

**A Discipline of Quantitative Design Metrics
for Very Large Software Systems**

by

Jeffrey Martin Harris

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba

© Jeffrey Martin Harris, 1987

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-37161-7

A DISCIPLINE OF QUANTITATIVE DESIGN METRICS
FOR VERY LARGE SOFTWARE SYSTEMS

BY

JEFFREY MARTIN HARRIS

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

© 1987

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

ABSTRACT

When designing very large software systems it is difficult to maintain consistency throughout the development lifecycle because such systems cannot be mentally managed by one person. The lack of consistency, or "conceptual integrity", results in systems that experience higher development costs and decreased flexibility. A high degree of flexibility is crucial in very large software systems; these systems are too large to recreate for every new adaptation so they must evolve. This thesis presents techniques to increase conceptual integrity from the design stage through to the maintenance stage of the software development lifecycle.

First, a quantitative metric is developed for module cohesion; this is done by externalizing the five factors that influence module cohesion.

Second, a quantitative metric is developed for module coupling; this is done by externalizing the seven factors that influence module coupling.

Third, the fundamental concepts of a tool for modelling a software system design, called SPAM (an acronym for System Planning Model), are presented. The SPAM tool bases system design on the data flow diagram methodology by divorcing data elements from control elements early in the design stage. The prospective design is represented with a system schema, which allows flexible scoping, connection mapping, system wide cross referencing, version control, project team control, and various other features. SPAM is capable of evaluating a prospective system design before any code has been written by applying the quantitative coupling metric to the design model.

As well as aiding in the design stage of development, extensions to SPAM are introduced for increasing the degree of automation in the programming, testing, deployment, and maintenance stages of development, although the design stage is the focus of this dissertation. It should be noted that this dissertation is concerned with the specifications of SPAM, and does not describe a specific implementation of such a tool.

INDEX TERMS

Software Engineering

Design Metrics

Design Models

Design Representations

Module Cohesion

Module Coupling

Design Methodologies

Programming Environments

ACKNOWLEDGEMENTS

The problems described in this thesis were encountered during the author's tenure as a member of scientific staff at Bell-Northern Research, Ottawa, Canada. The author would like to take this opportunity to thank his colleagues at BNR, particularly David Hudson, Adrian Brown, and Stan Ball, for sharing their enthusiasm, encouragement, and professionalism.

Much of this work has already been published, or has been accepted to be published, as a series of individual papers in various volumes of *Congressus Numerantium*. The author wishes to thank the referees and editors for accepting and publishing these papers.

The author expresses his gratitude to his supervisor, Dr. Howard Ferch, whose help and ideas greatly improved this presentation. The author also thanks the other members of the examining committee, Dr. Donald Cowan, Dr. David Scuse, and Professor Maurice Yunik, for their time and effort.

The author would like to take this opportunity to thank his mentor, Dr. Ralph Stanton, whose patient prodding was the motivating force behind the completion of this work.

It is not an easy task to try to support a family and be creative in one's spare time; the pressures are often taken out on anyone in the immediate vicinity. The author is grateful to those around him, especially his wife Arlene and his son Jonathan, for putting up with him.

This research was made possible in part by the financial assistance of the National Science and Engineering Research Council of Canada through a postgraduate scholarship.

This document was generated on
Wednesday April 1, 1987 at 08:20:30
using Waterloo SCRIPT - Version 84.1 (84OCT19).

DEDICATION

This is for my mother,
who always wanted a Doctor in the family,
and for my father,
may he rest in peace.

CONTENTS

COPYRIGHT NOTICE	ii
ABSTRACT	iii
INDEX TERMS	iv
ACKNOWLEDGEMENTS	v
DEDICATION	vi

<u>Chapter</u>	<u>page</u>
I. INTRODUCTION	1
Background	2
Motivation	3
Method	7
Audience	10
Limit of Scope	13
Organization of This Thesis	15
II. TERMINOLOGY	17
Abstract	17
Introduction	18
Background	18
Goals	19
Elucidating The Definition of a Module	20
What a Module Is Not	21
What a Module Is	24
Goals of Modular Design	27
Non-Traditional Packaging Schemes	30
Connections	31
Timing	34
Visibility	39
Usage of Information	41
Data Interfaces	42
Control Interfaces	45
Aggregates	47
Complexity of Information	49
Myers' Complexity Attributes	51
Homogeneous Complexity	51
Heterogeneous Complexity	51
Atomic Complexity	51
Attributes of Information Complexity	53
Number of Atomics	53

Number of Data Types	53
Order Dependency	54
Presence of Structure Information	55
Ranking The Attributes	56
Linear Complexity	58
Molecular Complexity	59
Abstract Complexity	60
Data Complexity Summary	60
Conclusion	61
Summary	61
Analysis	61
 III. COHESION	 62
Abstract	62
Introduction	63
Background	63
Structured Design	64
Design Metrics	65
Review of Cohesion	65
Coincidental Cohesion	68
Logical Cohesion	69
Temporal Cohesion	70
Procedural Cohesion	71
Communicational Cohesion	71
Sequential Cohesion	72
Functional Cohesion	73
Informational Cohesion	74
The Factors That Influence Cohesion	75
Internal Aspects of a Module	77
Functional Elements	77
Data Elements	79
Time Elements	80
Identifying The Factors That Influence Cohesion	83
Number of Data Elements	83
Number of Logical Operations	84
Number of Functional Segments	84
Dependency of Time Elements	84
Exclusivism	85
Module Cohesion Coefficient	85
Weight Factors	87
Comparing The Factors to The Accepted Classifications	87
Modifying The Accepted Cohesion Classifications	90
Decomposing Logical Cohesion	90
Removing Coincidental Cohesion	91
Comments on Gathering Empirical Data	92
Conclusion	92
Summary of Results	92
Critique of Results	94

IV. COUPLING 96

Abstract	96
Introduction	97
Review of Coupling	97
Myers' Work on Coupling	98
Content Coupling	98
Common Coupling	100
External Coupling	102
Control Coupling	104
Stamp Coupling	110
Data Coupling	111
No Direct Coupling	112
Summary of Myers' Coupling Classifications	113
Deficiencies in the Coupling Classifications	114
Motivation for Developing Factors	120
Yourdon and Constantine's Work on Coupling	121
The Factors That Influence Module Coupling	124
Number of Connections	125
Interface Type of Connection	126
Complexity of Information	128
Usage of Information	130
Binding Time	131
Direction of Information Flow	132
Number of Distinct Destinations	133
Describing Classifications With the Factors	135
Describing Null Coupling with Factors	135
Describing Data Coupling with Factors	135
Describing Stamp Coupling with Factors	136
Describing Control Coupling with Factors	136
Describing External Coupling with Factors	137
Describing Common Coupling with Factors	138
Describing Content Coupling with Factors	138
Disagreements Between Factors and Classifications	139
Dimensions of Coupling	141
A Theoretical Quantitative Coupling Metric	144
Bounds of the Metric	144
Factor Weight and Factor Precedence	144
An Algorithm for Coupling Dependency	146
Normalization of CD(X:)	147
Notes on Gathering Empirical Data	148
Conclusion	150

V. SOFTWARE SYSTEM ENGINEERING 152

Abstract	152
Introduction	153
Overview of This Chapter	154
Design Documentation	154
Software Design Metrics	157
Models	159

The Philosophical Foundations of SPAM	161
Scope	165
History of Scoping	165
Flexible Scoping	168
Components	169
System Schema	170
Nodules	172
Sub-Systems	173
Data Entities	177
Connections	177
Data Lifespan and Visibility	180
Test Data	180
Reuseability	182
Portability	182
The Underlying Programming Language	183
Language	184
The Interactive Nodule Editor	187
The SPAM Integrated Compiler	188
Compilation Units	188
Separate Compilation Consistency	189
Changes to the System	189
Images	190
Schema Versions	191
Features	191
Conditional Compilation	192
Fixes	193
Summary of Flexibility Provisions	193
Provisions for Trackability	193
Conclusion	195
Summary	195
Analysis	196
VI. CONCLUSION	197
Reviewing The Original Objectives	197
Review of Results	198
Improved Terminology	198
Cohesion Factors	199
Coupling Factors	199
Software System Engineering	199
Additional Work	200
REFERENCES	203
INDEX OF DEFINITIONS	208

LIST OF ILLUSTRATIONS

<u>Figure</u>	<u>page</u>
1. Module Size versus Program Complexity	28
2. Temporally Synchronized	35
3. Temporally Coincidental	36
4. Temporally Permanent	37
5. Temporally Persistent	38
6. Attributes of Information Complexity	56
7. Comparing Cohesion Classifications and Factors . . .	89
8. Common Coupling	100
9. Scope of Effect/Scope of Control Rule	109
10. Myers' Qualitative Coupling Values	115
11. Coupling Dependency	116
12. Connection Density: Scenario 1	133
13. Connection Density: Scenario 2	133
14. Three Dimensions of Coupling	142

Chapter I

INTRODUCTION

"The whole of science is nothing more than a refinement of everyday thinking."

- Albert Einstein

Computer technology has given man the capability of creating systems that are so complex that their behavior cannot be predicted under many circumstances. As computerized systems become more and more predominant, society as a whole is increasingly dependant on their quality and reliability. Computers are now engaged in systems that were traditionally dominated by human beings. Examples may be drawn from transportation, communications, finance, as well as host of others too numerous to enumerate. With the advent of computers, many of these systems have expanded to the point where humans are incapable of resuming the task should the machines fail. An example of such a system is the telephone network which is moving towards total computer control by the year 2010 [7]. It has been estimated that if the North American telephone system were to revert back to total human control it would take every female between the ages of 16 and 60 to act as telephone operators [8]. Of course it may be counter argued that if that many people were occupied at work the load on the telephone system would be greatly diminished!

The amount of computer software controlling important functions in society is likely only to increase in the future. As older software becomes obsolete it may be physically impossible, and certainly prohibitive in the monetary sense, to create replacements for the obsoleted software. This suggests that the software created today should be adaptable to respond to changing requirements. The formalization of software engineering techniques is an increasingly important topic as more and more systems rely on some degree of computer control.

1.1 BACKGROUND

Computer science has benefited from the formalization of software engineering techniques. Methodologies such as structured programming, structured analysis, structured design (also known as composite¹ design), structured testing, and a host of others have contributed to the formalization of software composition. The term "structured" seems to appear in computer science literature frequently in the past number of years when referring to software development techniques. It seems that "structured" is added to articles and techniques because it is perceived by the software engineering community that "to be structured is to be good". What does the word structured mean in the context of computer science? The Webster's dictionary [56]

¹ Myers used the term "composite design" in [43], [44], [45] to describe what is now commonly known as "structured design".

defines structured as

"to build or construct in a definite pattern of organization".

This definition intimates that there is a formal plan involved in the construction process. Indeed, it seems that wherever the word "structured" appears in software engineering literature the reader may substitute the word "formalized" to obtain a clearer understanding of the intent of the techniques.

