to availability of real data [Snee95],

   d) parallel processing of the reengineering tasks [Snee95],

   e) quality improvement [Snee95],

   f) easier to out-source, which leads to many other advantages, such as cost estimation [Dede95],

   g) easier migration process [Snee95],

   h) greater reliability achievement [Snee95],

   i) functional enhancement [Snee95],

j) fault discovery [Snee95].

3. Disadvantages

a) Reengineering is not straight forward as it appears. It is a risky process [Luca96] that requires proper expertise that may not be present.

b) It is significantly affected by the quality and state of documentation. The assumption that reengineering a legacy system is simpler or more successful than building a new system from scratch is questionable [Adol96].

c) Furthermore, the reengineering business is very difficult to justify [Snee95].

I. Software Reverse Engineering (SRE)

Reverse engineering is a superset of code clustering and it has been nominated to be the most promising solution of solving legacy system problems [Adol96], [Aike94], [Baum96], [Benn95], [Broo95], [Canf92], [Dods96], [Ingl94], [Mack95], [Mark95], [Ning94], [Patr95], [Prem94], [Ricc95], [Sage95], [Samu90], [Semm95]. More about software reverse engineering is discussed in Appendix B.

1. Code Clustering

a) Definition

Code clustering is the concept of grouping together related processes of the source code. These processes are related by having either similar characteristics or interacting closely with each other [Raz 93]. More about this will be covered later in a stand-alone chapter.

b) Advantages

Code clustering contributes to reducing maintenance and improving effectiveness of development [Raz 93].

c) Disadvantages

Requirement of budget, time, resources, and expensive expertise.

2. Reusability

More about reusability is explained in Appendix C.

It is worth noting that some of the previous solutions are inter-related. Accommodating one of them, e.g. new paradigm, can help with others, e.g. maintenance. Later, in a different chapter, it will be seen how similar-characteristic code clustering can help the flow of these provided solutions.

Many tools have been written to assist dealing with legacy system problems. Most of these tools adapted reverse engineering discipline to reach their goal, and subset of the latter have been written to directly attack badly/unorganized written code by using code clustering methodology, which along with two of its tools will be examined and discussed in the "CODE CLUSTERING" chapter since they are related to the subject of this thesis.

Chapter 3

Code Clustering

It was stated earlier that code clustering is the concept of grouping together related processes of a computer, usually high level, language source code. These processes are related by either: 1) having similar characteristics, or 2) interacting closely with each other (i.e. strongly coupled). Code clustering is thought to contribute to reduced maintenance and improved effectiveness of software development [Raz 93].

Similar characteristics processes will be thoroughly discussed in the methodology chapter (Chapter 4); whereas, a brief discussion about coupling and cohesion is presented in Appendix D.

Tools of Code Clustering

There are some available tools that aid code clustering source code. These tools can be classified as:

1. Specific; or
2. General

A *specific* tool is a tool that was developed to aid code clustering a particular project. This project may be either a very large project that is written in more than one high level language for multiple platforms, or a small to medium size project that is written only in one specific language for a distinct platform. A combination of both may exist as well. An example of a specific tool, not necessarily addressing code clustering alone, is given in [Aike94] concerning reverse engineering DoD legacy systems that have more than 1.4 billion lines of code with thousands of heterogeneous information systems located at more than 1,700 data centers. The mission of the previously defined project was to migrate legacy systems scattered among heterogeneous systems into homologous target systems. Another example is the case demonstrated in [Mack95] "Software Migration and Reengineering: A Pilot Project in Reengineering." This project relates to military equipment that involve upgrading both hardware and software. Most of the code was written in assembly language for

16-bit microprocessors. The migration mission was to convert the code to Ada programs that run on 32-bit microprocessors. Although this is more of a migration process, this event is a core classical case of reverse engineering of large systems that involved both hardware and software, where code was not the paramount concern.

A *general* tools is a tool that was not developed for a specific project. However, usually a general system has to be written for software implemented in a certain high level language. In the meantime, there is no general system that would work for all the languages or even few languages, even though having a general system to work according to a high-level language grammar is attainable . An example of a system that addresses COBOL is Cobol/SRE found in "*Automated Support for* **Legacy Code Understanding**" [Ning94].

However, there are algorithms that are not based on a particular language. Such algorithms give a general idea of how to solve the problem of code clustering. An algorithmic example is the "Process Clustering with an Algorithm Based on a Coupling Metric" encountered in [Raz 93].

Two code clustering tools are presented here as examples. The first one has been chosen because it is a general algorithm that disregards the nature of the source code. The second tool is a tool that was written specifically for COBOL source code. It has been chosen because of its generality and the intersection of some issues with this thesis-proposed tool.

1. Process Clustering with an Algorithm Based on a Coupling Metric

Raz and Yaung emphasized the importance of process management [Raz 93]. They assumed that grouping related processes (i.e. subprograms) would lead to better system control.

Process management is not straight forward. Process management requires a lot of efforts and resources. To minimize these efforts, processes should be sorted and placed in groups that possess similar characteristics, (i.e., user interface processes), or those that closely interact with each other, (i.e., subprograms that share data) [Raz 93].

Raz and Yaung placed emphasis on the inter-process linkages and contributed very little to those processes that possess similar characteristics. They criticized the previous work regarding the same subject as being either not accurate or not addressing the clustering problem properly [Raz 93].

Therefore, Raz and Yaung suggested a clustering algorithm that would produce better results than previous algorithms that may produce objectionable solutions, as these algorithms were criticized by [Raz 93]. The suggested Raz algorithm runs in the order of $O(n^3)$, where n is the number of processes.

A linkage matrix reflects the connectivity between two processes as "1", and no-connectivity as "0". Process *A* may have a link with process *B*, but it is not necessary to have a link from *B* to *A*, although this case may happen.

An example of a linkage matrix *M* is as follows:

M	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	1	1
3	1	1	0	0	0
4	0	1	1	0	1
5	0	0	0	1	0

After getting the matrix *M*, column and row elimination starts by taking a 2 X 2 sub-matrix *M'* and merging the higher index number into the lower one. Sub-matrix *M'* will look like the following:

M'	1	2
1	0	1
2	0	0

After combining column/row 1 and 2 into column/row 1, *M* will produce M_1. The formula used to produce M_1 is:

$M_1[\text{ I }] = M[\text{ I }] + M[\text{ I+1 }]$, {Matrix addition}

M_I[J] = M[J] + M[J+1], {Matrix addition}

where M[I] represents a complete row in M and M[J] represents a complete column in M. In the first iteration, I and J will be given the value of "1".

M_I	1	3	4	5
1	0	0	2	1
3	2	0	0	0
4	1	1	0	1
5	0	0	1	0

It should be noticed that only column and row 1 have been affected due to borrowing values from column/row 2, which disappeared in the new matrix M_I.

The process continues until a desired result is consummated. Then a transformation of the result matrix is applied to get a coupling matrix. For example, the transformation of matrix M_I will produce the coupling matrix C_I. The steps required to get from M_I to C_I are as follows:

$$C(I,J) = (m_{IJ} + m_{JI}) / 2 \tag{1}$$

$$C(I,J) = (m_{IJ} + m_{JI}) / (2 \text{ X } n_I) \tag{2}$$

$$C(I,J) = (m_{IJ} + m_{JI}) / (2 \text{ X } n_I \text{ X } n_J) \tag{3}$$

C_l	1	3	4	5
1	x	0.5	.5	0.0
3	-	x	0.5	0.75
4	-	-	x	0.0
5	-	-	-	x

Both M_l and C_l will contribute to the repetitive algorithm to get to the final result.

The final result will decide which processes should go together according to their coupling values. The details of this algorithm are found in [Raz 93], where the algorithm is explained thoroughly with some industry examples.

The Raz and Yaung algorithm is easy and good only for processes that interact closely to each other.

A drawback of the Raz and Yaung algorithm is that it is a pseudo code algorithm that does not specify a target language, which is a plus sometimes, and it does not provide a tool either. The algorithm is just providing a base for a useful tool.

2. Cobol/SRE

One of the useful tools for COBOL source code clustering is Cobol/SRE illustrated in [Ning94]. This tool is comprehensive and complete when used for code clustering COBOL source code. This tool was presented after thorough researching, and came to solve a wide spectrum of the problems faced when reverse engineering a COBOL legacy system.

Cobol/SRE, as depicted in Figure 3.1, consists of six subsections:

1. Workspace management
2. System-level analysis
3. Data model recovery
4. Concept recognition
5. Program-level analysis
6. Distributed execution architecture

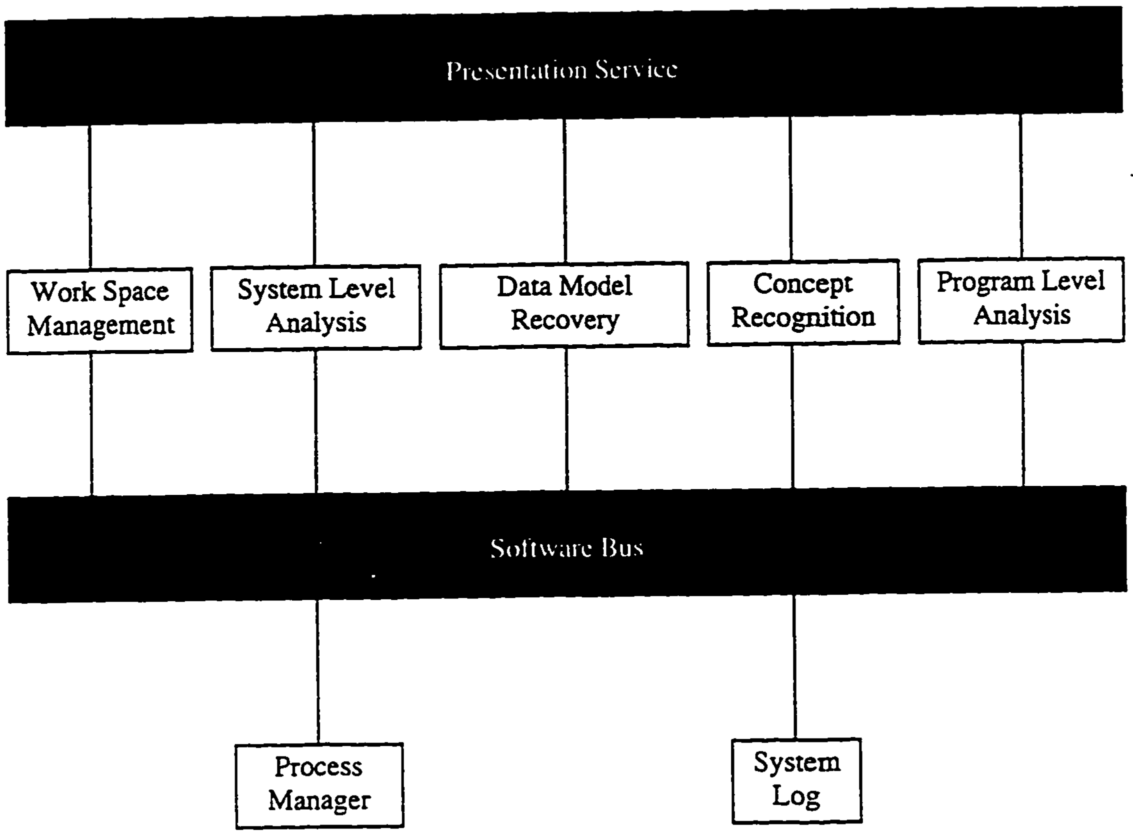


Figure 3.1: Cobol/SRE[2]

[2] This figure has been adopted from [Ning94].

Workspace management is used to manage a working team.

System-level analysis holds information about COBOL source code libraries and manages relationships among modules.

Data model recovery is a virtual data model of a system where Cobol/SRE builds segments according to this model.

Concept recognition is a knowledge-based technique to automate the recognition of functional patterns.

Program-level analysis is a code level parser to help automation.

Distributed execution architecture uses the *software bus* to facilitate process integration.

Cobol/SRE uses two major steps: *focusing* and *factoring*.

Focusing will cluster related segments that are scattered around a program. *Focusing* can be done according to select statements, COBOL PERFORMed statements, Condition-Based Slice, Forward slice, and Backward Slice.

Factoring clusters focused code segments into independent modules. In Cobol/SRE, *factoring* shows/sets entry and exist points of subprograms, and allows extracting segments in three modes: a callable subprogram, an independent program, or a text file containing all the statements in the segment [Ning94].