1.2 MOTIVATION

The current state of software engineering has the system development process refined to a series of steps which are generally referred to as the **Software Development Lifecycle (SDLC)** [51]. These steps are also known as the **Waterfall Model** of System Development [10]. Although the exact names of the steps in the SDLC vary from author to author and the levels of refinement differ, the aim of the process is generally accepted. The phases of the software development lifecycle are:

1. **Requirements.**

This stage is sometimes referred to as the systems analysis stage because it is here that the problem is described and a fundamental understanding of the problem obtained. Usually a feasibility study is undertaken at this stage to determine whether it is appropriate to computerize the application.

2. Specifications.

The specifications stage involves the formalization of the data gathered into a clear and concise statement of what the system is to do. These specifications describe what the system is intended to do from the users' perspective as opposed to the designers' perspective.

3. Design.

The design stage involves detailing system entities, referred to as modules, and their interconnections. The design produced is independent of any computer language or implementation.

4. Programming.

The programming stage involves implementing the system design in some computer language.

5. Testing.

Testing is the stage where the program's quality is demonstrated and the program is shown to be reliable to an acceptable level of confidence.

6. Deployment.

Deployment is the process of introducing the finished system into its intended environment.

7. Maintenance.

Maintenance refers to changes made to the system after delivery to the customer.

Recently some research has been done in an effort to automate one or more stages of the SDLC. The automation of all or part of the SDLC would be advantageous because, left undetected, an error that propagates through the entire cycle is costly to correct. The SDLC is highly labour intensive and any increase in automation would mean a decrease in labour costs and an increase in reliability and flexibility. For example, if there were an error in the specifications for a system that went undetected through to the testing stage, an automated life cycle would allow the software engineer to change the incorrect portion of the specifications and regenerate the system. Essentially this means that the only labour involved in the construction of the system would be the analysis of requirements and the writing of the specifications. Some researchers believe that virtually all labour can be eliminated from the software development process, and their research is geared in that direction. (Akin to pruning a tree while sitting on a branch.)

Very few people claim to have a contrivance that automates the SDLC [35]. This is because it is not an easy thing to do. Specifications are almost always written in a natural language like English, which means that they are ambiguous, imprecise, and incomplete. Mathematical-like software specification languages are required which are capable of describing a problem in a non-procedural manner.

Many of the fourth generation report generating languages are a first step in this direction. The research in this field continues and software engineers shall have to be content for the time being with the automation of individual parts of the SDLC. Ultimately, the fate of any procedure that does not require creative effort is automation.

While it is theoretically known that errors that propagate through the SDLC are costly to correct, it is much more dramatic to show the amount of time actually spent on each phase of the SDLC for a number of systems. This has been empirically measured [34]. As it turns out, about 67% of SDLC effort goes into maintenance and only about 5% goes into the design stage. This seems intuitively wrong, especially since software is intangible and does not degenerate with time as would a road or a bridge. Instinctively one would think that more effort would go into creating the system than maintaining it.

The lack of automation in the software development lifecycle is evidence that we simply do not fully understand the process well enough yet. The goal of this thesis is to identify and formalize new techniques, create new tools, and refine manners of thinking in the design stage of the SDLC. An improved design stage would catch errors earlier in the development and permit the process to be more flexible in the event that an unplanned change to the system was required.

The software crisis comes into focus where very large software systems are involved. (Very large is defined to be 10 million lines of source code or more.) An article in IEEE COMPUTER on the subject of the Strategic Defense Initiative (SDI) [46] reported that a software project of 10 million lines would contain about 5000 errors, or even more, at the time it became operational. It is not obvious to many software professionals that there is a software crisis with respect to very large software systems; this is because these people have not come into contact with software systems of such immense size. In [46], David Parnas is quoted as follows with respect to very large software systems:

"Those who think that software designs will become easy, and that errors will disappear, have not attacked substantial problems."

The only hope for very large software systems is a much more automated, integrated, and formalized way of managing them.

1.3 METHOD

In this thesis a set of quantitative software design metrics is developed that measure module coupling and cohesion. The concepts of module coupling and module cohesion were pioneered by Glenford Myers [40],[41],[42],[43],[44], and [45]. Module coupling is a measure of the inter-relatedness of a set of modules. The higher the degree of inter-relatedness, the less independent the modules are. Thus,

low coupling between modules is desireable. Module cohesion, also known as module strength, is a measure of how the elements internal to a module are related. The higher the degree of cohesion, the more tightly bound the statements in the module are. In effect, the cohesion metric measures to what degree a collection of program elements deserves to be a module. For any given program it is desireable to minimize coupling and maximize cohesion.

This thesis develops a set of quantitative metrics for module coupling and module cohesion. Before proceeding, the term "metric" should be defined. An elegant and concise description of software metrics is given by Tom DeMarco in [15], chapter 7, pages 49 through 61. It is tempting to paraphrase DeMarco because he is so succinct; that chapter of his book should be read. The Webster's dictionary [56] defines metric to be:

"Of or relating to an art, process, or science of measuring."

In this thesis two different kinds of metrics are discussed, **qualitative metrics** and **quantitative metrics**. An example of a qualitative metric might be "We played a good game even though we lost". An example of a quantitative metric might be "The score was 29-30 for them". A quantitative metric attaches a numerical value to the item being measured while a qualitative metric uses non-numerical descriptive phrases.

The presence of a quantitative metric does not imply that there is no human judgement involved in the measurement process. It would be ideal if human judgement could be eliminated entirely, but this is not always possible. In the above example, even though the score evaluated to 29-30, the life of the game was filled with judgement calls by the referees. The final score 29-30 is the culmination of a series of small independent judgements by the referees and the players. Even so, the score 29-30 is more descriptive and meaningful than the phrase "we lost".

The current design metrics for coupling and cohesion are qualitative, unlike the quantitative ones that are developed in this thesis.

The enhancements that are made to coupling and cohesion are used to develop a tool for modelling the design of a software system, called SPAM (an acronym for System Planning Model). In other scientific and engineering disciplines models are often employed to gain a better understanding of a problem area because:

- Models are cheaper to construct than the product itself.
- It takes less time to construct a model of the product than the product itself.
- Since the model is small it is easy to study in isolation.

In this dissertation a model of a software system is constructed based on the data flow methodologies defined by DeMarco [14], and Gane and Sarson [19]. The design model is called a **system schema**. Program data elements are divorced from program control elements early in the design stage. The coupling metric developed in this thesis is applied to the model to study the model for weaknesses.

1.4 AUDIENCE

The intended audience of this dissertation is the software engineering community. Before proceeding with the matter at hand, the term "software engineer" begs to be defined. Software engineering is a relatively new field when compared to other scientific and engineering disciplines. What is a software engineer and what does he do? An accurate and descriptive definition of software engineering [10] is given as:

"Software Engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation."

It should be emphasized that software engineering is related to other engineering disciplines only in name and philosophy; software engineering principles are equally as applicable to business systems as to scientific systems. By examining the definition given above it may be concluded that the software engineer is a software expert. Given that

understanding, what are the relationships between the programmer, the systems analyst, the systems designer, the computer scientist, and the software engineer? Is the computer scientist not a software expert? The systems analyst is an expert at the requirements and specifications stages of the SDLC. The systems designer is an expert at the design phase of the SDLC, and the programmer an expert at the programming phase. A programmer/analyst is an expert at the requirements, specifications, design, and programming phases of the SDLC. The software engineer is a software expert who is intimate with all phases of the SDLC. The terms programmer, system analyst, and system designer arose historically from the order in which the SDLC phases were identified.

An engineer has the function of applying known solutions to new problems in an attempt to satisfy user requirements. In effect, an engineer is someone that builds tools for others to use. This is especially true in software where the product that is developed is often the means to some other end (eg. compilers, data bases, text editors, etc.). What if a problem requires a new technique unknown to the software engineer? The computer scientist is the individual that creates the new techniques for the software engineer. That is to say, the computer scientist is the theoretician while the software engineer is the practitioner. The relationship between a computer scientist and an engineer is

similar to the relationship between a mathematician and an engineer. The mathematician creates a general solution for an equivalence class of problems; the engineer identifies a particular problem as belonging to a certain equivalence class and applies the known solution.

It has been observed in many software development projects that the people acting as software engineers have little or no background in computer science [28]. These 'born-again'² computer scientists come from other disciplines such as engineering, mathematics, accounting, etc. For example, of the 1200 staff working on Bell-Northern Research's (BNR) **DMS (Digital Multiplexed Switch)** less than 20% had a formal software background; the remainder were largely electrical engineers with little or no formal software background. By either design or circumstance it seems to be the rule rather than the exception to employ such people in the construction of all kinds of software, some of which is of a life and death critical nature. It is important to point out this abnormality at a time when management is concerned with the increasing costs of software. Management employs people, who are undoubtedly gifted in their own field but never the less know very little about computer science, to construct

² The expression "born-again" is a biblical term [5] used to refer to a person who has been awakened to a new doctrine. The term is used above in a tongue in cheek fashion to refer to people who were not specifically trained as software engineers but feel strongly that they are qualified to produce software systems.

elaborate software systems. Management then wonders why costs overrun and projects are late. The analogous situation would be to hire a lawyer, who is probably a very clever fellow when it comes to the law, to remove an appendix; removing an appendix is a simple procedure by surgical standards, yet most would agree that it is a complex problem if the practitioner is not surgically trained. Regardless of why this practice seems to be taking place in the software industry, the fact is it just is. The reversal of this trend is a political adventure that is beyond the scope of this thesis³.

This thesis introduces new techniques that will aid software engineers, whatever their background, in building reliable, flexible, and cost-effective software systems.

1.5 LIMIT OF SCOPE

As well as being theoretical in nature, the concepts presented in this thesis are intended to apply to a limited class of systems, namely to telephony type digital communications switches[8],[7],[6],[9]. Some aspects of the presentation are general enough to be applied to all systems in general, although this was not the goal. Below are a few characteristic features of such systems. This is not intended to be an exhaustive list; rather, it is intended to give the reader the flavour of these systems.

³ George Orwell said, "In the end, everything is political."

- The systems are totally RAM based in real memory with no dynamic address translation or virtual memory. The size of real memory is large, between 6 and 12 megabytes for data and another 6 to 12 megabytes for program.
- Data bases and files are maintained in RAM for fast access. The only external file medium is a magnetic tape log of toll transactions (for toll switches only).
- The machines and their associated systems run 24 hours per day, 365 days per year without interruption. The computer runs off of the same 24 volt battery supply that provides power for the telephone system, therefore the power supply is never interrupted.
- The hardware is designed to be extremely reliable, often with redundant CPU's and memories that execute in parallel. The rated unscheduled down time for such systems is 6 minutes every 50 years.
- Security in the software is not a consideration. The machines are logically and physically secured from tampering. The machine is only programmed by authorized personnel.
- Speed of execution and efficiency is a prime consideration. The software runs in real-time. The machine must be capable of handling network traffic without delay during peak traffic conditions.
- The software running on these systems is large, in the order of 3 to 6 million lines of high level source

code. The system is literally composed of thousands of modules.

- The systems use state of the art software techniques and tools to increase flexibility. The systems are constantly undergoing change.

The systems described are BNR's DMS 100, 200, and 300 series.

1.6 ORGANIZATION OF THIS THESIS

Chapters 2 through 5 of this dissertation contain a short introduction and abstract at the beginning of each chapter. This is because the results from each of these chapters can be studied in isolation.

Chapter 2 of this thesis addresses many of the shortcomings of the current terminology that is used to describe concepts in software design. Many of the concepts discussed in chapter 2 are folklore, but the terminology assigned to the concepts is new. As well as discussing many known concepts, several new concepts are introduced. The terminology created in chapter 2 is used in later chapters.

Chapter 3 of this thesis discusses and reviews the accepted module cohesion metrics. The factors that influence module cohesion are externalized and a theoretical formula for a quantitative cohesion metric is presented.

Chapter 4 of this dissertation reviews and discusses the accepted module coupling metrics. The factors that influence module coupling are externalized and a theoretical formula for a quantitative module coupling metric is presented.

Chapter 5 of this dissertation outlines and discusses the concepts involved with an automated tool for representing software system designs. The design technique developed is new and as such only the theoretical concepts and requirements are outlined; the implementation details of the tool are not discussed.

Chapter 6 of this thesis presents a summary of the topics discussed in this thesis and outlines further work that should be undertaken.

Chapter II

TERMINOLOGY

"By relieving the brain of all unnecessary work, a good notation (terminology) sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race."

- A. N. Whitehead

2.1 ABSTRACT

The state of the art in terminology used to describe the internal relationships of modular software systems is incomplete, imprecise, and in some cases incorrect. It is difficult to communicate ideas in an articulate fashion when the terminology is so vague. As well, it is difficult to create new concepts when the current knowledge base lacks the lucidity that a proper terminology would provide. Often the definition creates the meaning, and the definitions are poor. This chapter develops a comprehensive terminology that can be used to model the internal relationships of program elements in a traditional modular programming environment. As well as providing clarification of accepted terms, this chapter also introduces several new terms. The resulting terminology provides the software designer with a consistent tool for understanding the internal relationships of software systems.

2.2 INTRODUCTION

2.2.1 Background

For as long as computer software has been written there has been a search for improved techniques in an effort to take the intuitivism out of software composition. These techniques vary and cover a wide range of topics from development methodologies, planning and estimating procedures, diagramming techniques, to a host of others too numerous to enumerate. The state of the art of software construction is a highly intuitive and judgemental process. The intent of inventing new techniques and formalisms is to externalize the underlying concepts of software composition. Although many individuals, the author included, consider themselves to be "software engineers", software composition is far from being a true scientific or engineering discipline. Even so called "experts" in the software engineering field differ by as much as three hundred percent or more [52] when asked to estimate the resources required to complete a given software project. Individuals must rely largely on intuition rather than a set of accepted metrics. A higher level of understanding might promote software engineering from an art to a true scientific or engineering discipline.

There are several reasons why the current level of understanding of software concepts is inadequate and should be improved. Software composition is a creative process

that must be performed by individuals so gifted. The externalization of the underlying concepts of software might allow individuals of a less creative nature to develop software that can now only be written by highly creative individuals. Ultimately, the process may be simplified to the point where software could compose software. While there has been success with automated software development in limited areas, in general, such projects have not been fruitful to date. (Parser generators are an example of automated software development that has been successful in one limited well defined scope.)

2.2.2 Goals

One of the aspects of software that is in dire need of improvement is terminology. Often the terminology that is assigned to objects and ideas reflects and constrains one's thinking on those topics. The current terminology does not accurately describe the state of the art of software. Given that, it cannot be expected to lend itself to new and undiscovered areas. It is this author's opinion that this is one of the main reasons that the general case of automated software development has not been successful.

The purpose of this chapter is to define a terminology to describe the aspects of software design associated with coupling and cohesion [43], [45]. In particular, this chapter defines terminology to describe the interactions and

complexity of both program executable elements and program data elements. As well as introducing new terminology, this chapter will also reexamine many of the accepted terms. The ultimate goal of these definitions is to work towards a quantitative metric for module coupling and cohesion, which is measured qualitatively at the moment.

2.3 ELUCIDATING THE DEFINITION OF A MODULE

One of the aspects associated with structured design [65] is the decomposition of a problem into a number of separate pieces called modules. There are a number of varied techniques for decomposing a problem into modules, the discussion of which are beyond the scope of this thesis. Each module models only one particular aspect of the problem and as such, each module is easier to implement than the whole problem. Before such terms as "modular design", "modular systems", and "modularity" can be discussed it is essential that the term "module" be better understood. The term "module" is probably one of the most used and least understood terms used in software engineering. Even those who intuitively understand the concept are not capable of giving a coherent definition. The purpose of these next two sections is to clarify the term "module".

2.3.1 What a Module Is Not

Yourdon, in [64] (chapter 3), spent an entire chapter trying to describe a module. Yourdon does not actually give a formal definition of a module. Yourdon spends a lot of time telling the reader what a module is not, but very little time on what a module is. Most notably, Yourdon intimates that a module is not:

- **a fixed but otherwise arbitrary number of programming language statements.** While experience suggests that most good modules have fewer than 50 program statements, this does not imply that any set of less than 50 program statements is a module, nor does it preclude a set of greater than 50 program statements from being a module.
- **a unit of machine resources.** While experience shows that most good modules have small resource requirements, this does not imply that anything that fits into 4K of memory or on 1 track of disk is a module, nor does it preclude anything that requires larger resources from being a module.
- **a unit of man resources.** While experience suggests that modules should be written and debugged by only one person, this does not imply that the amount of code generated by one person in some fixed but otherwise arbitrary unit of time is a module, nor does it preclude anything written by more than one person from being a module.

Yourdon (on page 93) admitted that pinpointing the definition of modularity is difficult and left the reader with the following bit of wisdom:

"Modularity in a computer program seems to be very much like honesty in a politician: everyone claims that he has it, but it seems that nobody knows what it is, how to instill it, how to achieve it, or how to enforce it."

It seems that the difference between a modular program and a non modular program in the eyes of many programmers can be described using the same axiom that distinguishes art from pornography, namely, "I know it when I see it".

The definition of a module is complicated by the new package oriented and object oriented programming languages. (A package or an object encapsulates both procedures and data in one program scope [1], [9], [54], [39], [47], [60], [61], [12], [50], [20], [48].) Many programmers are wrongly under the impression that a module is another name for a compilation unit. Is a package a module? Is a procedure a module? If a package is a module then what is one of the procedures within the package? If a procedure is a module then what is a package? Questions such as these are fundamental and should be answered before the term "module" is used in program complexity problems; it seems unreasonable to compute the complexity of a module in a quantitative fashion when the intuitive definition is elusive.

Clearly, it is difficult to find a real world analog of a module since it does not correspond exactly to any known object. There are many references that discuss modules and their part in modular programming, but there are few that actually define the term module. Independent of how they are worded, most of the software design scriptures allude that:

- A module is part of a program.
- A module has a name.
- A module performs one function.
- A module can be tested separately.
- A module can be compiled separately.
- A module contains both the data elements and executable elements necessary to model one aspect of a problem.

While the above definition seems reasonable, it is in fact inadequate. Consider a program entity that performs a multitude of functions on the same data item, perhaps controlled by passing down an opcode or a flag. While most designers would agree that such an entity is not an ideal module, it is nevertheless a module. The part of the definition that claims that a module performs only one function refers to a good module; in the real world there are always tradeoffs to consider, and as such, modules are often not ideal. In addition to that, not all program entities contain functional elements; some package oriented languages allow packages to contain data and/or type

definitions only. This argument suggests that the number of logical operations that an entity performs does not dictate its ability to be called a module.

The qualification that a module have a name by which it may be referenced leads many programmers to draw the conclusion that "module" is a generic name for subroutine, procedure, function, etc. because each of these program entities has a name. Many individual program statements are labelled with either a name or a number, yet one does not jump to the conclusion that all such labelled statements are modules. To demonstrate that having a name is not a necessary qualification for a module, consider a PL/I procedure with multiple entry points. Although the procedure itself has a name, the name serves no purpose and may be omitted because all transfers could be accomplished via the entry points. Yet, most designers would say that the entity containing the entry points was a module. These arguments suggest that having a name is not necessarily a qualification for a program entity to be a module.

2.3.2 What a Module Is

The problem that is being encountered is the mapping of the logical concept of a "module" onto some physical program entity. That is just the point. A module is a logical concept, not a physical entity. It would be highly desirable if the program's physical boundaries coincided

with the logical boundaries of the modules; if these boundaries were in harmony then program design and maintenance would be simplified. The intent of developing programming languages with varied scope rules such as package oriented languages and object oriented languages is to have the compilation units of a language more closely emulate logical module boundaries.

A module therefore, is whatever the designer defines it to be using the programming language facilities available. This does not mean that the entity that the designer defined as a module is a good module, it is just a module. For example, a designer might say that every set of 20 program statements is a module. If he says that they are modules then they are, even though they may not be good modules. (One must not overlook the extreme remote possibility that dividing a program along arbitrary lines may result in a collection of good modules.)

This discussion brings into focus Yourdon's observation, namely, that even though people decompose their programs into a series of compilation units which they refer to as modules, their programs are not very "modular". The same observation was made by Parnas [49], only in a much more round about way. This is true because there is such a vast difference between a collection of good modules versus a collection of bad modules. A program that is "modular" is composed of good modules and is thus easy to maintain

because program complexity has been reduced. A program that is composed of good modules is easy to maintain because the logical boundaries correspond to the physical boundaries; this is important because the logical structure of the program can be viewed by a reader as part of the physical structure of the program, without having to first understand the workings of the program. This is desirable for the same reasons that indenting the layout of structured control structures in a program is desirable. There is further precedent in storing program source code in a file structure that emulates the logical structure of the program. In essence, information that was previously implicit is now being conveyed explicitly by one means or another.

Any definition given for a module must contain two points: the definition of a module in general and the definition of a good module. Given the above discussion, the definition of a module presented in this thesis is:

A **module** is a collection of program elements encapsulated within a logical boundary. A good module is a collection of program elements that are encapsulated within a logical boundary, exhibit high cohesion, and the logical boundary coincides with physical programming language constructs.