Figure 3.2.a shows the Cobol/SRE process flow, while Figure 3.2.b explodes Cobol/SRE to show its internal processes.

Cobol/SRE has the following limitations:

1. Language specific (COBOL)
2. It does not have precedence among segments if they intersect, although it handles Union, Intersection, and Difference operations
3. It works only on interactive mode and requires human operation
4. The provided interface can handle one piece of code at a time

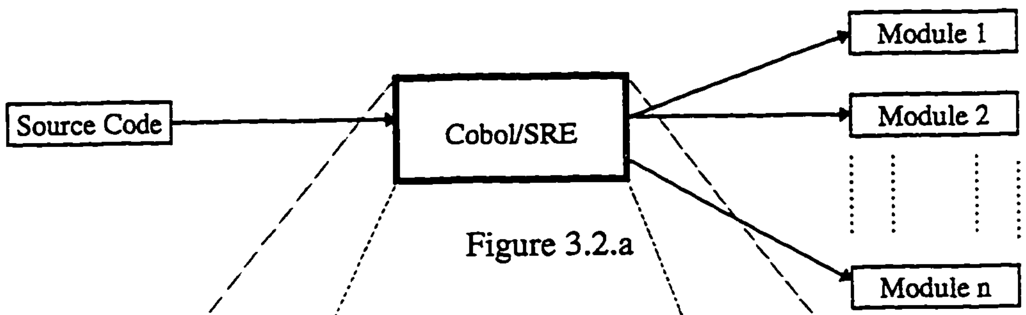


Figure 3.2.a

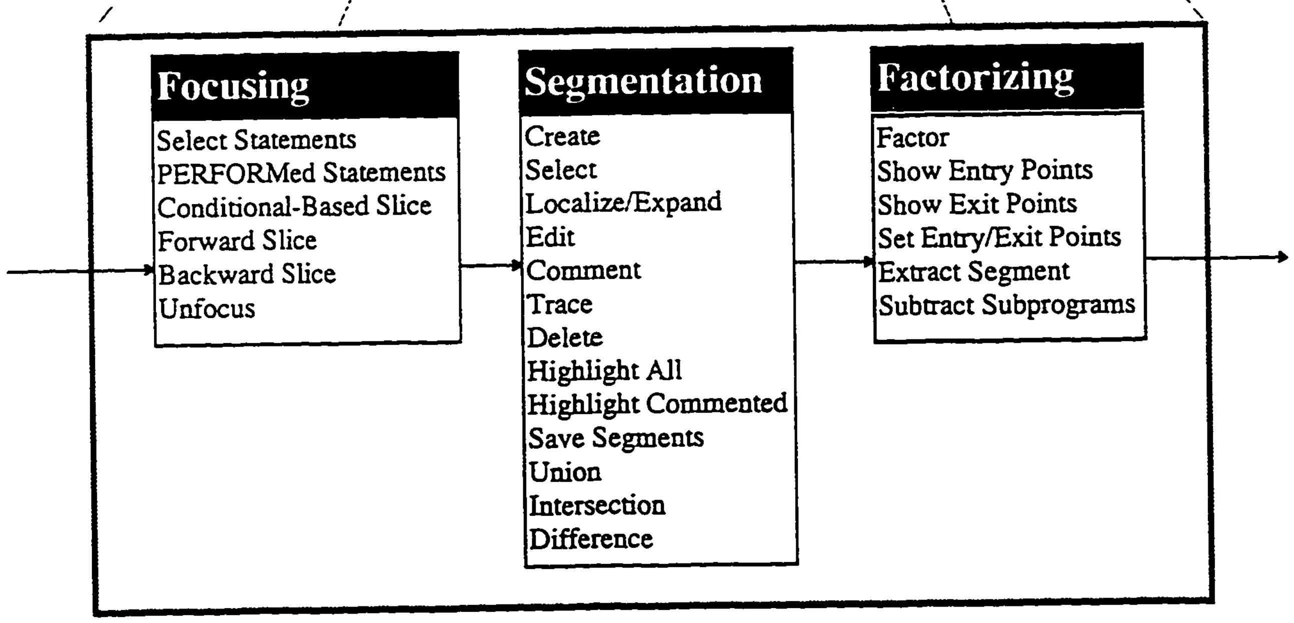


Figure 3.2.b

Figure 3.2: Cobol/SRE process flow

Discussion

There have been few attempts at tool development that would help with reverse engineering and its components (i.e. code clustering). Most of these tools were written for a specific system to be reengineered. Nevertheless, The C Abstraction System found in [Chen90] is a ubiquitous tool that runs under UNIX for the C language programs. This tool works as a fully blown database of C source code that accepts different queries regarding routines or variables. In this tools there are indices for all routines, variables and relations among process/variables. For example, if a user wants to collect all related variables for a function, the user can do it with a query. Then the user can redeem a particular function as one entity.

There are other general tools that exist; however, they are tailored for non-commercial languages, e.g. [Canf92], which proposes a general solution for Pascal software systems.

In summary, code clustering is a major part of reverse engineering a legacy system. Some tools have been implemented to aid in code clustering. Some of these tools are explicit and others are general. Two tools have been discussed previously: Process Clustering with an Algorithm Based on a Coupling Metric and Cobol/SRE. Due to the complexity of the process of code clustering, many existing tools suffer due to some limitations that make them inept for certain circumstances. The next section discusses the limitations of code-clustering tools.

Limitations of Some existing Tools of Code Clustering

Code clustering tools are fairly new [Arno94]. Consequently, there are many difficulties that accompany any new technology. It is not easy to implement a new technology. Furthermore, its benefits are not clear [Wino95].

Since code clustering has not matured yet, some limitations and causes of limitations (obstacles) exist with the currently-available tools [Cand96], [Canf92]. Some of the causes to limitations (obstacles) are:

1. Lack of knowledge acquisition

   In a large project, knowledge needed for reverse engineering to apply code clustering over a legacy system is scattered and difficult to converge to a focal point. Neither the naked-eye nor the human brain can deal with a huge project to acquire needed knowledge [Brod95]. Automation is hindered because of the difficulty in knowledge acquisition, that is to identify and specify [Ning94].

2. Complexity of concept matching process

   Program comprehension is acclaimed to be the most time-consuming chore in systems maintenance and reengineering [Wood96]. Concept recognition is a knowledge-based methodology grown to automate the recognition of functional patterns in the code. Concept-matching is not an easy matter to comprehend [Ning94].

3. Functionality separation

   Functionality separation is identifying a functionally complete piece of logic that is functionally related yet de-localized throughout old code [Ning94].

4. The state of documents

   Usually, the documentation of a project is either extravagant or insufficient. Even fair documentation lacks the components that are required for reverse engineering [Canf92] in order to perform code clustering. In many instances, prevalent technical data is absent, inaccurate, or outdated.

5. Bad coding

Bad and old coding cause frustration in the use of current tools. Handling such cases is not an easy task and requires intensive human interaction. Besides, it is more difficult to identify the functionally-related pieces of bad/old code [Ning94].

Bad and unclear coding is not confined only to old programs. A contemporary C++ example has been borrowed from [Whee96J] to illustrate the vague style that well-known developers are still practicing:

```
char * strdgdup (char * s1)
{
 if ( s1 == NULL) return (NULL);
 int strgSiz = strlen(s1);

 char *dup = new char [strgSiz + 1];

 for (int j = 0; dup[j] = s1 [j]; j++)
                ; // All work done in test expression

 return(dup);
}
```

"dup[j] = s1 [j]" within the for loop is both evaluation and testing construct that is not clear from the first glance, and it may never be clear for those who apply good coding styles and standards.

Code clustering tools limitations include:

1. Human intervention

   Code clustering tools require heavy human interaction and expertise in order to analyze a software system or to get a buried-desired functionality.

2. The assumption that the code exists only in one flat file

   Most of the current tools assume that the code exists in one monolithic large file. Code clustering, such as Cobol/SRE, tools can be used to fragment such a file. There may be many functionalities scattered around many different files that require clustering into one file so that the tool can be used.

3. Inflexibility of tools

   Most of the current tools are inflexible and tailored toward a specific product or project [Canf92].

Chapter 4

Similar-Characteristics-Code Clustering Methodology

The methodology of this thesis was earlier proposed to reduce the effect of the obstacles, mentioned earlier in the legacy systems and following chapters (i.e. environment instability, user interfaces, data manipulation, maintenance, documentation, and others), and to enhance the flow of the proposed solutions recommended by literature and industry (i.e. maintenance, redevelopment, reengineering, migration, warping, integration, new paradigm adaptation, and others).

This methodology is centered around a specific type of code clustering, that is similar-characteristics code clustering (SCCC), analogous to *opportunistic strategy* of code understanding discussed in [Litt86], that is the study of code on an as-needed basis [Mayr96]. SCCC can also support *systematic strategy* discussed in [Koen91] and [Litt86], that is the strategy of studying line by line of blocks of code while maintaining rational representation at higher and higher levels of abstraction [Mayr96]. Besides all of that, SCCC methodology can support *dynamic code comprehension* proposed by [Mayr96].

Similar-characteristics code clustering methodology also has proposed the division of source code into manageable number of intersecting subsets highlighted in [Abd-96], [DeMa96], [Hsia96].

In addition to the methodology effort, this thesis proposed a tool (Chapter 5) that was based on the methodology. Part of the tool was developed and implemented to illustrate the usefulness of the methodology and tool (Chapter 6). To demonstrate the usefulness of the tool, four different case studies have been used to cluster and analyze their code (Chapter 7).

Similar Characteristics of Code

There are plenty of similar-characteristic classes embedded in a legacy system code, e.g. user interface [Cowa95], file management, database connectivity calls, and many others. Characteristic-wise, the definition and discovery of these similar-characteristics is categorized to be: *objective*, *subjective*, and *cognitive.*

Following are some similar characteristics possessed by code:

A. *Objective characteristics*

1. Low level language code

2. High level language code

    a) **Dialects [Mark95]**

    b) **Declarations**

        (1) Intrinsic type declaration

            (a) Basic types

                (i) Integer

                (ii) Boolean (Logical)

                (iii) Float

(iv) Pointer

(v) Character

(b) Compound types

(i) Arrays

(ii) Records (Structures)

(c) Scope

(d) Storage class

(2) User defined types

(a) Scope

c) Constructs

(1) Exception handling

(2) Subprograms

(a) Functions

(i) Functions that return a certain type

(ii) Functions that use certain constructs

(iii) Recursive functions

(b) Subroutines (Procedures)

(i) Argument-less subroutines

(ii) Subroutines that accept certain number of arguments

(3) Loops

(a) While

- *(b)* *For*
- *(c)* *Do while/until*
- *(d)* *Goto*

(4) I/O

- *(a)* *Print*
- *(b)* *Read*
- *(c)* *Write*
- *(d)* *Rewind*
- *(e)* *Open*
- *(f)* *Close*
- *(g)* *Seek*
- *(h)* *Create file*
- *(i)* *Delete file*
- *(j)* *Lock file/record*
- *(k)* *Buffer manipulation*

(5) Concurrent

- *(a)* *Fork*
- *(b)* *Wait*
- *(c)* *Child*
- *(d)* *Parent*

(6) Logical

- *(a)* *If statement*

(b) *Case (Switch)*

(c) *Logical operators*

(7) Macros

(a) *Definition: #define <macro>*

(b) *Usage*

d) Calls

(1) Dependent calls

(a) *Platform dependent calls*

(b) *System dependent calls*

(c) *Package dependent calls*

(i) Graphic package calls

(ii) Math package calls

(iii) User interface package calls

(d) *Library dependent calls*

(e) *Remote procedure calls (RPC's)*

(f) *API calls*

(g) *Graphic calls*

(h) *Database calls*

(i) Data manipulation calls (DML's)

(a) Insertion calls

(b) Updating calls

(c) Queries

(ii) Data definition calls (DDL's)

(a) *Table definition calls*

(b) *Rules calls*

(c) *Data dictionary calls*

(2) Non-dependent calls

(a) *Intrinsic calls*

(i) Math intrinsic's

(ii) I/O intrinsic's

(iii) Text manipulation intrinsic's

(iv) Memory management intrinsic's

(b) *Language I/O calls*

(c) *Internal calls*

(d) *External calls*

(e) *Macro calls*

e) Resource management

(1) Code that dynamically allocates memory

(2) Code that does not free up dynamically allocated memory

(3) Code that allocates/frees general resources

f) Miscellaneous code

(1) Code that manipulates certain variables or memory addresses

(2) Code with no comments

(3) Dead code

(4) Redundant code

(5) Code that uses certain operators

B. Subjective characteristics

1. Code to be wrapped
2. Front-end code
3. Static/Dynamic code generator segments
4. Code that allocates many resources
5. Active Code
6. Efficient code
7. Deficient code
8. Segments of code created by a certain developer
9. Segments of code updated/modified by a certain developer
10. Segments of code that were created/modified at a certain date
11. Report generation code
12. User interface code [Cowa95]
13. Code that uses expensive operators

C. Cognitive characteristics

1. Code that performs a certain algorithm [Lewi81]
2. Code that parses code
3. Coding tricks [Adol96]
4. Application management code [Adol96]
5. Code that will have large effects to improve capacity [Mack96]

The formerly mentioned characteristics are not exhaustive.

Algorithmic-wise, the definition and automatic discovery of previously mentioned characteristics can be categorized to be:

A. Easy

Some of the similar-characteristics are algorithmically easy to identify, such as language keywords, platform/system dependent calls, intrinsic's, library calls, and many other similar characteristics. This category mostly fall within the *objective* domain, which is very simple to identify and implement.

B. *Difficult*

There is another category, that exhibits similar-characteristics, which are not easy to identify (i.e. code to be wrapped [Lyon95], and coding tricks [Adol96]). This category mostly fall in the *subjective* domain. A good example that represents such a subjective case is searching for large-size arrays. Whatever is considered large is a subjective matter that depends upon the computer era, platform type, mood of developers, and some other factors. This category could be identified by some human interaction.

C. *Impossible*

There are some sets of characteristics, such as knowing what an algorithm does, that are algorithmically impossible to identify, as stated that "... many questions *about* algorithms cannot be answered *by* algorithms ... in any sort of language that can be used for stating all computational procedures, there is no systematic way to tell whether a given set of instructions actually describes a procedure that is guaranteed to terminate and deliver an answer, no matter what input it is given" [Lewi81]. This is categorized mostly to be *cognitive* that requires human intelligence, analysis, and decision to be able to identify.

Of course, there are some gray areas between each of these categories [Booc94]. There are some categories that could fall in the mid-range, that is neither easy nor too difficult to identify; however, such categories can be encompassed within one of the categories defined previously. The contribution of this thesis will be mostly centered around the objective similar-characteristic category.

Classification

Similar-Characteristic classes are hierarchical in nature, that is, some of these characteristics are subsets of other higher class characteristic(s), each of which attacks a finer class. An example of this is a graphical engine that involves three levels of abstraction (hierarchies). At a high form, a user may request to draft a circle. Some of the advanced circle algorithms implement circle drafting as segments of lines-[Yong93b] or arcs-drawing [Yong93a]. Line drawing, at its lowest form, is just pixel identification and illumination [Fole90], [Hear86]. As can be implied, the highest class in this case is circle drafting, then lines drawing, and the lowest class is pixels identification and illumination. The choice of class of interest is dependent upon the user or arising need.

These hierarchical similar-characteristic classes exist in any general system, source code, or in particular legacy system. Identifying classes of similar-characteristics is a favored feature, that can dramatically help legacy system global understanding as will be demonstrated in Chapter 7.

Because of the interest of legacy systems and their problems, and because of the hierarchical nature of similar-characteristic classes of a system, in this context, a legacy systems has been defined to consist of characterized hierarchies that work together to accomplish a specific (pre-defined) objective. A legacy system (e.g. graphical system) could be viewed as a tree, whose nodes reflect specific characteristics. Any node (e.g. circle drafting) may have multiple number of children (algorithms) that inherit and share the node generic criteria. A child may have a distinct criterion (e.g. pixel illumination) that constitutes its identity. Siblings (e.g. circle-line and -pixel-identification algorithms) of a node share some criteria (e.g. producing a circle) identifying their parents (e.g. circle drafting).

A pre-defined-hierarchical scheme of the legacy system in this thesis is illustrated in a simplified tree shown in Figure 4.1 that describes the major components that contribute to this research.

This hierarchical scheme can be identified in mathematical notation (recursive logical ProLog-like definition) as the following:

L: $f_L(f_{People}, f_{Hardware}, f_{Software}, f_{Documentation}, f_{Data})$

S: $f_{Software}(f_{Dependent}, f_{Non\text{-}dependent}, f_{Scripts})$

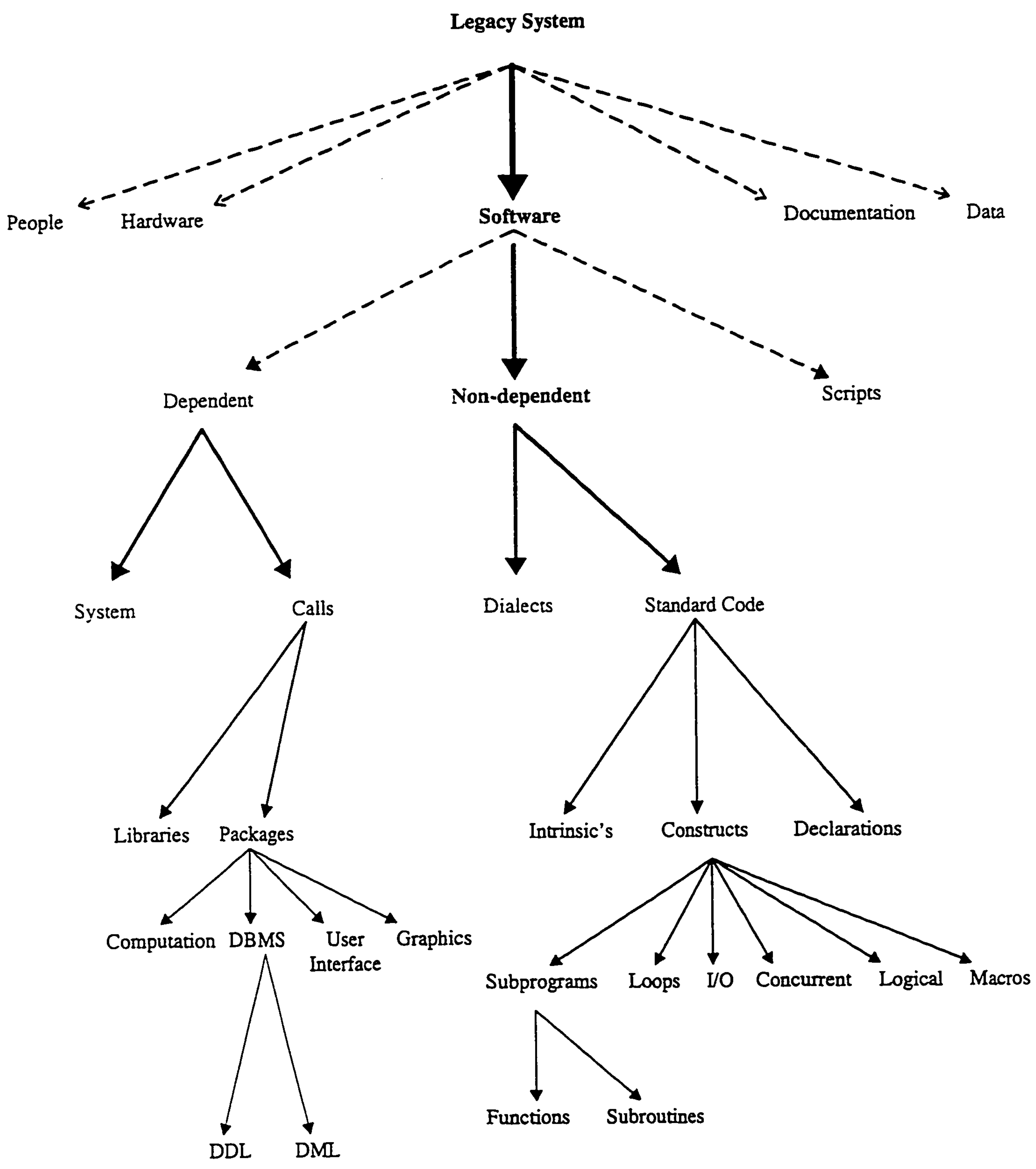
D: $f_{Dependent}(f_{System}, f_{Calls})$

N: $f_{Non\text{-}dependent}(f_{Dialects}, f_{StandardCode})$

C: $f_{Calls}(f_{Libraries}, f_{Packages})$

P: $f_{Packages}(f_{Computation}, f_{DBMS}, f_{UserInterface}, f_{Graphics})$

etc.


Legacy System
People
Hardware
Software
Documentation
Data
Dependent
Non-dependent
Scripts
System
Calls
Dialects
Standard Code
Libraries
Packages
Intrinsic's
Constructs
Declarations
Computation
DBMS
User Interface
Graphics
Subprograms
Loops
I/O
Concurrent
Logical
Macros
DDL
DML
Functions
Subroutines

Discussion

As could be seen in the previous figure, the legacy system is the root of the tree. Mainly, there are five children of the legacy system, which are *people*, *hardware*, *software*, *documentation*, and *data*. There is some mutual dependencies among siblings (e.g. between the software and documentation). Although hardware, documentation, people, and data could be further decomposed, they have been left at their consolidated state because they do not contribute as much to the framework of this thesis. Nevertheless, some reference to them may appear within the body of this thesis to illustrate certain points.

The *software* child (node) consists primarily of three children, that are *dependent code* (i.e. system or platform dependent code), *non-dependent code* (i.e. standard high level language), and *scripts* (i.e. shell scripts, make files, JCL's etc.). The non-dependent code falls within the strategic path of the framework.

The *dependent* code could be code that is dependent on the *system* or certain *calls*. These calls could be either *library(ies)* or *packages* calls.

Although there could be many packages, however, four main packages have been selected: *Computations* , *DBMS*, *User Interface*, and *Graphics*. These four categories could be system or platform dependent (e.g. Microsoft Windows), or stand-alone packages (e.g. Oracle RDBMS under many different platform running different operating systems).

A non-dependent code can be classified as *actual segments of code* (e.g. standard code), or *dialects* (e.g. FORTRAN dialects accepted in certain systems).

The non-dependent standard code could be decomposed into three major categories: *Intrinsic's*, *Constructs*, and *Declarations*. Most high level languages (e.g. C) require type and storage class declaration. Some languages (e.g. FORTRAN) come with rich intrinsic calls. High level languages code is usually composed of constructs. Constructs could be further decomposed into finer components, as seen in the figure. I/O has been selected to be an example of a construct child. In many cases, certain languages (e.g. FORTRAN) has I/O code embedded within the language (e.g. PRINT). Each of these children and many others can be further decomposed.

The tree in the previous figure may not reflect all the combinations and permutations of siblings or children; however, it has been chosen this way to illustrate the idea and define the framework of this thesis. If all options had been identified, the tree would have been complex, ramified, and hard to comprehend. Simplicity has been sought to illustrate the core idea.

In the previous tree, it is noticed that each node possesses certain criteria that are passed to its children. Each of the children contains its identifying criteria, and so on. Nodes could be dropped or added according to the nature of a given system.

The provided tree classifies the legacy system and defines relationships among its constituents. Converging a legacy system toward this tree will help organize, maintain, migrate the system, and reduce the amount of faults.

In the case of migration from one system/platform to another, for example, a critical path(s) of the tree should be identified. The path(s) should pass through those nodes that need modification, enhancement, or extraction in order to work under the new environment. Theoretically, paths can be highlighted and identified according to their identifying criteria. Segments of code that possess similar-characteristics should belong to a certain node, which may be liable for further decomposition.

Although code classification and breaking a legacy system into hierarchical tree can help source code understanding [Ball96], current source-code dictionary is required to further enhance functionality definition, maintenance, and to improve documentation. This concept can be sought throughout a third party commercial software.

Chapter 5

Similar-Characteristics-Code Clustering Tool

Part of the similar-characteristics code clustering tool has been implemented to help in clustering source code according to the methodology defined in the previous chapter, i.e. *objective*, *subjective*, and *cognitive*.