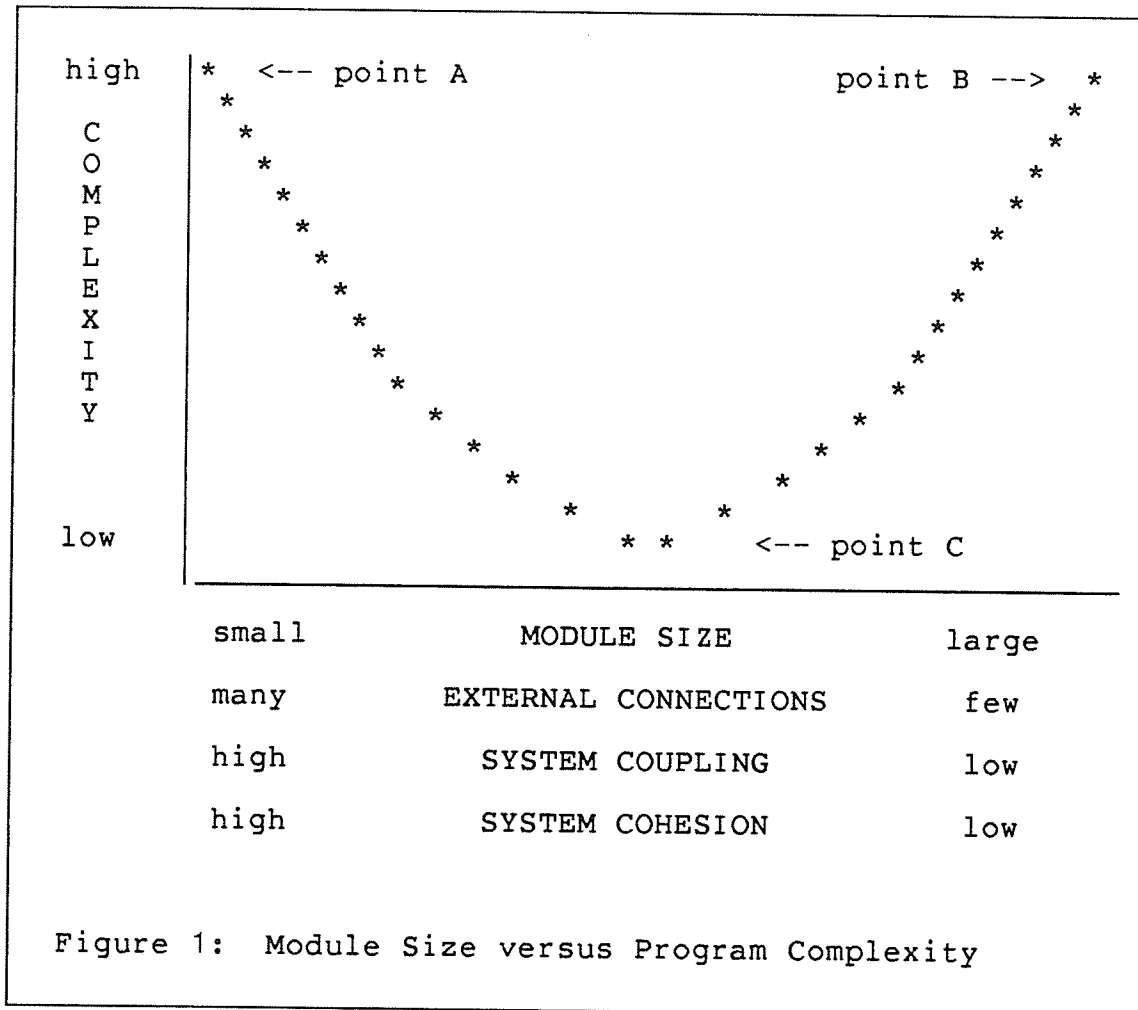
The above definition highlights the fact that a module is a logical program unit. Entities such as procedures, subroutines, functions, packages, and other compilation units are physical program units. Therefore, modular design is not simply the decomposition of one large physical

compilation unit into several smaller physical compilation units; this type of action would result in systems that were not very "modular" (as Yourdon observed).

2.3.2.1 Goals of Modular Design

It has been said [45] that the process of decomposing a program into modules could be carried on to the point where each module consists of only one statement, and that such a design would be preeminent. On the contrary, this would likely not achieve the primary goal of modular design, namely, that of reducing program complexity. Reducing a program's complexity has a positive effect on its cost and maintainability [43], [45]. As more modules are introduced into a design the complexity of the program may tend to increase because the number of connections between modules would increase. The increased number of connections leads to more complicated interactions between the modules, thus obscuring the desired clarity. A design that effectively decreases the overall complexity of a program is a fine (and not well understood) balance between the size of the modules and the number and type of interactions between the modules. While it is desirable to have a large number of modules, it is also desirable to minimize the number of interactions between modules. A good design tends to be a tradeoff between module size and the number of inter-module connections, as depicted by the curve in figure 1. It is already generally accepted that a program containing a small

number of extremely large modules is more complex than it need be. The premise of figure 1, namely that arbitrarily small modules do not reduce program complexity, is supported by a recent empirical study [13]. (Figure 1 is discussed



further in chapter 5.) Obviously figure 1 is a generalization. Different programming problems are of different intrinsic complexity. For example, it is safe to

assume that the program for the sieve of Eratosthenes is less intrinsically complex than the program for the Strategic Defense Initiative (SDI). No matter how good the design of an SDI program might be it would be far more complex than a sieve program. As the intrinsic complexity of a program increases the vertex of the parabola depicting the complexity would move up the Y axis. For any one problem however the shape of the curve would be a truthful description of the design tradeoffs. Thus from the above we arrive at a definition for modular design. In this thesis, modular design is defined to be:

Modular design is the decomposition of a problem into logical units called modules, and then the mapping of those modules onto physical program units with the intent of reducing the overall program complexity. The quality of the design is determined by its effectiveness at reducing program complexity.

The above definition fails to give any insight into what logical elements should be grouped together to form a module. Researchers in the cognitive sciences strongly suspect that such information is domain dependant. In any case, the logical characteristics of the system are determined during the specifications stage of development, not during the design stage. The purpose of the design stage is to propose a computer model of the logically defined system constructed during the specification stage.

2.3.2.2 Non-Traditional Packaging Schemes

Given the above definition of a module, both a package and any procedures contained within that package are modules. In fact what is developed is a hierarchy of modules with all entities at each level being called a module. For example, each procedure within a package exhibits a level of cohesion. The elements of the procedures are the executable statements and the data definitions contained within the procedure. The package also exhibits a level of cohesion. The elements of the package are the procedures and data definitions contained within it. For example, if a package contained a number of procedures and one data item where each procedure performed a different operation on the data item, then the package would exhibit information cohesion [43], [44], [45]. On the other hand, should the procedures and data items within the package be logically unrelated, then the package would exhibit coincidental cohesion [65]. In the same way, packages may be grouped together into sub-systems which also exhibit cohesion. In this case the elements are the packages within the sub-system. Sub-systems may be grouped together into systems, and so on. In this way, the definition of a module and its qualitative metric can be nested. The package concept is a convenient method of introducing a level of scope that superposes the normal module scope.

2.4 CONNECTIONS

The composition of modules to make a program requires that the modules interact with each other. The number and type of the interactions affect the complexity of the program. In order to evaluate the quality of various designs it is important that these interactions be classified and ranked. There are two different types of information that modules may exchange:

- Data information.
- Control information.

Information is conveyed between program segments via connections. A **connection** is a reference from one program element to another. There are connections internal to a module as well as connections between different modules. Each connection can carry either control information or data information. There are **implicit connections** and **explicit connections**, both of which must be considered. For example, there is an implicit control connection between program statements that are executed in a sequential manner within a single module. This can be said of all classic Von Neumann machines. An explicit connection is a named reference to the program element in question.

Connections are synchronous information paths. Connections that are capable of transmitting information and receiving information are defined to be **bi-directional**

connections whereas connections that can either transmit or receive but not both are defined to be **uni-directional connections**. There are two instances of uni-directional connections, one for each direction in which the information may travel.

In analyzing information flow direction one has to be careful when using the relative terms "in" and "out" to describe the movement of data. This is because the flow of data information in a program is not necessarily related to the control flow of the program. For example, a certain module may never be given control yet data may flow in and out of the module. Consider the following scenario:

```
PACKAGE P1 USES P3;
  PROCEDURE S1;
    A := 29;
  END S1;
END P1;

PACKAGE P2 USES P3;
  VAR B: INTEGER;
  PROCEDURE S2;
    B := A;
  END S2;
END P2;

PACKAGE P3;
  VAR A: INTEGER;
END P3;
```

In the above example, package P3 is never given control yet variable A in package P3 moves in via procedure S1 in package P1, and moves out via procedure S2 in package P2. The same sort of scenario can be developed using FORTRAN common blocks. This is why program designers use data flow

diagrams [19] to depict the flow of data in a program whereas other techniques such as flowcharts, hierarchy diagrams, etc. [36] are used to depict the flow of control in a program. Normally the terms "in" and "out" refer to objects being received and transmitted by the module having control respectively. In the analysis of coupling, the correct perspective is from that of the data object being transferred, not from the module performing the transfer. In the terminology of structured systems analysis [19], the data is being transferred to or from a data store. In this thesis, movement of data out of a data store is defined to be a **withdrawal** whereas a movement into a data store is defined to be a **deposit**. A deposit is roughly analogous to the WRITE operation and the withdrawal is analogous to the READ operation. In this context a withdrawal is not destructive and a deposit is destructive.

In accordance with the above definitions there are two types of uni-directional connections, a **uni-directional deposit connection** and a **uni-directional withdrawal connection**. A bi-directional connection is a single path that can both deposit and withdraw information but not both at the same instance in time. For a connection being used for a deposit, the module that acquires the information is defined to be the **receiving module** and the module transmitting the information is defined to be the **sending module**; it is important to note that the module that has

control is not necessarily the sending module. For a connection being used for a withdrawal, the module from which information is being withdrawn is defined to be the **sending module** and the module that acquires the information is defined to be the **receiving module**. With respect to the above example, there are two connections between the three packages:

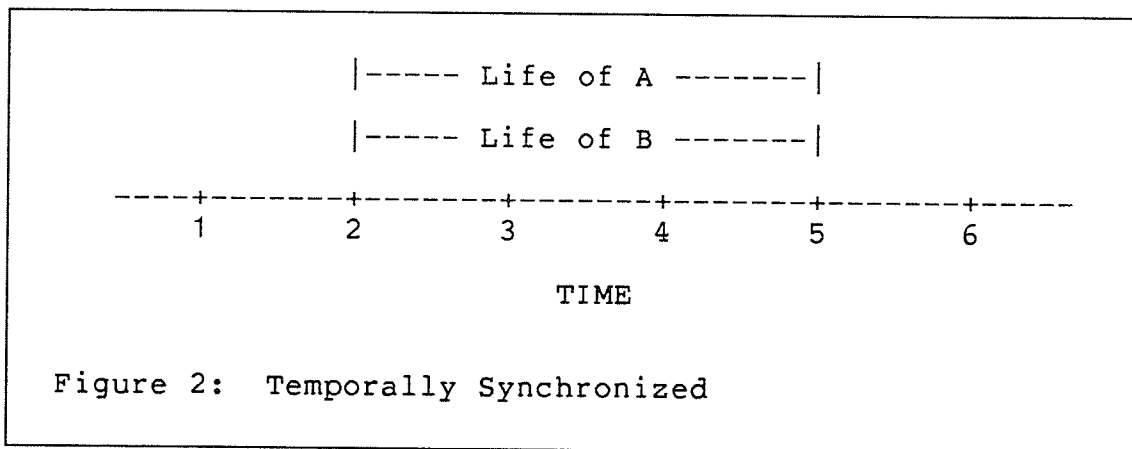
- Between S1 and P3. In this case P3 is the receiving module and S1 is the sending module.
- Between S2 and P3. In this case P3 is the sending module and S2 is the receiving module.

Notice that the terms "sending" and "receiving" are not necessarily with reference to the module that has control, but are with respect to the data transfer itself.