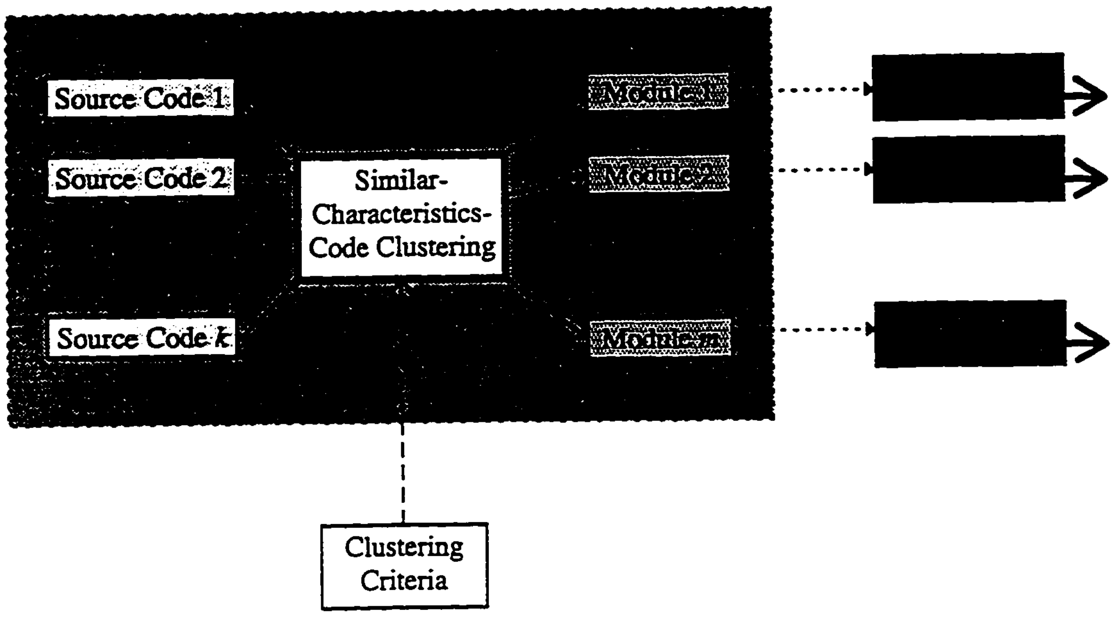


Figure 5.1: Similar-Characteristics Code Clustering tool flow

Tool Description

The previous figure shows the basic idea of a similar-characteristics code clustering tool (SCCCT):

$\Rightarrow$ SCCCT accepts more than one input file at a time.

$\Rightarrow$ SCCCT engine is a black box to the end user.

$\Rightarrow$ SCCCT produces a number of modules that are equivalent to the number of criteria fed by the user. The user should be able to define as many criteria as possible during one run of SCCCT, that was shown to be *m* in the previous example, which does not have to equal to *k*; *m* could be less, more, or equal to *k*.

$\Rightarrow$ Although the modules produced by SCCCT can be ready for use, it is recommended to run these modules into a coupling based clustering tool, which can attack finer detailed problems since SCCCT works on subprogram basis; that is to cluster code at subprogram level without modifying the contents of any of these subprograms. These subprograms could have been badly coded from the beginning.

⇒ SCCCT can support intrinsic clustering criteria, such as keywords, however the user should be provided the ability to define his criteria of interest in order to be able to attack more involved problems.

Tool Mechanism

Basically, the mechanism of the tool is depicted in Figure 5.2. The tool will require three categories of user input and will generate three classes of output.

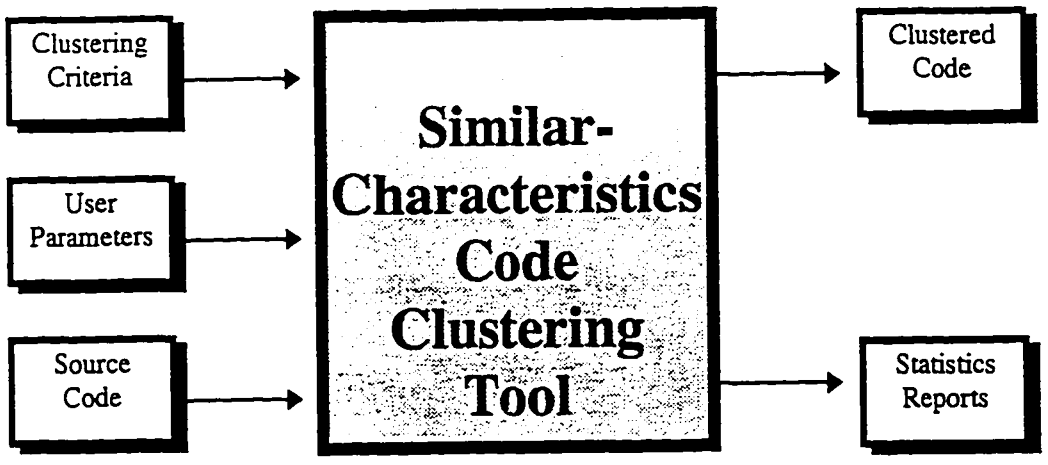


Figure 5.2: Similar-Characteristics Code Clustering Tool I/O

A. Required Input

1. Clustering Criteria

The clustering criteria role is to identify pieces of related code that need to be clustered together. There can be more than one clustering criterion at a single run of the tool. Each criterion can intersect with other criteria. Each criterion is represented by a cluster file. A criterion could be literal pattern matching or wild card expressions equivalent to the UNIX *egrep* filter. All cluster-file pointers of interest should be identified within a single file. Examples of clusters are: system calls, file manipulations, I/O calls, user interface calls, database connectivity routines, library calls, etc. A clustering criterion can be wide, narrow, or specific, such as clustering code that deals with a certain input file, for example. The user has to define the clustering criteria (classes) against certain objective using *opportunity strategy*.

2. User Parameters

The tool requires certain user parameter in order to generate desired output and deals with collisions among clusters. In the case of intersection among clusters, the user can prioritize clusters (that is to assign more gravity toward certain clusters where the intersected code will be attached to heavier gravity clusters and will not show up in the lower gravity clusters), allow intersecting code to go with more than one crossing clusters, or dynamically create distinct clusters to hold intersections.

3. Source Code

Source code is required to extract clusters. The tool can work with multiple source code files at the same time to enhance the possibility of collecting related code even if it is scattered amongst many distinct source code files. The source code file names should be collected in an input file as parameters.

B. Generated Output

1. Clustered Code

Clustered code is the result of filtering source code files fed by the user as well as user parameters. Normally, there will be a module per defined cluster, unless the default output is overridden by user parameter, e.g. generating dynamic clusters to handle intersection among different clusters.

2. Statistics Reports

Statistics report about user input parameters, number of clusters, intersecting clusters, and the percentage amount of intersection between clusters will be dispensed to the user to relate the whole work together. Sometimes, users want to try different parameters under different runs in order to accomplish the desired output. Having statistics reports will smooth this operation by relating output together besides the availability of other statistics that can help on achieving better judgment.

Tool Capabilities

The capabilities of the tool, whose general flow is depicted in Figure 5.2, are:

1. **Converting a legacy system to a hierarchical tree:**

   The tool is able to work with a scattered functionality and multi-file legacy system and converge it to a tree-like system, as depicted in Figure 4.1, with minimal human interaction.

2. **Intrinsic-automatic search for basic objective criteria:**

   The tool can automatically manipulate language keywords and objective characteristics discussed earlier.

3. **Supporting automatic smart-search for user-defined criteria:**

   The tool can automatically manipulate user-defined criteria. These user-defined criteria could range from too specific (e.g. pixel illumination), to very general (e.g. graphical library calls). Some criteria of interest could be user interface calls, database connectivity statements, or any other desirable criterion that mandates grouping, e.g. functions that contain certain variables[Gris95], occurrence of *goto* statement, etc.

4. **Hierarchical clusters (modules) generation:**

   The tool can automatically generate a hierarchy of interest. For example, if the user was interested to isolate the "Dependent" software "Calls", found in Figure 4.1, he can perform this operation through the usage of this tool while preserving the hierarchical characteristic.

5. **Background processes:**

   The tool allows concurrent execution with different parameters in order to reduce turn-around time and be able to work with huge source code file(s) as a background process since human intelligence fails to deal with or comprehend huge source code [Booc94]. Interactive tool may not help the user much when dealing with scattered-colossal-source code. A smart-background process can collect user ideas (criteria) that the user is looking for, then delivering them in a more manageable modular form.

6. **Reusable components extraction:**

   The tool can help its user to look for reusable components of interest written in a standard high-level language.

7. **Code understanding improvement:**

   Code understanding can be achieved through similar-characteristics code classification, as will be seen in Chapter 7.

8. **Modular output:**

   The tool can provide a modular output that can be more meaningful and easier for the use by Coupling Based Clustering Tools (CBCT), such as Cobol/SRE.

Chapter 6

Benefits of Methodology and Applied Tool

Similar-Characteristics-Code-Clustering methodology and tool present a number of benefits that can be used to reduce the severity of legacy system problems and enhance the flow and understanding of the proposed solutions, both of which are introduced in Chapter 2. Referring back to Chapter 2 and revising the annotated references can lead to more understanding of the following benefits:

1. **Source code understanding**

Source code understanding is highly desired and sought about in both industry and literature [Abd-96] because it is a key factor to solving some software problems, even though it is a complex cognitive task [Wood96].

Similar-characteristics code clustering (SCCC) is a concept used to hierarchically break a legacy system into manageable related pieces [Hsia96]. Besides its desired envisioned-structure, the hierarchical system produced aids source code understanding, which in turn, reduces the effects of legacy system deficiencies, and supports enhancing solutions proposed by literature and industry [Abd-96].

Moreover, the table produced of the tool can heuristically aid to calculating coupling and cohesion of clusters This result can be seen in Chapter 7, after applying the tool over four different cases.

2. **Clustering code of interest**

SCCC tool allows the clustering of code of interest, such as the user interface or others. This can help in many aspects of the deal, such as migration, wrapping, maintenance, and others.

3. **Enhance reusability**

   Reusability can be increased by clustering portable/reusable code together.

4. **Automation**

   Automation can speed up a certain project and reduce the amount of errors. SCCC tool provide a good clustering automation as will be seen in Chapter 7.

5. **Productivity increase**

   Productivity can be increased through automation, reusability, and code understanding.

6. **Heuristic-global calculations of coupling and cohesion**

   Chapter 7 will show how cohesion and coupling can be heuristically calculated in an easy manner and minimal parameters.

7. **Heuristic approach**

   Usage of the tool is approached heuristically. The user is not required to fully understand the system he is dealing with. Only minimal number of parameters are required to be input to the tool in order to achieve the previous mentioned benefits.

8. Ability to deal with colossal applications

Huge applications are difficult to understand and deal with. They are very difficult to deal with interactively. SCCC tool allows the user to input few parameters without having to vigorously deal with or understand the application before hand. Then, the process will be run in a batch mode.

Chapter 7

Case Study

Four software systems are used to demonstrate the benefits of the tool. Table 7.1 illustrates these systems properties which are:

LOC	Number of lines of code.
NF	Number of functions (subprograms)
SD	Starting date of implementation
MM	Man-Month of development
MP	Maintenance percentage after development.
ALF	Average lines per function.
MCPS	Main clusters percentages sum

#	*LOC*	*NF*	*SD*	*MM*	*MP*	*ALE*	*MCPS*
1	23,404	[illegible]	06/95	[illegible]	80%	[illegible]	175.65
2	5,587	[illegible]	03/96	[illegible]	1%	[illegible]	125.54
3	2,438	68	12/95	[illegible]	1%	[illegible]	122.06
4	39,453	244	N/A	N/A	N/A	151.67	158.20

Table 7.1

Required background

Background that is needed for analysis is given in this section. More background is given in Appendix D.

Coupling

Coupling is "The degree to which separate software components are tied together" [Budd97]. *Coupling* is protested because it hinders program understanding [Budd97], and it has different facets that can be seen in Appendix D.

Cohesion

Coupling describes the relationships between modules, and *cohesion* descries the relationships within them [Budd97]. *Cohesion* is "The degree to which components of a single software system (such as members of a single class) are tied together" [Budd97]. It comes in varieties (see Appendix D), one of which is the *coincidental cohesion*, that is when components of a module are grouped together for no evident reason, which usually suggests poor design [Budd97]. Nevertheless, some other types of *cohesion* are more desired (i.e. *data cohesion*).