2.5 TIMING

The timing aspects of a computer program are most often associated with its executable elements. This is because data is thought of as passive and that actions personify the data. Any set of actions are meaningless unless the objects that are being acted upon are described fully. In a computer program these objects are data, which are usually given secondary consideration to control. In the majority of the commonly used programming languages the creation and destruction of variable instances occur as a side effect to entering or exiting a program scope; the "block-structured"

languages were intentionally designed in this way. The obvious exception to this rule is that of the nodes of a linked list which are usually created on demand rather than in accordance with program scope. The timing aspects associated with data are equally as important as those associated with control and shall be highlighted in this

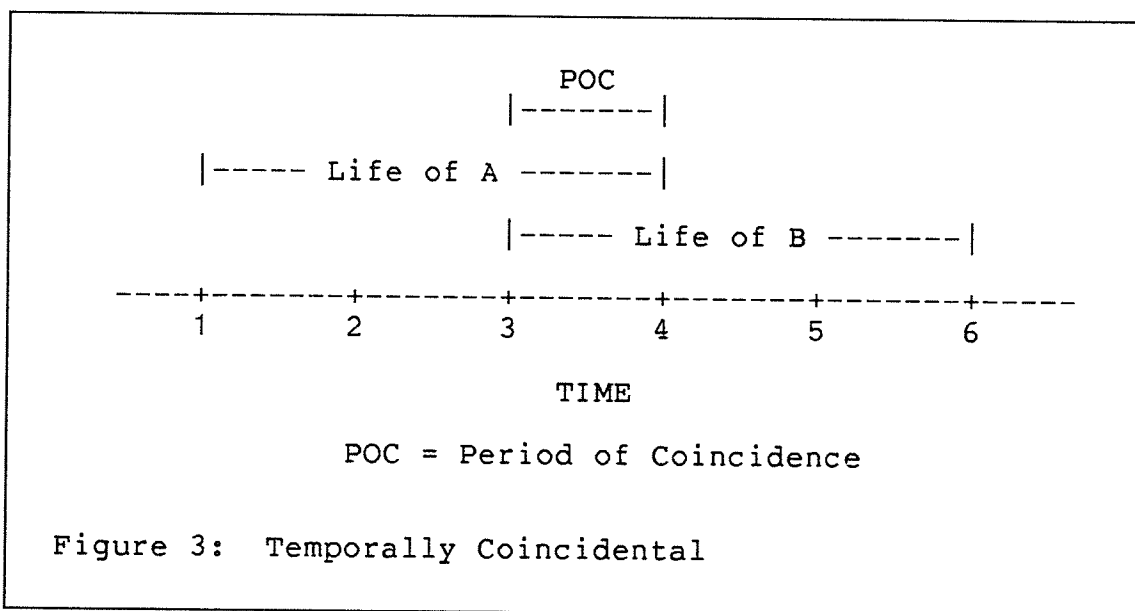


section.

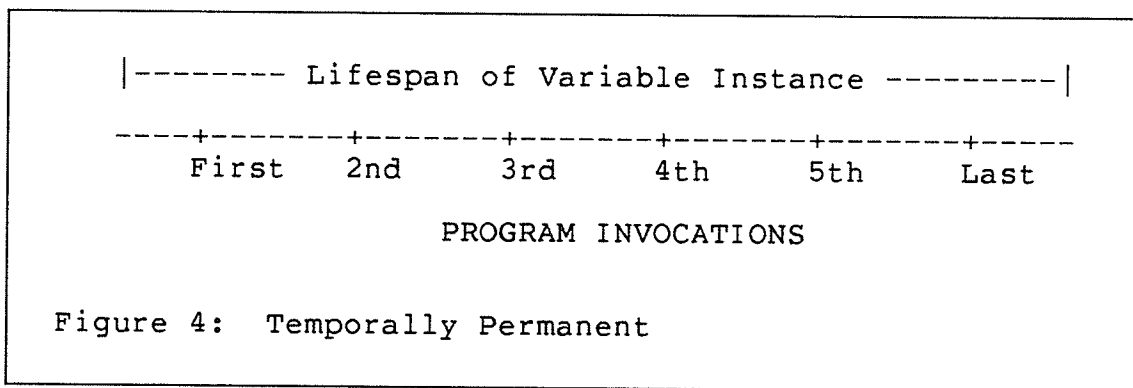
The point in time at which a particular instance of a variable is created is defined to be the **birth** of that instance. When an instance of a variable is born two things happen:

1. Space for the variable is allocated.
2. A memory address is associated with the variable name.

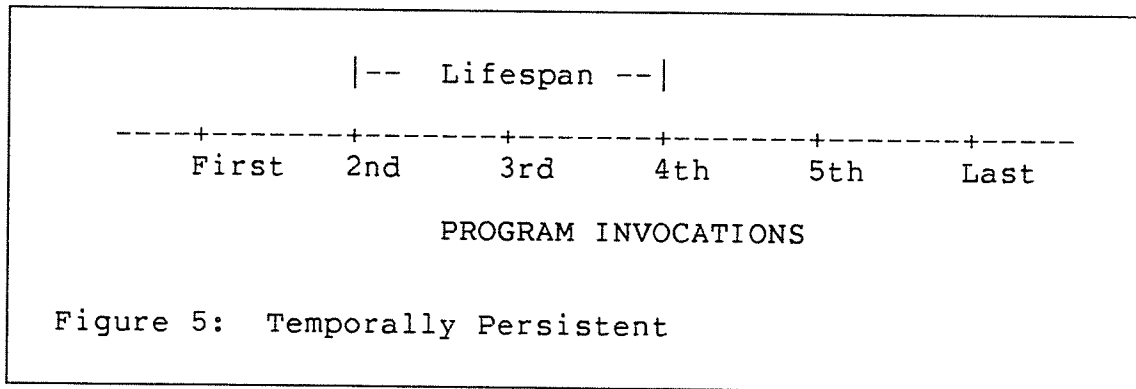
Note that the birth of a variable instance is a separate issue from the visibility of the variable instance, which is discussed below. The point in time at which a particular instance of a variable is destroyed is defined to be the **death** of that instance. When a variable instance dies the space associated with the variable is released. The **lifespan** of a variable instance is the amount of time between the instance's birth and it's death. The instances of two or more variables that are born at the same fixed but otherwise arbitrary time T and die together at the same fixed but otherwise arbitrary time $T+N$ ($N>0$) are defined to be **temporally synchronized** (see figure 2). The instances of two or more variables that are not necessarily temporally synchronized yet coexist for some time interval T , are defined to be **temporally coincidental** during the time



interval T (see figure 3). Some variables of a program, such as those stored on an external file, exist before the execution of a program and persist after the program has terminated. Instances of variables that are created before the virgin execution of a program and are only destroyed after the program is obsolete are defined to be **temporally**



permanent (see figure 4). For example, Pascal constants can be thought of as temporally permanent variables. Normally temporally permanent variables would be stored on a file or some secondary storage device. Instances of variables that are not necessarily temporally permanent yet persist between one or more executions of the program are defined to be **temporally persistent** (see figure 5). Normally temporally persistent variables would be stored on a file or some secondary storage device, but this is not a necessary qualification. Variable instances that are temporally persistent could be static variables depending on the



particular definition of obsolescence. The instance of a static variable is born when the program is loaded into main memory for execution and put to death when the program is ejected from main memory. In the case of a serially-reuseable program, the program may remain memory resident through several executions. In this case the variable instance persists across the invocations in main memory rather than secondary storage. The effect is the same. The static variables of a program that is only executed once and then discarded from main memory are not temporally persistent unless the load module is copied from main memory back to secondary storage at completion (which is unlikely).

2.6 VISIBILITY

The rules that restrict when a variable instance becomes accessible to an executable element for a particular programming language are referred to as the **scope rules** of the language. In this thesis these same rules will be referred to as **visibility rules** to emphasize the difference

between lifespan (which was discussed in section 2.5) and accessibility. In many cases the accessibility of a variable instance is directly associated with the entry and exit of a specific range of executable elements.

Let $\{I:J\}$ be the set of lexically contiguous executable elements bounded by and including a fixed but otherwise arbitrary executable element I and a fixed but otherwise arbitrary executable element J . The notation $\{I:J\}$ is called the **executable range** $I-J$. Given the above definition, I could be the same executable element as J , which would mean that the executable range consisted of one executable element, which is permissible. As a shorthand the executable range of the module M shall be written as $\{M\}$.

Given the above definition the necessary visibility rules can now be defined. A set of variable instances are defined to be **visibly synchronized** if whenever one of the variable instances becomes visible to an executable element then all of the variable instances of the set are visible as well. For example, when one variable in a FORTRAN common block is visible to a subroutine, then all of the variables of the common block are visible as well; in this sense the variables are visibly synchronized. The set of all variable instances that become simultaneously visible at the beginning of $\{I:J\}$, remain visible throughout $\{I:J\}$, become simultaneously invisible at the end of $\{I:J\}$, and are

invisible outside of $\{I:J\}$ are defined to be **visibly contained** within $\{I:J\}$. Two variable instances that are not visibly synchronized in general yet are both visible to $\{I:J\}$ are defined to be **visibly coincidental** with respect to $\{I:J\}$. A set of variable instances that are both temporally synchronized with respect to $\{I:J\}$ and visibly contained within $\{I:J\}$ are defined to be **completely synchronized** with respect to $\{I:J\}$.

The majority of data types provided by programming languages have their lifespans and visibility synchronized in such a way as to prevent variable instances from being visible either before their birth or after their death. This is true for most data types supported by programming languages but there are exceptions. Consider a variable instance that is based upon some pointer reference. In most languages there is nothing to stop the programmer from referencing an instance that has not been born, nor is there any mechanism to prevent such an instance from being referenced after its death. A variable instance that is visible either before its birth or after its death is defined to be **improperly synchronized**. A variable instance that is constrained in such a way as to be visible during the lifespan of the instance only is defined to be **properly synchronized**.

2.7 USAGE OF INFORMATION

The **usage** of information refers to whether the information is being used as data or for control. Most programming languages constrain the use of information as control or data via the type mechanism (assembly languages are an exception, because in general names are not typed). For example, two procedure names could not be added together in Pascal because the compiler will not permit it. There are three typical usages of information:

- **Data Information.**

A variable name refers to location in memory containing a value.

- **Control Information.**

A procedure or entry name stores the address of executable code.

- **Hybrid Information.**

Information that is declared as data and transmitted like data but is used to exchange control between modules. For example, flags, opcodes, and return codes. Hybrid data tends to create higher inter module dependencies than either data or control.

The interfaces and complexities for data information and hybrid information are the same and will be discussed together below. Since data and hybrid information are conveyed in the same way, they will often be discussed together in this dissertation. To distinguish control

information in general from hybrid information it will often be referred to as **pure control information**. The interfaces and complexities for pure control information are different from that of data information and will be discussed separately from data and hybrid.

2.8 DATA INTERFACES

In this section of the thesis, the interface mechanisms for data connections are discussed. The **interface** of a module is defined to be the set of all connections which are not satisfied internally. The **data interface** of a module is defined to be the set of all data connections which are not satisfied internally. A **formal interface mechanism** is a programming language construct provided by the host language that explicitly declares which connections are being satisfied externally. For example, the formal parameter list in Pascal is an instance of a formal interface. A formal interface is defined to be a **complete formal data interface mechanism** if each connection described in the interface is detailed with at least the following information:

- The name of the object being accessed via the connection.
- The data type of the object being accessed via the connection.