Mission

The mission here is to cluster subprograms according to similar characteristics code clustering methodology using the SCCC tool provided with this thesis. The tool breaks the monolithic source file into smaller related components. After separating the applications, the tool will provide some statistics (i.e. clusters intersection ratio table) that can be used to analyze each system as a whole. This table should give enough information that describes the amount coincidental cohesion within each cluster as well as the amount of intersection between clusters. Moreover, the table can provides some coupling hints, as will be seen in the analysis part later.

The separation and clustering that will be implemented should announce how close this application to the three-tier-architecture: the interface, the database services, and the application [Brod95]. The clustering mission followed the concept of dividing the main applications into the following clusters:

A. X (GUI interface)

   1st. XLib (X library primitives)

   2nd. Xt (X Intrinsic's)

   3rd. Xm (X Motif)

   4th. Xint (a third-party X library specialized in graphics rendering)

B. db (database services)

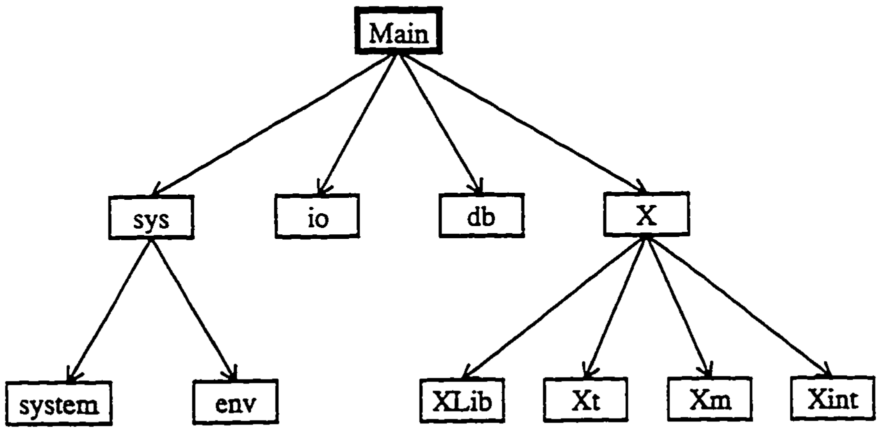
C. sys (system part of the application)

1st. system (actual system invocation using the "system" command)

2nd. env (environmental inquiry and update)

D. io (I/O part of the application)

The hierarchical nature of this decomposition is shown in the following tree:



It should be noted that some of the examined applications do not have all the components of the previous tree. This is why some of the leaves will be omitted from the statistics table.

Interpreting the table

The tool produces a square-table, that reflects the intersection among all clusters. The first column of the table "Main" shows the number of subprograms per cluster; for example, table 7.2 shows that there are 193 subprograms in the whole application, 61 of which are system related, 105 are I/O related, 22 are database related, and 151 are X (interface) related. Those subprograms are still further decomposed, such that 39 out of the 151 X subprograms are Xint related, and so on.

The first row of the table "Main" shows the percentage of each cluster over the main application. For example, the number of the database related subprograms of the whole application is 22 functions out of 193, that is11.40%.

The diagonal of the table represents the relationship of a cluster with itself; this is why all the diagonal cells should be 100%.

The rest of the table body represents the intersection relationship among all clusters. For example, it shows the percentages of intersection between a cluster and the rest of the other clusters. In table 7.2, the amount of intersection between the system calls and the database-services is 26.23%, which heuristically shows a degree of coincidental cohesion that could be subjectively calculated as 26.23%. This means that 16 subprograms out of 61 are common between the system and database-services clusters, which indicates that these subprograms posses *coincidental cohesion* [Mart85], [Scha90], [Your79], which usually implies poor design [Budd97].

One more interesting feature of the table is the sum of percentages of the main clusters (i.e. sys, io, db, and X) can be used as another metric to give a clue about the total *coincidental cohesion* of the whole application. In this case, the sum (31.61+54.40+11.40+78.24) is equal to 175.65. The closer this number to a hundred, the more ideal the application is.

I. Case 1

Functionality

This program was initiated to solve a problem of numbering and sequencing batch jobs over a number-crunching machine.

Method of development

This case represents ad hoc design and implementation. The requirement was never stated at starting time. Ad hoc and late requirements continued to be thrown into the project (i.e. the project started with a hierarchical DBMS as a corporate database, then ended up with a relational DBMS, and ,unfortunately, eventually both DBMS's had got to be supported).

Platforms

This applications runs under one UNIX platform for user interaction, although it involves two other different-heterogeneous platforms (i.e. mainframe and number crunching machines) due to scattered data, and two heterogeneous-distributed-corporate database management systems (one of which is hierarchical and the other is relational).

Human resources

Initially this system was assigned to a neophyte, who lacks the awareness of both the working environment and the problem domain. People who implemented the system kept changing, from one novice to another.

Maintenance

The final product was an evident spaghetti case of undocumented, unstructured, and non-maintainable piece of software. Whence, 80% of its administrator time was spent over maintenance.

Clustering

After feeding all the user parameters and the source file to the SCCC tool, ten clusters of similar-characteristics were generated. These clusters can be shown in the intersection-ratio table below:

[illegible]	Main	sys	system	env	io	db	X	XLib	Xt	Xm	Xint
Main	193	31.61%	13.99%	26.42%	54.40%	11.40%	78.24%	56.99%	73.58%	56.99%	20.21%
sys	61	100.00%	44.26%	81.97%	90.16%	26.23%	83.61%	52.46%	78.69%	54.10%	32.79%
system	27	100.00%	100.00%	62.96%	92.59%	29.63%	74.07%	40.74%	62.96%	40.74%	14.81%
env	51	100.00%	35.29%	100.00%	88.24%	31.37%	96.08%	62.75%	90.20%	60.78%	37.25%
io	105	52.38%	23.81%	42.86%	100.00%	18.10%	79.05%	54.29%	75.24%	60.00%	24.76%
db	22	72.73%	36.36%	72.73%	86.36%	100.00%	100.00%	77.27%	90.91%	72.73%	31.82%
X	151	33.77%	13.25%	31.79%	54.97%	14.57%	100.00%	72.85%	93.38%	72.85%	25.83%
XLib	110	29.09%	10.00%	28.18%	51.82%	15.45%	100.00%	100.00%	93.64%	81.82%	29.09%
Xt	142	34.51%	12.68%	32.39%	56.34%	14.79%	100.00%	73.24%	100.00%	72.54%	27.46%
Xm	110	30.00%	10.00%	28.18%	57.27%	14.55%	100.00%	81.82%	92.73%	100.00%	32.73%
Xint	39	51.28%	10.26%	48.72%	66.67%	17.95%	100.00%	82.05%	97.44%	92.31%	100.00%

Table 7.2

Analysis

From the previous two tables and the environment of implementation, the analysis of this case shows that:

A. The program is interface dominant because the X cluster makes up 78.24% out of the whole application.

B. Weak cohesion due to high *coincidental cohesion.*

C. The amount of coupling among the application clusters can be inferred to be high due to functionality mixing, coincidental cohesion, and high amount of data that needs to be moved around clusters, especially the database services cluster. The amount of intersection between the other clusters and X (the interface) are:

   1. 083.61% of sys,
   2. 079.05% of io,
   3. 100.00% of db.

D. The previous point indicates that this system is faraway from the three-tier-architecture.

E. Functionality mix-up due to low-cohesion and maybe high-coupling.

F. The subprogram size is quite large, i.e. 121.26 lines per function on the average. This is an effecting factor on the amount of coupling, cohesion, and complexity.

G. The application is difficult to maintain due to coincidental cohesion. complexity, and large function size.

H. The application is difficult to migrate because this type of application is non-decomposable [Brod95] due to the factors previously mentioned.

I. The amount of I/O of this application is quite high; that is 54.40%.

J. Automation of reverse engineering tools over this application is not possible without heavy human interaction.

These points consolidate this case description of being ad hoc design and implementation and other points mentioned earlier.

Recommendations

After using SCCC tool and methodology, one or more of the following recommendations are made:

1. Complete system rewrite.
2. System migration [Brod95] toward three-tier-architecture.
3. Employing some other reverse engineering tools, such as Cobol/SRE [Ning4], to fix up the subprogram-internal-mess, then, re-implementing SCCC tool.
4. Reduce each subprogram size to fifty lines instead of 121.26.

II. Case 2

Functionality

Provide a query-by-example under a graphical user interface (GUI) for end users. The application collects user parameters, retrieves the data, and then draft the data on the screen and/or generate output flat files to be fed to other applications.

Method of development

This case represents a more systematic way of design and implementation. The requirements were, almost, fully stated at the beginning. The actual implementation started on 03/96 and lasted for three months. The application should utilize the available corporate relational DBMS.

Platforms

The application runs under one UNIX-platform.

Human resources

The application has been designed and implemented by a relatively experienced programmer who is more aware of the working environment and the problem domain.

Maintenance

After delivery, this application did not require much maintenance except for some functionality addition.

Clustering

The clustering of this application is similar to Case 1, except the system part did not need to be furthered decomposed because this is relatively smaller application.

CLUSTER	Main	system	io	db	X	XLib	Xt	Xm	Xint
Main	94	4.26%	44.68%	2.13%	74.47%	56.38%	55.32%	51.06%	24.47%
system	4	100.00%	100.00%	25.00%	75.00%	50.00%	75.00%	50.00%	25.00%
io	42	9.52%	100.00%	4.76%	80.95%	61.90%	73.81%	69.05%	21.43%
db	2	50.00%	100.00%	100.00%	100.00%	0.00%	50.00%	50.00%	0.00%
X	70	4.29%	48.57%	2.86%	100.00%	75.71%	74.29%	68.57%	32.86%
XLib	53	3.77%	49.06%	0.00%	100.00%	100.00%	75.47%	67.92%	39.62%
Xt	52	5.77%	59.62%	1.92%	100.00%	76.92%	100.00%	82.69%	25.00%
Xm	48	4.17%	60.42%	2.08%	100.00%	75.00%	89.58%	100.00%	25.00%
Xint	23	4.35%	39.13%	0.00%	100.00%	91.30%	56.52%	52.17%	100.00%

Table 7.3

Analysis

1. The application is interface dominant (74.47%).
2. Relatively high I/O usage (44.68%), which can be tolerated because this application dumps many flat files into different formats.
3. The application is well designed to concentrate the database services into two functions only.

4. However, these two functions are highly intersected (100%) with X interface.
5. Little interface with the system (4.26%).
6. The total sum of the main clusters percentages is 125.54, which makes it closer to an ideal system.
7. The system was designed without decomposition intention. This can be seen from the high intersection found between all other clusters with the main X cluster.
8. Although, X cluster is dominant, there is still high degree of coincidental cohesion, and functionality mix-up exists, therefore.
9. Except for the I/O part, the application is relatively easy to maintain and/or migrate.
10. Automation of reverse engineering tools will be impeded by the strong intersection of X and the I/O and interface.
11. The complexity and intersection of this application could be aggravated if the size of other clusters grows.

Recommendations

1. Due to its small size, manually strip out the database services cluster from the X interface.
2. Due to its small size as well, manually strip out the system code from the X interface.

3. Implement a reverse engineering tool, such as Cobol/SRE, to separate I/O from the interface. This will be the most time consuming.
4. Maintain the system toward a three-tier-architecture.

III. Case 3

Functionality

This case attacks the problem of archiving and retrieving end user projects' data.

Method of development

The problem domain and the environment are completely clear. A quite good analyst worked on this problem. It took him around 1.5 months to finish implementation. The average function length was around 35.85 lines. The final application consisted of 2,438 lines of code and 68 subprograms. The analyst was pressurized to finish on time. There were not many options left, but to finish the product as soon as possible.

Platforms

One UNIX platform.