- The direction that the information is expected to flow through the connection.
- The complexity of the information passing through the connection. The complexity is part of the data type in some programming languages. Data complexity will be discussed in section 2.11.

An example of a complete formal data interface would be the formal parameter lists in Ada⁴. Many programming languages which support a formal interface mechanism do not support a complete formal interface mechanism. Such languages are defined to have an **incomplete formal data interface mechanism**. An example of an incomplete formal interface is that provided by FORTRAN. The formal interface provided by FORTRAN does not require the user to specify the type, direction, or complexity of the information passing through the connection.

The interface of a module may consist of a formal interface plus some other connections not made through the formal interface. All data connections that are not members of the formal interface are defined to be members of the **informal data interface mechanism**. By definition all informal interfaces are incomplete. A module is defined to have an **ideal data interface** if all data connections are members of the formal interface and the informal interface is null. A module that has all data connections through the

⁴ Ada is a registered trademark of the United States Department of Defence, Ada joint programme office.

informal interface and none through a formal interface is defined to have a **pathological data interface**. Often a module will have some data connections through the formal interface and some data connections through the informal interface. Modules that have a mixture of data connections of this sort are defined to have a **composite data interface**.

For the purpose of this thesis the module interface is defined such that all external connections pass through the interface. The generic term interface is used to refer to the formal and informal interfaces collectively. It should be noted that the notion of an "interface" presented above is different from the concept described by Yourdon and Constantine [65]. The description above has one interface per module through which all external connections must pass. Yourdon and Constantine's modules had many interfaces, one per external connection.

In summary, there are several different interface mechanisms:

1. Formal Interface.

- a) Complete Formal Interface.

- b) Incomplete Formal Interface.

If all connections run through the formal interface the module is said to be ideally connected.

2. Informal Interface.

If all connections run through the informal interface the module is said to be pathologically connected.

3. Composite Interface.

Modules connected in this way are defined to be compositely connected.

2.9 CONTROL INTERFACES

The **control interface** for a module has different requirements than the data interface for a module. Recall that the data interface for a module contained all data usage and hybrid usage connections emanating from the module. For data interfaces it is desirable to know where the connection is running from and where the connection is running to. This means that data connections that run through the formal interface are superior to data connections that do not run through the formal interface.

The relationship between desirable and undesirable control interfaces is inverse to that of the relationship between desirable and undesirable data interfaces. This is because the origin and destination of a global control connection are implicit, whereas the origin and destination of a formalized control connection are not. First, consider the following statement that makes a global data reference:

A := A + GLOBAL

The source and destination of the connection for GLOBAL are not obvious from the above statement. Second, consider the following statement that makes a global control reference:

```
CALL PROC(P1,P2,P3);
```

The control connection runs from the calling procedure to procedure PROC. The source and the destination of the global control connection above are both obvious from examining the statement. Third, consider a control connection that runs through the formal parameter mechanism as follows:

```
CALL PLOT(X,SIN);  
:  
PROCEDURE PLOT(REAL P; PROCEDURE TOOL);  
:  
CALL TOOL(P)  
:  
END PROCEDURE PLOT
```

It is not obvious by examining the invocation of procedure TOOL that the procedure SIN is being invoked. Formalized control connections of this sort tend to increase program complexity rather than decrease it.

Therefore, a control connection running through the formal interface of a module is disadvantageous; in fact such practices obscure the true origin of the connection. Therefore it is expected that normal control connections will run through the informal interface instead of the formal interface. This is discussed further in chapter 4.

For the purpose of this thesis it is assumed that only structured entry/exit [65] pure control connections exist in the system. This means that there are no GO TO's, abnormal returns, or any other undisciplined transfer of control within the system. For the purposes of interrupts, the software system provides interrupt handlers which themselves have structured control conventions.

2.10 AGGREGATES

An **aggregate** (or a **data aggregate**) is a set of variable instances that are temporally synchronized in general and visibly synchronized with respect to any fixed but otherwise arbitrary executable element. The elements of an aggregate are disjoint heterogeneous data items each of which may be of varying complexity. They are disjoint because the physical or logical placement of the item in the aggregate is inconsequential (according to the definition). Essentially, the elements of an aggregate are independent variables that have identical lifespans, and whenever and wherever they become visible they do so as a group. Often an aggregate can be named, but it is not a necessary qualification. There are several examples of a data aggregate:

- A FORTRAN common block. FORTRAN is one of the few traditional programming languages that supports a construct that allows the user to explicitly declare a data aggregate.

- A package that contains only data, where when imported, the entire package is visible. That is to say, that this does not include the case where select items may be imported from a package.
- A record (a la Pascal, COBOL, etc.). It is worthwhile to emphasize that while a record may be an aggregate, an aggregate is not a record. The fundamental difference between some records and an aggregate is the placement of the data elements. In some records the physical ordering of the data items is paramount. For example, the carriage control on an output record in a COBOL program must be the first field of the record. There is no such ordering qualification for an aggregate. In general the elements of a record can be expected to be more cohesive than the elements of an aggregate. This actually highlights why a FORTRAN common block is more likely an aggregate than a record. In a common block the elements are grouped for their lifespan and visibility whereas the elements of a record are grouped together because of their logical relationship.
- The stack based variables that are created and become visible upon entry into a procedure of a Pascal program. Such variables are visibly contained within the host procedure and are visibly synchronized from the viewpoint of any embedded procedures. Note that aggregates of this type are not explicitly named

(except perhaps if one considers the procedure name to be the aggregate name as well) and as such are normally not thought of by the programmer as a collective entity.

Although the elements of a data aggregate are independent variables that could be used for totally different purposes, they have in common their lifespan and their visibility.

2.11 COMPLEXITY OF INFORMATION

Each connection, be it a data connection, a hybrid connection, or a pure control connection, can carry information of varying complexity. Ultimately all information is stored and exchanged in the form of data; it is a function of the program whether the information is to be interpreted as data or control.

For example, a given word of memory might contain either a floating point value or the address of a subprogram to be invoked. While the first case is an example of data information and the second case is an example of control information, both elements are stored and manipulated as data in memory. This is true of all classic Von Neumann architectures and this thesis will restrict discussion to these types of computers.

There is precedent in other scientific and engineering disciplines for having connections convey information of

varying complexity. In electronics for example, a connection is represented either by a wire or some other medium that carries intelligence. The wire may carry a simple current such as the B+ (power) or it may carry something as complex as a time division multiplexed signal. According to the Webster's New Collegiate Dictionary [56] complex is defined to be "composed of two or more parts; not simple". Therefore the information exchanged over a connection may either be **simple** or **complex**. Simple means that only one component can be identified and complex means that more than one distinct component can be identified.

It must be noted that an item's complexity is a separate issue from that of the item's data type or usage. While one may argue that a floating point item is more complex than an integer item, both are considered to be simple for the discussion in this thesis. Essentially, complexity refers to both the amount of information expressed in a connection and the human capacity for dealing with the data in the manner that it is organized.

2.11.1 Myers' Complexity Attributes

Myers made two definitions with respect to the complexity of information in [43], [45]. Myers used these two definitions when describing different levels of module coupling. These two levels will be used in chapter 4 to describe Myers' coupling classifications. This thesis defines new levels of

complexity that differ from Myers' levels of information complexity. The levels of information complexity introduced in this thesis will be described in a later section. Myers' levels are described immediately below.

2.11.1.1 Homogeneous Complexity

Myers defined that an item had **homogeneous complexity** if it was either a scalar or an array.

2.11.1.2 Heterogeneous Complexity

Myers defined that a data structure such as a Pascal record had **heterogeneous complexity** (even though the record may be homogeneous). In essence, Myers based the item's complexity on the syntax used to declare the item.

2.11.2 Atomic Complexity

Rather than using the terms "simple" or "scalar" to describe elementary data items, this thesis shall use the term **atomic** because of the analogy to nuclear physics. The atom, while considered to be one of the "elements", can be further broken down into other particles that do not have identifiable physical properties. This is very similar to a floating point variable that is composed of a mantissa and a characteristic, which are in turn composed of sequences of bits. While the mantissa and the characteristic can be identified by an intelligent observer, they are of little use unless they are used in conjunction with each other. It

is for this reason that the term "atomic" shall be used to refer to elementary data items in this thesis.

An item that is atomic is by the definition given above an item that cannot be separated into one or more useable components; it is an intrinsic unit. An example of an atomic datum is a scalar variable of any base data type. In other words, if the host programming language supports a data type's declaration and its intrinsic operations, then it is an atomic data item providing that no meaningful operation can be performed on any one of the individual components. Using this scheme, a PL/I FIXED BINARY(31) variable [29] would be considered to be an atomic item whereas a CHAR(10) VARYING string variable would not. This is because there are mechanisms whereby operations can be performed on the length part or on any contiguous subset of the individual characters. One of the objectives of the newer programming languages such as Ada is to provide enough of an abstraction mechanism in the base language to allow the user to extend the language by adding new types and operations. These new types appear to be intrinsic, even though they are not, and are thought of by some programmers as atomic types.

2.11.3 Attributes of Information Complexity

Myers defined the complexity of a data item based on the programming language construct that was used to declare it. In this thesis, the complexity of a data item is based on the following four attributes of the data:

1. The number of atomics in the data item.
2. The number of different data types in the data item.
3. The dependence of order in the data item.
4. The presence of structure information in the data item, over and above the atomic items themselves.

2.11.3.1 Number of Atomics

As the number of atomic items in a data item increases the complexity of the data item increases. For the purposes of illustration, this thesis will count the number of atomics in a data item as either one or many.

2.11.3.2 Number of Data Types

Some data items contain atomics of only one type whereas other data items contain atomics of many data types. As the number of data types in a data item increases the complexity of the item increases. For example, consider the following data items:

```
DECLARE DS1(2)    INTEGER;
```

```
DECLARE 1 DS2,  
        2 F1    INTEGER,  
        2 F2    INTEGER;
```

```

DECLARE 1 DS3,
        2 F1    REAL,
        2 F2    INTEGER;

```

In terms of the number of data types, DS1 and DS2 exhibit the same complexity. Data structure DS3 exhibits a higher complexity than either of DS1 or DS2 in terms of the number of data types. For the purpose of illustration in this thesis, the number of data types in a data item will be counted as either one or many.

2.11.3.3 Order Dependency

In some data items the order of the atomics is important, and in others it is inconsequential. For example, consider the following record:

```

DECLARE 1 OUTPUT_RECORD,
        2 CARRIAGE_CONTROL CHAR(1),
        2 OTHER_DATA      CHAR(132);

```

In the above record the field CARRIAGE_CONTROL must appear as the first field in the record. It cannot be arbitrarily moved about within the record. On the other hand, consider the item COMPLEX defined below:

```

DECLARE 1 COMPLEX,
        2 CREAL    REAL,
        2 CIMAG    REAL;

```

In the data structure COMPLEX, the fields within the record can be moved about without consequence. In this thesis, this is referred to as **order dependency**. The greater the

number of dependencies, the greater the complexity of the data item. For the purposes of illustration in this thesis, the number of dependencies will either be counted as none or some.

2.11.3.4 Presence of Structure Information

In some data items there is information present over and above the actual data stored in the atomics. For example, consider the following data structure:

```
DECLARE 1 WEEK1,  
        2 DAY1 INTEGER,  
        2 DAY2 INTEGER,  
        2 DAY3 INTEGER;
```

The above data structure contains three atomic items, and so does the following data structure:

```
DECLARE 1 WEEK2,  
        2 WEEKDAY,  
        3 DAY1 INTEGER,  
        3 DAY2 INTEGER,  
        2 WEEKEND,  
        3 DAY3 INTEGER;
```

Data structures WEEK1 and WEEK2 both contain three atomic items, and have the same number of types and order dependencies. It is clear that WEEK2 conveys information over and above the information conveyed by WEEK1. This type of information is called **structure information** in this thesis. As the amount of structure information increases, the complexity of the data item increases. For the purpose of illustration in this thesis, the structure information will either be counted as none or some.

2.11.3.5 Ranking The Attributes

The attributes that affect the complexity of a data item

DATA ITEM COMPLEXITY	ATTRIBUTES			
	ATOMICS	TYPES	ORDER	STRUCT
Atomic	one	one	no	no
Aggregate(1)	many	one	no	no
Aggregate(2)	many	many	no	no
Linear	many	one	yes	no
Molecular(1)	many	many	yes	no
Molecular(2)	many	one	yes	yes
Molecular(3)	many	many	yes	yes

Figure 6: Attributes of Information Complexity

have been ranked and placed in figure 6. In figure 6 the labels have the following meaning:

- ATOMICS

This column represents the number of atomic data items found in the data item being evaluated.

- TYPES

This column represents the number of different data types present in the data item being evaluated.

- ORDER

This column represents whether or not there are any order dependencies in the data item being evaluated.

- STRUCT

This column represents whether or not there is structure information present in the data item.

- DATA ITEM COMPLEXITY

This column names the class of data items with a particular set of attributes.

Figure 6 classifies data complexity as falling into one of four different classes:

1. Atomic.

Atomic complexity was discussed above.

2. Aggregate.

Aggregate complexity was discussed in general above. The names aggregate(1) and aggregate(2) refer to two different levels of aggregate complexity. These two different levels of aggregate complexity will be referred to collectively as "aggregate".

3. Linear.

Linear complexity will be discussed below.

4. Molecular.

There are three different levels of molecular complexity, molecular(1), molecular(2), and molecular(3). They are collectively referred to as "molecular". Molecular complexity will be discussed below.

2.11.4 Linear Complexity

A data item that exhibits **linear complexity** has:

- More than one atomic item.
- Only one data type.
- The item has order dependency.
- The item has no structure information.

Given that definition, the following two declarations would both represent items having linear complexity:

```
DECLARE ARRAY(3) INTEGER;

DECLARE 1 RECORD,
        2 F1    INTEGER,
        2 F2    INTEGER,
        2 F3    INTEGER;
```

The name "linear" was chosen to represent data items having the complexity attributes mentioned above because typically such items are implemented with arrays. Linear complexity is not limited to arrays, as the example above demonstrates.

2.11.5 Molecular Complexity

Figure 6 defines three different levels of **molecular complexity**. A data item exhibits molecular complexity if:

- There is more than one atomic in the item.
- There is order dependency in the item.
- There is either structure information present in the item, or there is no structure information present and there is more than one data type in the item.

The following three data structures are examples of molecular(1), molecular(2), and molecular(3) complexity respectively:

```
DECLARE 1 MOLECULAR1,  
        2 F1      REAL,  
        2 F2      INTEGER;
```

```
DECLARE 1 MOLECULAR2,  
        2 EXTRA,  
        3 F1  INTEGER,  
        3 F2  INTEGER;
```

```
DECLARE 1 MOLECULAR3,  
        2 EXTRA,  
        3 F1  REAL,  
        3 F2  INTEGER;
```

The name "molecular" was chosen to identify data items having the above mentioned attributes because of the name's original meaning in physics. The molecule is composed of other atomic particles and has order and structure.

2.11.6 Abstract Complexity

A data item is defined in this thesis to have a complexity of **abstract** if the actual data item to be operated on must be obtained via an additional memory reference. In particular, any pointer variable is abstract since the information transmitted is not the data itself but an indirect reference to the data. In [43], [45] Myers states that the coupling class of a pointer variable is the class that would be assigned to the item the pointer ultimately points to. This is not always a simple problem since the

pointer may point to a scalar, a record, or a recursive data structure such as a binary tree or a circular linked list. As well, a pointer variable may point to different items at different instances in time. Linked lists may point to other linked lists and so on. A pointer variable has complexity "abstract" since the number of distinct components to the information cannot be easily determined in all cases. In human terms, an extra level of complexity is added to the item as soon as a pointer reference is introduced.

2.11.7 Data Complexity Summary

In this thesis, a data item is classified as having one of the following levels of complexity, arranged from least complex to most complex:

1. Atomic.
2. Aggregate.
3. Linear.
4. Molecular.
5. Abstract.

2.12 CONCLUSION

2.12.1 Summary

This chapter has clarified some important topics, namely:

- The definition of a module and modular design.

- The concept of representing external references as connections.
- The concept of a module interface and the associated data interfaces and control interfaces.
- The concepts for variable timing and visibility have been emphasized as separate issues.
- The identification and ordering of the complexity of data items.

2.12.2 Analysis

Some of the ideas presented in this chapter are new while others are a tidying up of some state of the art concepts. The terminology developed in this chapter for dealing with the design concepts in question is both consistent and unambiguous. This allows design ideas to be communicated in a structured way.

Chapter III

COHESION

"There is a rule of sorts in government, or ought to be, that you never do anything about a problem until you learn to measure it. Paul F. Lazarsfeld, the great Columbia University professor who developed the art of polling, used to tell his classes that the moment you attach a number to an idea you have learned something. You know more than you knew before. The number seems too large, too small, about right. You are already thinking in a more precise way."

- Senator Daniel Patrick Moynihan

3.1 ABSTRACT

Ever since software engineers have recognized the need to explicitly design computer software, there has not been a general technique for evaluating prospective designs in a scientific manner. One of several qualitative design metrics that has been established is "cohesion", also known as "module strength". In this chapter, the established cohesion metrics are enumerated and summarized. The factors that influence cohesion, which were previously undetermined, are asserted. The level of cohesion known as "logical cohesion" is decomposed into two distinct and discontinuous levels of cohesion called "logical data cohesion" and "logical function cohesion". The qualitative factors of cohesion are identified and decomposed into five distinct

parts, each of which is easier to conceptualize than the whole. A numerical representation of a module's cohesion called the module cohesion coefficient is developed. With the assertion of its influencing factors and the introduction of a quantitative metric, cohesion becomes a more precise design aid that can be utilized by a wider range of people.

3.2 INTRODUCTION

3.2.1 Background

Much of the work in the area of software engineering has been an attempt to reduce program complexity; reducing a program's complexity has a positive effect on the program's quality and cost [44]. One of the first methodologies introduced with this thought in mind was structured programming [17]. The basic premise of structured programming was that the flow of control is altered in only a finite number of ways, enforced by the use of a set of control structures. At the time it was introduced, structured programming was very controversial. There were many arguments as to its effectiveness, pro [16],[62] and con [27]; there were also arguments as to which control structures were structured and which ones were not. Eventually it was reasoned that it was not the control structures themselves that were important, but the manner in which the programmer thought of and expressed program logic [33],[18].

Structured programming by itself did not provide a complete solution to the program complexity problem. It was found that very large programs could easily become complex even if they were written using structured control constructs. The next step in the reduction of program complexity was "modular" programming which has come to be known by the term "structured design" [65]. (Myers [43], [45] referred to the same discipline as "composite design" but the term never caught on.)

3.2.2 Structured Design

Using structured design, the complexity of a program is reduced by decomposing the original program into a number of distinct modules with each module performing one particular aspect of the original program's function. Individual modules are smaller than the original program and therefore easier to plan and maintain. The concept of structured design is simple: we use functional decomposition and successive refinement. There are many possible design outcomes given the same functional specifications. When comparing two possible designs for the same program, it is difficult to measure the benefits of one design over another. By definition, a well designed program requires less man effort to change than a poorly designed program. Different changes will require different amounts of effort. Changes always come in the future and are difficult to anticipate; unfortunately no one has a crystal ball that

works with any degree of accuracy. A particular design's worthiness can only be proven by the test of time. Because of all the tradeoffs involved, there are good designs and bad designs but never right designs and wrong designs.

3.2.3 Design Metrics

Using experience as a guide, several qualitative metrics for measuring the maintainability of software designs have been established [30],[65],[43],[45], [44]. Probably the most effective pair of qualitative metrics that were devised are coupling and cohesion [43], [45]. Coupling measures the way that modules interact with other modules, whereas cohesion measures how elements of an individual module interact. In this chapter, cohesion is examined in detail and is made more effective by the exposure of its underlying factors.

3.3 REVIEW OF COHESION

Module cohesion, also known as module strength, is a measure of how the elements internal to a module are related. In effect, cohesion measures to what degree a collection of program elements deserves to be a module. This is not always easy to do because a module is a logical entity, not a physical entity like a procedure or a compilation unit. The logical boundaries of a module do not necessarily correspond to the physical boundaries provided by a programming language, although it would be beneficial if they did. The guidelines for a group of statements to

qualify as a module is elusive; the classification of modules is therefore enigmatic. Together with coupling [43], [45] and several other factors, cohesion provides a qualitative metric for software designs.

The higher the degree of dependence between elements of the same module, the higher the degree of cohesion. The goal is to obtain a module that contains program elements that are highly interdependent (i.e. "tightly bound"). Thus, high cohesion is desirable. Low cohesion, which is undesirable, indicates that the module in question is likely to be difficult to maintain. Myers [43], [45] defined cohesion as falling into one of seven distinct categories listed below from least desirable to most desirable:

1. Coincidental cohesion.
2. Logical Cohesion.
3. Classical cohesion.
4. Procedural cohesion.
5. Communicational cohesion.
6. Informational cohesion.
7. Functional cohesion.

Yourdon and Constantine [65] also arranged cohesion into seven categories although some of the categories are different. Their categories are listed below in the order of least desirable to most desirable:

1. Coincidental cohesion.
2. Logical Cohesion.
3. Temporal cohesion.
4. Procedural cohesion.
5. Communicational cohesion.
6. Sequential cohesion.
7. Functional cohesion.

The differences between the above two rankings and their relative advantages and disadvantages will be discussed in the remainder of this chapter.

Although both Myers' and Yourdon and Constantine's orderings imply a relative ranking, the actual distance between each level is not linear. For example, the distance between coincidental cohesion and logical cohesion is not necessarily the same as the distance between procedural cohesion and communicational cohesion, and so on. Both Myers and Yourdon and Constantine have arbitrarily associated numbers with each of the above levels that quantify the distance between each level. The exact distances between each level were chosen by intuition rather than by scientific research.

A given module may exhibit one or more of the levels of cohesion enumerated above. The accepted rule is that when a module exhibits more than one level of cohesion it is classified as exhibiting the lowest ranking level of the

ones that it possesses. Since cohesion measures the quality of a module and an exact numerical value cannot be placed on its quality, it makes sense that a "weakest link in the chain" rule would apply.