Clustering

CLUSTER	Main	sys	system	env	IO	X	Xt	Xm
Main	68	16.18%	4.41%	2.94%	26.47%	79.41%	76.47%	51.47%
sys	11	100.00%	27.27%	18.18%	81.82%	72.73%	72.73%	54.55%
system	3	100.00%	100.00%	0.00%	33.33%	100.00%	100.00%	33.33%
env	2	100.00%	0.00%	100.00%	100.00%	0.00%	0.00%	0.00%
IO	18	50.00%	5.56%	11.11%	100.00%	61.11%	61.11%	55.56%
X	54	14.81%	5.56%	0.00%	20.37%	100.00%	96.30%	64.81%
Xt	52	15.38%	5.77%	0.00%	21.15%	100.00%	100.00%	65.38%
Xm	35	17.14%	2.86%	0.00%	28.57%	100.00%	97.14%	100.00%

Table 7.4

Analysis

1. The application is X (interface) dominant (79.47%).
2. The sum of percentages of the main clusters is 122.06, which is the closest to the ideal case from the other cases presented.
3. This application has the lowest coincidental cohesion among the examined cases. Moreover, this application maintains relatively higher cohesion, which could be due to smaller sizes of subprograms, as the average function length is 35.85 lines.
4. The I/O part is relatively low compared to the other cases.
5. There are no databases involved, which reduces the severity of the problem.
6. A lot of 0% *coincidental cohesion's* are found in this table, even between siblings (as shown in the system and *env* whose amount of *coincidental cohesion* is 0). 0% of *coincidental cohesion* is highly desired and it indicates very good cohesion. The *env* cluster is a good indicator of this where it has two totally independent functions.
7. Relatively, a smaller-number *coincidental cohesion's* are dominant among clusters.
8. For its size, this application seems to be relatively good. Although, the original analyst had no clue regarding either coupling or cohesion, he was inclined toward good coding practices.

Recommendations

1. With minimal efforts, this application can be modified to comply with a three-tier-architecture.
2. Reduce the system and I/O *coincidental cohesion.*
3. Reduce *coincidental cohesion* associated with X interface.

IV. Case 4

Functionality

Simulation

Method of development

This was more of a migration process from an older system running on the mainframe. Moreover, rigorous planning and implementation efforts have been put into place due to the importance and sensitivity of the application.

Platforms

A number crunching machine for algorithm processing plus an end UNIX workstation for graphical display.

Human resources

Three people with variant experience.

Maintenance

Could not attain maintenance information.

Clustering

CLUSTER	Main	sys	system	env	io	X	XLib	Xt	Xm
Main	244	3.28%	0.82%	3.28%	74.18%	80.74%	6.97%	61.48%	56.97%
sys	8	100.00%	25.00%	100.00%	87.50%	100.00%	12.50%	100.00%	87.50%
system	2	100.00%	100.00%	100.00%	100.00%	100.00%	50.00%	100.00%	100.00%
env	8	100.00%	25.00%	100.00%	87.50%	100.00%	12.50%	100.00%	87.50%
io	181	3.87%	1.10%	3.87%	100.00%	77.90%	9.94%	65.75%	53.04%
X	197	4.57%	1.52%	4.57%	71.57%	100.00%	9.14%	76.65%	71.07%
XLib	17	5.88%	5.88%	5.88%	100.00%	100.00%	100.00%	70.59%	35.29%
Xt	150	5.33%	1.33%	5.33%	79.33%	100.00%	8.00%	100.00%	70.67%
Xm	139	5.04%	1.44%	5.04%	69.06%	100.00%	4.32%	76.26%	100.00%

Table 7.5

Analysis

1. A positive attribute of the program is that it has already been segmented into many modules (source files) to accommodate different functionality. Also, the declaration part has been separated from the actual source into include files. These two attributes, enhance reusability and maintenance. Besides, the initial task can be distributed easily among a software team. This also indicates that the implementation of this application was more systematic than ad hoc.
2. The program size is quite large for individual to work on. Having a team to work on this application was a good decision.
3. Although a good plan was in place, good software design practices were not observed. The average subprogram size was quite large, that is 153.67 lines per function, and the amount of coincidental cohesion was relatively high as can be

seen from the table. This indicates that the team involved had long programming experience, but lacked the software engineering theory. This was a successful application because the team involved knew the problem domain quite well and the final program size is still classified to be of small scale. If the size of this program exceeds 100,000 lines of code, most probably, it will reach a bad stage of complexity, that might lead to software apoplexy as defined in [Brod95].

4. The software understanding of this application tend to be difficult because the number of subprograms exceeds the number of lines per function [Till95].
5. Although this application is dominated by the user interface (i.e. 80.74%), it has fairly large I/O percentage (i.e. 74.18%) that competes with the interface part.
6. Unfortunately, as can be seen from the table, all other cluster types are being absorbed by the user interface part, except for the I/O cluster whose intersection rate with the GUI cluster is 77.90%, which is still high by all measures.
7. Due to its size and problems, this case is similar to Case 1, which indicates similar analysis, that is many of Case 1 analysis points are applicable with this case.

Recommendations

1. A complete reengineering of the system is required. This may involve other reverse engineering tools, such as Cobol/SRE, in order to clear the internal subroutine mess.
2. Since the data structures (i.e. the declaration part) is already separate from the actual code, a better chance of paradigm shift toward object-oriented exists. It is highly recommended to adapt object-oriented technology due to the complexity and sensitivity of the application.
3. Whichever path is chosen, the coincidental cohesion needs to be minimized. Also, the subprogram size needs to be shrunk to one third (i.e. 50 lines per function) of whatever size being practiced (i.e. 153.67 lines per function).

Chapter 8

Conclusion

It is understood that software systems, especially legacy ones, are complex and difficult to deal with. *Software engineering* is used to organize the software life cycle and minimize certain development and maintenance pitfalls. Software engineering proposed some solutions to deal with the software dilemma, however, they were not enough to deal with certain circumstances.

Software reverse engineering (SRE) has recently surfaced as a software engineering branch (i.e. solution) that tries to enhance legacy systems and lengthen their lives. SRE has been used to recover some consequences, and sometimes is used as the only solution for certain catastrophes [Ingl94]. SRE also surfaced to improve software reusability and other software engineering tool.

Code clustering is a special class of SRE that deals with software code. It has been seen that code clustering is a very effective tool to organize code according to certain criteria, such as strong coupling discussed earlier in Chapter 3.

Similar-Characteristics code clustering (SCCC) is a special type of code clustering that was proposed by [Raz 93]. A framework of SCCC has been identified and characterized in this thesis, as was illustrated in Chapter 4.

Theoretically, it has been noted that SCCC can support solutions presented by software engineering besides reducing the factors that provoke software problems.

In order to put theory into practice, a tool was developed to demonstrate SCCC. This tool was seen to promote nice features and advantages as discussed in Chapters 5 and 6.

Four case studies have been analyzed using this tool (as was seen in Chapter 7) where a good demonstration of the tool usage was established. The tool was able to accumulate related clusters, and heuristically compute the amount of *coincidental cohesion*. The tool usage guides to global understanding of the examined application, which leads to better code understanding . The SCCC methodology was used as a metric for calculating the *coincidental cohesion* as was seen the in Chapter 7. The result of the tool eased the global analysis of a particular application, which induced certain recommendations and remedies to the application problems.

The SCCC methodology and tool contributed to legacy system understanding that can alleviate problems and enhance solutions.

SCCC can be used with other code clustering methodologies for effective solution not only when dealing with legacy systems but also while founding a new software system project.

APPENDICIES

A. Glossary

Code Clustering Code clustering is a reverse engineering methodology used to bundle together related (usually scattered) code segments according to certain criteria.

Data warehousing Data warehousing is the process of sterilizing data, stripping it of application context, and then isolating this data behind one or more relational database engine [Ecke95a], [Ecke95b], [Hard95].

Functionality Separation Functionality separation is identifying a functionally complete piece of logic that is functionally related yet de-localized throughout old code [Ning94].

GUI GUI stands for Graphical User Interface that is currently used with graphical window managers such as X-Motif. Currently, GUI's are the most popular way of interfacing with end users.

Legacy system Legacy system is a large and complex software system that is neither feasible to abandon nor practical to maintain.

Maintenance Maintenance is the effort and cost exerted in order to keep a software system in-shape and up-to-date.

Recycling Recycling is the collection of reusable components without having reengineering intentions.

Redevelopment Redevelopment is the act of rewriting a new system from scratch without taking into account a preexisting system.

Re-Engineering (Reengineering) Re-engineering (Reengineering) is applied over a system to enhance its maintainability without adding or extracting functionality.

Reverse Engineering (RE) Reverse engineering is essentially the development of the technical data necessary for the support of an existing production item developed in retrospect as applied to hardware systems [Ingl94].

Similar-Characteristics Code Clustering (SCCC) SCCC is a special type of code clustering that promotes grouping subprograms the posses similar characteristics, such as user interface related code.

Software Engineering Software engineering is a set of rituals used for software development and maintenance.

Software Migration Software migration is the process of migrating a software system (usually legacy system) from one environment (e.g. mainframe) to another (e.g. client-server). Usually, the migration process is accompanied with difficulties when trying to run the old system under a new environment.

Software Reverse Engineering (SRE) Software reverse engineering concerns the extraction (recovery) of higher-level design or specification information from the computer software [Ingl94].

Wrapping Wrapping is the process of building GUI's around a legacy system old user interfaces.

B. Reverse Engineering

The origin of reverse engineering is unknown and it is not necessarily new [Ingl94]. Nevertheless, software-wise, the first implicit work in software reverse engineering was done by Boehm and Jacopini in 1966. Their work regarded eliminating GOTO's from code. Dijkstra put more significance into it in 1968, when he wrote "Go To Statement Considered Harmful" [Arno94]. Interest continued to grow rapidly as more profound problems than GOTO's surfaced.

Reverse engineering is a broad subject whose definition changes according to the working environment. Therefore, many definitions exist for the term *reverse engineering*. One definition of reverse engineering is that "Reverse engineering is a set of methods and tools that may enable a software designer to create new and better programs" [Samu90].

Software reverse engineering includes the following subjects:

1. Code clustering and reusability
2. Object code reverse engineering
3. Reengineering databases to support data fusion

4. Reverse engineering within the life cycle of a project
5. Reverse engineering to hunt for errors or flaws, or in debugging
6. Reverse engineering source code from one language to another

In conclusion of reverse engineering definition, it was stated that reverse engineering is a technique beneficial to solving system-specific requirements that cannot be achieved by traditional means. The main goal of reverse engineering is to increase productivity through improved documentation. Reverse engineering requires highly professional practitioners. When applied with caution and thorough planning, reverse engineering can be very effective and profitable [Ingl94].

Current Status of Software Reverse Engineering

In the early 1990's, interest shifted to maintaining code rather than developing new systems. This is considered a blossoming area for reverse engineering [Arno94].

Part of reverse engineering is evolving to utilize old code, that can be implemented through code clustering. Because of the importance of reverse engineering and code clustering, some automation processes have been written to accommodate certain needs, such as platform or operating system migration of a software system. Most of these packages were written to accommodate the reengineering of some particular application. All practitioners who were involved claimed the benefit and time reduction of using reverse engineering methods and tools for achieving their goals. These automation packages may not be general enough; however, they serve their intended purposes and have proven advantages in organizing and utilizing time more effectively. Two of these packages and some pervious work are explained in Tools of Code Clustering chapter.

Rationale for Software Reverse Engineering

There are many appealing reasons for reverse engineering. These reasons include:

1. Cost of developing new software applications

   It is neither easy nor cheap to develop a new software application. It is not practical to maintain old, out of date software, either [Mack95], [Prem94], [Tile95]. Reverse engineering has to be implemented over legacy systems to be able to take advantage of earlier development, reduce the cost, and improve software quality [Mack95], [Prem94].

2. The nature of software and hardware

   Software is not catching up with hardware advancements. Usually organizations upgrade their hardware regularly in order to keep up with technology; however, the software upgrade is much less frequent because software is more difficult and costly to modify. With time, software becomes old and obsolete. In many cases, the original hardware that drove the initial software design does not exist anymore so that people are faced with different technology. In these cases, reverse engineering is needed to upgrade the old system to take advantage of the new hardware [Mack95].

There are two attributes of old code that make it arduous to comprehend and reuse [Ning94]. The first attribute is that most of the code is machine dependent [Mack95], which is useless across different platforms. The second is that older programs are not structured [Ning94] and lacks software engineering disciplines. The benefit of reverse engineering in this case is to get the reusable components as well as organizing the code for more flexibility and reduced expenses [Mack95], [Ning94].

Another nature of software is that almost 80% of it is not fully utilized [Tile95]. Reverse engineering could be employed to extract the remaining 20% out of the original code and try to enhance it and gear it toward perfection [Mak95].

3. Enhancing maintainability of software systems

Maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [Bank93]. Besides the reasons mentioned earlier, re-marketing could be another driver of software reengineering [Samu90].

Maintaining old unstructured software is not cost effective and causes chaos [Mack95], [Prem94]. Enhancement may include reusability, modularity, maintainability, etc. Also, Chikofsky supports this enhanced maintainability of software systems by defining reverse engineering as a way to better comprehend the system, which leads to more appropriate modifications applied over the system of interest [Chik90]. Even if the software is well written, grouping (clustering) processes will allow better control [Raz 93].

Software reverse engineering should be implemented over complex and pivotal applications in order to enhance maintainability as well as reducing the complexity [Mack95].

In its general sense, reverse engineering can be implemented over a system to discover some of the project pitfalls that were overlooked during the software design [Ingl94], which in turn helps reduce the cost of maintenance.

4. Enhancing reusability of system components

Reusability is the ability of software products to be reused, in whole or in part, for new applications [Meye88]. Reusability is a driving force behind reverse engineering a system [Prem94]. As it is almost impossible to run software in a totally different heterogeneous environment without involving emulators, which are impractical in many cases, breaking down the system to make use of its operational functions is indispensable. For example, developers cannot take a mainframe-based system and migrate it to a client/server-based system without fishing for reusable constituents by implementing RCR (i.e. reusable component recovery) concepts. RCR should produce components that have a higher reusability rate than the system as a whole [Ning94].

5. New developments

New developments and new requirements will exist as long as computers and users exist [Phil96], [Tile95]. In the contrary, the software productivity is not catching up with current demands [Mili95]. A very effective way of taking advantage of an old system is by reverse engineering the old system and utilizing its useful features and reusable components [Mack95], [Samu90].

Collecting already existing pieces that are sound and do not need testing will not only effectuate the life cycle of software [Benn95], but will also improve the total quality of the software as well as cutting down expenses [Haag96].

6. Inter-program interface

In practice, there are many cases where a different program interface is required than the one provided. One of the ways to know about a third party system is to reverse engineer it. Another reason to reverse-engineer software is to discover how the program's interface is constructed so developers can develop programs that will be compatible with that program [Samu90].

7. Disclosing the idiosyncrasies of a system

Disclosing the idiosyncrasies of a system is necessary sometimes for peaceful (i.e. education) or violent (i.e. war) directions. Apart from modifying and developing a new program, another category of things one might do with the results of reverse engineering would be to disclose those results to other people, for example, by publishing an article about them [Samu90].

Sometimes, the essential data of a project or a system could be lost for one reason or another (i.e. the destruction of some vital data for Kuwait during Gulf War in 1990). Implementing reverse engineering is a must in such a case to reduce damage [Ingl94].

8. The growth of software systems

The size of a system could grow very large. Dividing a large system will allow it to be better controlled. Many large companies are facing a problem: their legacy systems are inhibiting their business growth and capacity to change [Ning94]. The most effective way of dividing large systems is by reverse engineering them for all the reasons mentioned earlier.

C. *Reusability*

a) Definition

Reusability[3] is the ability of software products to be reused, in whole or in part, for new applications [Meye88], [Tull89]. Reusability is the creation and consumption of reusable elements [Fafc94]. Reusability has been given special attention in the past, present, and most probably in the future. Reusability was a driving force behind the development of object-oriented paradigm. Reusability is driven by quality, productivity, and economical reasons [Budd91, Frak96, Henr95, Isak96, Lim 94, Mili95, Pfle96], and as stated that "... reuse technology has the greatest potential to reduce the cost of software" [Card94]. Also, reusability can dramatically affect efficiency, as stated that "Reuse is a tool that can help organizations provide speedy and effective solutions ..." [O'Co94]. Reusability is important to the extend of writing new projects with reusability intentions. Even governments and companies have been involved to support reusability [Isak96].

[3] It is important to note that reusability is not limited only to software components [Roge95]; however, in this context, reusability is mainly concerning software components.

b) Advantages

In this context, reusability, that is analogous to software recycling, is needed to alleviate the effect of the obstacles mentioned previously. Reusability improves *maintenance* by injecting already tested and proven functions into a software body [Lim 94]. Reusability helps with *redevelopment* when the right components are there to reuse. Reusability has been a driving force behind *reverse engineering* [Prem94]. Whenever *migrating*, the more that can be reused, the more successful the migration project will be. It will be more successful in terms of economics, productivity, and quality. *Wrapping* is already reusing existing components of a legacy system. Wrapping cannot achieve its goals unless it reuses whatever is available. *Integrating* a system under a new environment is a process of reusing this system in whole. *New paradigm*, i.e. Object-Oriented [Faya96], has surfaced because it supports reusability. *Data warehousing* is reusing already existing data.

c) Disadvantages

Reusability has to be treated with care in order to get advantage of it, not the opposite [Budd91], [Frak94], [Isak96], [Pfle96]. Creating reusable code is not an easy task and requires astute professionals to accomplish the intended goal safely. Furthermore, sometimes, extravagant reusability intentions (i.e. object-oriented multiple inheritance), cause a lot of overhead, sophistication and complexity [Budd91].

D. *Coupling and Cohesion*

Coupling is a measuring attribute of the interaction level between/among a software system segments. Also, it is stated that *coupling* is the degree of **external** interactions between/among modules of a given software system [Your79]. *Cohesion* is another attribute that goes along with coupling, and it is defined to be the level of **internal** interactions within a specific module [Your79]. *Coupling* is a detested characteristic, whereas, *cohesion* is a desired virtue [Offu93], [Your79].

Coupling is classified to be:

1. Content coupling

   Content coupling is the reference (modification) of local data of a module by another module.

2. Data coupling

   Data coupling is passing homogenous data from one module to another that operates upon the passed data, which can be a simple element or an array whose all elements will be used by the referenced module.

3. Stamp coupling

Stamp coupling is the concept of passing a compound data structure (e.g. Pascal record) from one module to another that will operate only upon some elements of the passed structure.

4. Common coupling

Common coupling, as taken from FORTRAN COMMON block, allows different modules to operate upon global (external) data.

5. Control coupling

Control coupling is controlling the logic of a certain module throughout passing specific flags from a different module, e.g. fopen("fileName", "r"), which means open a file for read by passing "r" as the second argument. The way a file is accessed can be changed by passing "w" instead in order to grant writing capabilities to the opened file.

Cohesion is ranked, from lowest to highest preferred [Mart85, Scha90, Your79], to be:

1. Coincidental cohesion

   Coincidental cohesion is arbitrary execution of unrelated statements, that is mixing I/O with computational statements, for example.

2. Logical cohesion

   Logical cohesion is performing series of related functions by invoking other modules to do some desired task while performing some local statement. In other words, it is dividing the load between the active module and an external module to complete a certain task.

3. Temporal cohesion

   Temporal cohesion is the execution of sequential statements that are related in time, such as calculating the sum of one array and then calculating the sum for another array.

4. Procedural cohesion

Procedural cohesion is mostly procedures invoking (calling) to acquire data, then manipulating the data locally, then sending the calculated result through other procedures. This can be seen in dealing with databases where calling a routine to acquire certain record data is performed, then locally modifying some data attributes, then invoking another procedure(s) to store the result into the database.

5. Communication cohesion

Communication cohesion is the sequence of procedure calls to perform a desired task by communicating data from a source to destination, such as printing employees cheques, using a printer after querying the employees data from a corporate database.

6. Information cohesion

Information cohesion is the idea of blocking procedures (functions) to operate upon related data structure(s). This can be seen in Pascal procedure blocking mechanism when dealing with a stack or queue data structure. Also, this can be seen in a *class* defined in an object-oriented language such as C++.

7. Functional cohesion

Functional cohesion is performing a single simple task by a function, such as returning the cosine result of a value (e.g. cos(0)), or drafting a graphical line on the screen (e.g. line([0,0],[1,1]).

References

[Abd-96] Salwa K. Abd-El-Hafiz and Victor R. Basili, "A Knowledge-Based Approach to the Analysis of Loops", *IEEE TRANSACTION OF SOFTWARE ENGINEERING*, Vol. 22, No. 125 pp. 339-360, May 1996.

[Abde96] Tarek K. Abdel-Hamid, "The Slippery Path to Productivity Improvement", *IEEE Software*, Vol. 13, No. 4, pp. 43-52, July 1996.

[Acke89] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski, "Software Inspections: An Effective Verification Process", *IEEE Software*, pp. 31-36 , May 1989.

[Adhi96] Richard Adhikari, "Migrating legacy data: how to get there from here", *Software Magazine*, Vol. 16, No. 1, pp. 75-79, January 1996.

[Adol96] W. Stephen Adolph, "Cash Cow in the Tar Pit: Reengineering a Legacy System", *IEEE Software*, pp. 41-47, May 1996.

[Aike94] Peter Aiken et al, "DoD Legacy Systems Reverse Engineering Data Requirements", *COMMUNICATIONS OF THE ACM*, Vol. 37, No. 5, pp. 26-41, May 1994.

[Arno94] Robert S. Arnold, "Viewpoints", *COMMUNICATIONS OF THE ACM*, Vol. 37, No. 5, pp. 13-14, May 1994.

[Ball96] Thomas Ball, and Stephen G. Eick, "Software Visualization in the Large", *COMPUTER*, pp. 33-42, April 1996.

[Bank93] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig, "SOFTWARE COMPLEXITY AND MAINTENANCE COSTS", *COMMUNICATIONS OF THE ACM*, Vol. 36, No. 11, pp. 81-94, November 1993.

[Bart94] John J. Barton, *Scientific and Engineering C++*, ADDISON WESLEY PUBLISHING COMPANY, INC., U.S.A, 1994.

[Baum95] David Baum, "New life for legacy applications", *Data Based Advisor*, Vol. 13, No. 6, pp. 85-90, July 1995.

[Baum96] David Baum, "Legacy systems live on; but updating host apps to distributed computing can be a huge undertaking", *Information Week*, No. 572, pp. 10A-12A, March 25, 1996.

[Benn95] Keith Bennett, "Legacy Systems: COPING WITH SUCCESS", *IEEE Software*, Vol. 12, No. 1, pp. 19-23, January 1995.

[Boeh88] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", *COMPUTER*, pp. 61-72, May 1988.

[Boeh96] Barry Boehm, "Anchoring the Software Process", *IEEE Software*, Vol. 13, No. 4, pp. 73-82, July 1996.

[Booc94] Grady Booch, *OBJECT-ORIENTED ANALYSIS AND DESIGN*, second edition, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, U.S.A, 1994.

[Brod95] Michael L. Brodie and Michael Stonebraker, *MIGRATING LEGACY SYSTEMS*, Morgan Kaufmann Publishers, Inc., San Francisco, California, U.S.A, 1995.

[Broo87] Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accident of Software Engineering", *COMPUTER*, pp. 10-19, April 1987.

[Broo95] Frederick P. Brooks, Jr., *THE MYTHICAL MAN-MONTH*. ADDISON-WESLEY PUBLISHING COMPANY, USA, 1995.

[Budd91] Timothy Budd, *AN INTRODUCTION TO Object-Oriented Programming*, Addison-Wesley Publishing Company, Inc., U.S.A, 1991.

[Budd97] Timothy Budd, *An Introduction to Object-Oriented Programming*, Second Edition, Addison-Wesley Longman, Inc., U.S.A, 1997.

[Cand96] James W. Candler, Prashant C. Palvia, Jane D. Thompson, and Steven M. Zeltmann, "The ORIOEN Project: Stages Business Process Reengineering at FedEx", *COMMUMINCATON OF THE ACM*, Vol. 39, No. 2, pp. 99-107, February 1996.

[Canf92] Gerardo Canfora, Aniello Cimitile, and Ugo de Carlini, "A Logic-Based Approach to Reverse Engineering Tools Production", *IEEE TRANSACTION OF SOFTWARE ENGINEERING*, Vol. 18, No. 12, pp. 1053-1064, December 1992.

[Card94] Dave Card, and Ed Comer, "WHY DO SO MANY REUSE PROGRAMS FAIL?", *IEEE SOFTWARE*, pp. 114-115, September 1994.

[Char95] Michael Charter, "The love/hate legacy debate", *Computing Canada*, Vol. 21, No. 22, p. 47, October, 25, 1995.

[Chen90] YIH-FARN CHEN et al, "The C Information Abstraction System", *IEEE TRANSACTION ON SOFTWARE ENGINEERING*, Vol. 16, No. 3, pp. 325-334, March 1990.

[Chik90] Elliot J. Chikofsky, and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, pp. 13-17, January 1990.

[Cimi96] Daniela Cimino, "Grappling with client/server 'gotchas:' heterogeneity, legacy integration still daunting", *Software Magazine*, Vol. 16, No. 1, p. 32, January 1996.

[Clin95] Marshall P. Cline, and Greg A. Lomow, *C++ FAQs*, ADDISON WESLEY PUBLISHING COMPANY, INC., U.S.A, 1995.

[Cons95] Paul Constance, "DOD wants to stop maintaining legacy maintenance systems", *Government Computer News*, Vol. 14, No. 20, p. 58 , September 18, 1995.

[Cowa95] Donald D. Cowan and Carlos J. P. Lucena, "Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse", *IEEE TRANSACTION OF SOFTWARE ENGINEERING*, Vol. 21, No. 3, pp. 229-243, March 1995.

[Cox 96] John Cox, "Legacy transition tools aid move to client/server", *Network World*, Vol. 13, No. 5, p. 33, January 29, 1996.

[Davi95] Beth Davis, "Legacy-SNMP integration confronted", *Communications Week*, No. 562, pp. 1-3, June 19, 1995.

[Dede95] Guido Dedene and Jean-Pierre De Vreese, "Realities of Off-Shore Reengineering", *IEEE Software*, Vol. 12, No. 1, pp. 35-45, January 1995.

[DeMa96] Tom DeMarco, and Ann Miller, "Managing Large Software Projects", *IEEE Software*, Vol. 13, No. 4, pp. 24-27, July 1996.

[Dods96] Bill Dodson, "Inheriting a legacy system", *Data Based Advisor*, Vol. 14, No. 3, pp. 102-104, March 1996.

[Ecke95a] Wayne W. Eckerson, "Building the legacy systems of tomorrow: recipes for prevention and integration, part I", *Open Information Systems*, Vol. 10, No. 11, p. 2, November 1995.

[Ecke95b] Wayne W. Eckerson, "Building the legacy systems of tomorrow", *Open Information Systems*, Vol. 10, No. 12, p. 2, December 1995.

[Edge95] EDGE: Work-Group Computing Report, "Legacy migration: CST ships powerful legacy migration software; unique tool migrates legacy applications into Visual Basic applications for a client/server environment 100 percent automatically", EDGE Publishing, Vol. 6, No. 278, p. 56, September 18, 1995.

[Fafc94] Danielle Fafchamps, "Organizational Factors and Reuse", *IEEE Software*, pp. 31-41 , September 1994.

[Faya96] Mohamed E. Fayad, Wei-Tek Tsai, and Milton L. Fulghum, "Transition to Object-Oriented Software Development", *COMMUMINCATON OF THE ACM*, Vol. 39, No. 2, pp. 108-121, February 1996.

[Fole90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, second edition, ADDISON WESLEY PUBLISHING COMPANY, INC., U.S.A, 1990.

[Frak94] William B. Frakes, and Sadahiro Isoda, "Success Factors of SYSTEMATIC REUSE", *IEEE Software*, pp. 14-19 , September 1994.

[Frak96] William B. Frakes, and Christopher J. Fox, "Quality Improvement Using A Software Reuse Failure Modes Model", *IEEE*

TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 22, No. 4, pp. 274-279, April 1996.

[GeoQ96] GeoQuest, *First In Data Management Service Solutions*, Schlumberger, GMP-6601, July 1996.

[Gris95] William G. Griswold and Darren C. Atkinson, “Managing Design Trade-Offs for a Program Understanding and Transformation Tool”, *J. SYSTEMS SOFTWARE*, Vol. 30, pp.99-116, 1995.

[Haag96] Stephen Haag et al, “Quality Function Deployment”, *COMMUNICATION OF THE ACM*, Vol. 39, No. 1, pp. 41-49, January 1996.

[Hamm93] M. Hammer, and J. Champy, *Reengineering the Corporation*, HarperBusiness, New York, USA, 1993.

[Hear86] Donald Hearn, and M. Pauline Baker, *COMPUTER GRAPHICS*, Prentice-Hall, U.S.A, 1986.

[Henr95] Emmanuel Henry, and Benoit Faller, "Large-Scale Industrial Reuse to Reduce Cost and Cycle Time", *IEEE Software*, pp. 47-53, September 1995.

[Hsia96] Pei Hsia, "Making SOFTWARE DEVELOPMENT Visible", *IEEE Software*, pp. 23-26, March 1996.

[Ingl94] Kathryn A. Ingle, *Reverse Engineering*, McGraw-Hill, Inc., USA. 1994.

[Isak96] Tomas Isakowitz, and Robert J. Kauffman, "Supporting Search for Reusable Software Objects", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Vol. 22, No. 6, pp. 407-423, June 1996.

[Kasp94] Donna Kaspersen, "FOR REUSE, PROCESS AND PRODUCT BOTH COUNT", *IEEE Software*, p. 12, September 1994.

[Koen91] J. Koenemann, and S. P. Robertson, "Expert Problem Solving Strategies for Program Comprehension", *Proc. Human Factors in Computing Systems, CHI'91*, New Orleans, pp. 125-130, May 1991.

[Lede92] Albert L. Lederer, and Jayesh Prasad, "Nine Management Guidelines for Better Cost Estimating", *COMMUMINCATON OF THE ACM*, Vol. 35, No. 2, pp. 51-59, February 1992.

[Lern94] Moisey Lerner, "Software maintenance crisis resolution: the new IEEE standard", *Software Development*, Vol. 2, No. 8, pp. 65-69, August 1994.

[Lewi81] Harry R. Lewis, and Christos H. Papadimitriou, *ELEMENTS OF THE THEORY OF COMPUTATION*, Printice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A, 1981.

[Lewi96] Ted Lewis, "The limits of innovation", *COMPUTER*, pp. 7-9, April 1996.

[Lewi97] Ted Lewis, "If Java Is the Answer, What Was the Question?", *COMPUTER*, Vol. 30, No. 3, pp. 133-136, March 1997.

[Lim 94] Wayne C. Lim, "Effects of Reuse on Quality, Productivity, and Economics", *IEEE Software*, pp. 23-30, September 1994.

[Litt86] D. C. Littman, J. Pinto, and E. Soloway, "Mental Models and Software Maintenance", *Empirical Studies of Programmers: Fifth Workshop*, Soloway and Iyengar, eds., Ablex Publishing Corporation, pp. 80-98, 1986.

[Loud95] Stephen Loudenrmilk, "Prime paths for SNA routing: options abound for integrating host-based legacy apps into a PC LAN world", *LAN TIMES*, Vol. 12, No. 14, pp. 86-87 , July 24, 1995.

[Luca96] Henry C. Lucas, Jr. Donald J. Berndt, and Greg Truman, "A Reengineering Framework for Evaluating a Financial Imaging System", *COMMUMINCATON OF THE ACM*, Vol. 39, No. 5, pp. 86-96, May 1996.

[Lyon95] Daniel Lyons, "Wrapping up legacy code", *InfoWorld*, Vol. 17, No. 26, pp. 59-60, June 26, 1995.

[Mack95] Stephen R. Mackey, and Lynn M. Meredith, "Software Migration and Reengineering: A Pilot Project in Reengineering", *The Journal of Systems and Software*, Vol. 30, No. 1 & 2, pp. 137-150, July-August 1995.

[Mack96] Karen Mackey, “Why Bad Things Happen to Good Projects”, *IEEE Software*, pp. 27-32, May 1996.

[Mark95] Lawrence Markosian, “Salvaging legacy systems”, *Computing Canada*, Vol. 21, No. 13, p. 44, June 21, 1995.

[Mart85] J. Martin, and C. McClure. *Structured Techniques for Computing*, Printice-Hall, Englewood Cliffs, new Jersey, USA, 1985.

[Mart97] Robert A. Martin, “Dealing with Dates: Solutions for the Year 2000”, *COMPUTER*, Vol. 30, No. 3, pp. 44-51, March 1997.

[Mayr96] A. von Mayrhauser, and A. M. Vans, “Identification of Dynamic Comprehension Processes During Large Scale Maintenance”, *IEEE TRANSACTION OF SOFTWARE ENGINEERING*, Vol. 22, No. 6, pp. 424-437, June 1996.

[Meye88] Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall International Series in Computer Science, UK, 1988.

[McLe96] Jean McLendon, and Gerald M. Weinberg, "Beyond Blaming: Congruence in Large Systems Development Projects", *IEEE Software*, Vol. 13, No. 4, pp. 33-42, July 1996.

[Mili95] Hafedh Mili, Fatma Mili, and Ali Mili, "Reusing Software: Issues and Research Directions", *IEEE TRANSACTION OF SOFTWARE ENGINEERING*, Vol. 21, No. 6, pp. 528-562, June 1995.

[Mona95] Curt Monash, "Beyond the myth of legacy migration", *Software Magazine*, Vol. 15, No. 6, pp. 130-131 , June 1995.

[Moor94] James W. Moore, David Emery and Roy Rada, "Language-Independent Standards", *COMMUNICATIONS OF THE ACM*, Vol. 37, No. 12, pp. 17-20, December 1994.

[Mull89] Mark Mull, ***Object-Oriented Program Design With Examples in C++***, Fifth printing, July 1992, Addison-Wesley Publishing Company, Inc., U.S.A, 1989.

[Neum95] Peter G. Neumann, "Reviewing the Risks Archives", *COMMUNICATIONS OF THE ACM*, Vol. 38, No. 12, p. 138, December 1995.

[Ning94] Jim Q. Ning, Andre Engberts, and W. (Voytek) Kozaczynski, "Automated Support for Legacy Code Understanding", *COMMUNICATION OF THE ACM*, Vol. 37, No. 5, pp. 50-57, May 1994.

[O'Co94] James O'Connor, Catharine Mansour, Jerri Turner-Harris, and Grady H. Campbell, "Reuse in Command-and-Control Systems", *IEEE SOFTWARE*, pp. 70-79, September 1994.

[Offu93] A. J. Offut, M. J. Harold, and P. Kotle, "A Software Metric System for Module Coupling", *Journal of System and Software*, Vol. 20, No. 3, pp. 295-308, March 1993.

[Patr95] Andy Patrizio, "Magic 6 supports Windows clients: forms editor available for legacy systems", *PC Week*, 1995, Vol. 12, No. 25, p. 27, June 26.

[Pfle96] Shari Lawrence Pfleeger, "Measuring Reuse: A Cautionary Tale", *IEEE Software*, Vol. 13, No. 4, pp. 118-127, July 1996.

[Phil96] Dwayne Phillips, "Project Management: Filling in the Gaps", *IEEE Software*, Vol. 13, No. 4, pp. 17-18, July 1996.

[Prem94] William J. Premerlani, and Michael R. Blaha, "An Approach for Reverse Engineering of Relational Databases", *COMMUNICATIONS OF THE ACM*, Vol. 37, No. 5, pp. 42-49, May 1994.

[Rama96] C. V. Ramamoorthy and Wei-tek Tsai, "Advances in Software Engineering", *COMPUTER*, pp. 47-58, October 1996.

[Raz 93] Tzvi Raz, and Alan Yaung, "Process Clustering with an Algorithm Based on a Coupling Metric", *J. SYSTEMS SOFTWARE*, Vol. 22, pp.217-223, 1993.

[Ricc95] Mike Ricciuti, "Rule Finder traces rules in legacy apps.", *InfoWorld*, Vol. 17, No. 42, p. 28, October 16, 1995.

[Ridd89] William E. Riddle, "Session Summary: Opening Session", *Proceedings of the 4th International SOFTWARE PROCESS WORKSHOP, ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, Vol. 14, No. 4, pp. 5-10, June 1989.

[Sage95] Andrew P. Sage, "System Engineering and System Management for Reengineering", *The Journal of Systems and Software*, Vol. 30, No. 1 & 2, pp. 3-25, July-August 1995.

[Samu90] Pamela Samuelson, "Reverse Engineering Someone Else's Software: Is it Legal?", *IEEE Software*, pp. 90-96, January 1990.

[Scha90] S. R. Schach, *Software Engineering*, Richard D. Irwin, and Asken Associates, Asken Associates Incorporated Publishers, USA, 1990.

[Semm95] R. D. Semmel, and M. Wilson, "Guest Editors' Corner, Reengineering Complex Systems", *The Journal of Systems and Software*, Vol. 30, No. 1 & 2, pp. 1-2, July-August 1995.

[Selk96a] Ted Selker, "New Paradigms for Computing", *COMMUNICATIONS OF THE ACM*, Vol. 39, No. 8, pp. 29-30, August 1995.