3.3.1 Coincidental Cohesion

Coincidental cohesion is the absence of any (but chance) relationships between elements of the same module. Cohesion is a measure of the relationships between elements of the same module; if there are no relationships then there is no cohesion. This is the definition of coincidental cohesion. This level of cohesion is necessary to complete the enumeration but is not very interesting otherwise.

Modules that exhibit coincidental cohesion are rare since there is usually some justification for the creation of a module. Coincidentally cohesive modules may result from the modularization of an existing non-modular program. For example, one might modularize a 1000 statement program into ten modules of 100 statements each. The first 100 statements could be the elements of module A, the second 100 statements the elements of module B, etcetera. The flow of control would start with module A. At the conclusion of module A, module B would be invoked, and so on. The justification for each module in this case is the physical location of the statements in the original program and the desire to have each module a fixed but otherwise arbitrary

size. The results would have been no different in terms of cohesion if the module size was 150 statements instead of 100 statements. In both cases the relationships between the modules are not of a logical or functional nature, they are of a lexical nature. Coincidentally cohesive modules exhibit the lowest and therefore the most undesirable level of cohesion.

3.3.2 Logical Cohesion

Logical cohesion implies that elements of a module are related because they perform one function on a number of different data items or because they perform a number of different functions on one data item. That is to say, that the module groups together a class of operations or a class of data items. For example, consider a module that performs all input/output for a given physical device. The module performs more than one logical function. In this case the module performs two functions on one entity, input and output for the given device. The desired function would have to be passed down to the module as a parameter. Another example would be a module that performs output to a number of different device types. The type of device would have to be passed down to the module as a parameter. In both cases the exact function of the module is determined by the state of the parameters to the module. Whether the elements of the module are related because of function or because of the entity that it operates on, the control of

the module is divided into many parts either with a case like structure or a series of decisions. The module then can be thought of as a collection of smaller modules that are grouped together because of their logical relationship.

3.3.3 Temporal Cohesion

Yourdon and Constantine's temporal cohesion is equivalent to Myers classical cohesion except they are named differently. It is this author's opinion that temporal cohesion is a more appropriate name and thus will be used in this chapter.

Elements of a temporally cohesive module perform different functions but are grouped together because the functions are all performed at the same time. A common example of a temporally cohesive module would be one that performs all the initialization for a program. Although the initialization may require many operations that are totally unrelated in the sense of the objects that they act on, the operations are grouped together because the operations are performed at the same time.

The name temporal comes from the base word "time". Myers named this classification classical cohesion because he found that it occurred often in programs. Therefore, Myers named the classification after how it was typically used whereas Yourdon and Constantine named it after its functional properties.

Temporal cohesion differs from logical cohesion in that while both the elements of logically cohesive and temporally cohesive modules perform unrelated functions, elements of a temporally cohesive module performs the functions at the same time. This is not true of logically cohesive modules. For this reason temporal cohesion rates higher on the relative scale than logical cohesion.

3.3.4 Procedural Cohesion

Elements of procedurally cohesive modules are grouped together because they are identified as solving a specific problem. The algorithm to solve a problem may be composed of a number of discrete logical steps, but when executed in a specific manner the steps solve the specified problem. The type of statements found in a procedurally cohesive module may be iteration, decision, or a linear sequence of steps. Procedural cohesion rates higher than all the previously discussed types of cohesion and begins to be an acceptable level of cohesion for a module to possess.

3.3.5 Communicational Cohesion

Elements of a communicationally cohesive module are grouped together because they exhibit procedural cohesion and they operate on the same data elements. For example, consider the detail line of a report that contains many data elements. In this case let us suppose that the fields were not related to each other in terms of how they are to be

computed, that is to say, that the information used to compute each field is acquired from independent sources and not from some other field of the detail line. A module that computed all the fields of such a detail line would be communicationally cohesive.

Myers grouped communicational and sequential cohesion (discussed below) together as communicational cohesion. The definition of communicational cohesion given above is from Yourdon and Constantine's classifications. The differences between the two are discussed below in the explanation of sequential cohesion.

3.3.6 Sequential Cohesion

Elements of a sequentially cohesive module are related because the output from one statement is the input to the next statement. That is to say, that the statements all work on the same data elements and the statements must be executed in a specific order. For example, consider a report detail line where each successive field in the detail depended upon the value of the previous field. Once the first field was calculated the second field could be calculated; once the second field was calculated the third field could be calculated, and so on. Part of the input for the calculation of any one of the fields would be one or more of the previous fields. A module that computed all the fields of the detail line in this manner would be sequentially cohesive.

Sequential cohesion is a member of Yourdon and Constantine's cohesion classifications but it is not a member of Myer's classifications. This is because Myers considered sequential cohesion to be not sufficiently different from communicational cohesion to warrant its own classification. The difference between the two is that a communicationally cohesive module has elements that can be executed in any order, while the elements of a sequentially cohesive module must be executed in a specific order. It is this author's opinion that whenever a distinct difference can be pointed out between module types, then this in itself is justification for a new cohesion class. For this reason this author believes that communicational and sequential cohesion each warrant their own classifications.

3.3.7 Functional Cohesion

Functional cohesion represents the highest attainable level of cohesion possible on the relative scale. A functionally cohesive module is a module that performs one well-defined function on one data element. A module can be determined to be functionally cohesive if it does not exhibit any of the other forms of cohesion discussed above. Computer implementations of mathematical functions usually display functional cohesion. For example, computer implementations of the trigonometric functions such as sine and cosine are usually functionally cohesive.

3.3.8 Informational Cohesion

Informational cohesion is a member of Myers' cohesion classifications but it is not a member of Yourdon and Constantine's classifications. Myers rated informational cohesion second only to functional cohesion and claimed that functional cohesion was only marginally superior; although Myers ranked informational cohesion lower than functional cohesion on the relative scale, he gave them the same numerical rating on his absolute scale. Clearly, Myers thought of informationally cohesive modules as near ideal. Informational cohesion turns out to be very important to the thrust of this chapter and as such will be discussed at some length.

An informationally cohesive module performs multiple functions on one data element where each function is referenced with an entry point to the module. In essence, informationally cohesive modules are the same as logically cohesive modules except that instead of passing down an operation code (logical cohesion), the entry point to perform the desired function can be invoked directly by the caller (informational cohesion).

There are a number of questions that surface when considering informational cohesion. Does such a module deserve to have its own cohesion classification? Does the classification have more to do with coupling than cohesion? This depends on whether one regards a module with multiple

entry points as one module or many; recall that cohesion relates the elements internal to a single module while coupling measures the relationships between separate modules. If such an entity is considered to be a number of separate modules, then the problem falls into the realm of coupling instead of cohesion. These and other questions were left unanswered by Myers in [43], [45], [44] and were not addressed by Yourdon and Constantine in [65]. The answers to these questions will be discussed in the remainder of this chapter.

3.4 THE FACTORS THAT INFLUENCE COHESION

In the past, several authors have established different numbers of cohesion classifications, each with their own set of designations. Each level of cohesion is distinct because each is discernible from the others for qualitative reasons. In reality, each author developed a series of classifications based on intuition and then presented arguments to justify each class being distinct. The known levels of cohesion have been "proven" with a combination of practical experience, theoretical argument, and a little hand-waving [65].

The basis for cohesion classifications has evolved differently than the basis for coupling classifications. With coupling, a list of the factors that influence coupling was determined [65] and then the classifications were

derived from that. This is the way that one would expect that the scientific process would lead to the answer; first identify the individual components and then form the broader general classifications. Unfortunately it did not transpire in the order outlined. Up until this time, no author has published a list of the factors that influence cohesion, or at least none that were approached rigorously. The reason that the cohesion classifications were not derived in this manner is because the factors that must be identified are extremely elusive and difficult to quantify.

It would be advantageous to identify the factors that influence cohesion for several reasons:

- The identification of the factors that contribute to cohesion would ultimately lead to a better understanding of cohesion, and therefore higher quality software. It may turn out that the factors involved in cohesion are of more utility than the current classification scheme.
- The identification of the factors would allow the reevaluation of the current cohesion classifications in order to obtain classifications that were more representative of module behavior.
- Ultimately, the externalization of the underlying factors of cohesion may allow intelligent software to be written (perhaps as part of a compiler) that could evaluate the cohesion of a module. Currently the

process is so judgmental that only a human (and an intelligent one at that) can perform such an evaluation. Even further in the future, once the nature of software is understood, it will be easier for automated software to be generated.

3.4.1 Internal Aspects of a Module

Cohesion is a measure of how the elements internal to a module are related. Clearly, before the relationships between these elements can be quantified, the elements that are being related should be well understood. There are three types of elements internal to a module, they are:

- a) Functional elements.
- b) Data elements.
- c) Time elements.

These three elements are axioms adopted by the author.

3.4.1.1 Functional Elements

The functional elements of a module refer to the executable statements or groups of executable statements within a module. It is not practical to have all modules in a system containing only one executable statement. This is because such a system would undoubtedly have complicated coupling between the large number of modules. The whole purpose of structured design is to make the system more understandable and hence more maintainable. The other extreme would have

only one module in the system. This simplifies the coupling, because there is none, but complicates the cohesion of the one module involved. The "correct" module size is a trade off between understandability, coupling, and cohesion.

The goal of designing a module is to have all executable statements in the module work together to perform one and only one logical function. This is the definition of ideal cohesion from the perspective of the functional elements; there may be many executable statements in a module but they can be considered to be an indivisible unit that performs one logical operation. Note that the one logical operation may be composed of many intrinsic operations, just as one machine hardware instruction may be composed of many micro-code level instructions.

Any one logical operation in a module may be composed of one or more identifiable steps, each of which shall be referred to as a **segment**. By definition, an ideally cohesive module has only one segment. Lower forms of cohesion may have more than one segment. In the worst form of cohesion, each statement is a segment.

A module should perform one logical operation. Since the real world is not always ideal there will be modules that perform more than one logical operation. For example, a module that reads an input record and validates the fields

contained within it actually performs two logical operations. Some modules will perform many logical operations. Note that each logical function that a module performs may be composed of one or more segments. By definition, a module that performs one logical operation and contains only one segment is ideally cohesive.

3.4.1.2 Data Elements

The data elements of a module refer to both the input and output items that a module operates on. These items will be called **parameters** in this chapter, even if they are not explicit formal parameters to the module.

There are different complexities of data that a module may operate on. By definition, an ideally cohesive module operates on one indivisible (elementary) datum. For example, a module that manipulates a single integer datum would be ideally cohesive from the data perspective. A similar module that adds one to three separate integers exhibits a lower level of cohesion from the data perspective. It should be obvious at this time that there is a relationship between the cohesion exhibited from the data element perspective and the cohesion exhibited from the functional perspective.

A module may operate on many separate elementary data times. Such a module is not as tightly bound as a module that operates on only one elementary datum.

A module may operate on several elementary data items that are combined to form one logical data item. As well, a module may operate on many logical data items, each of which may be composed of one or more elementary data items. An example of a logical data item would be a stack or a queue, each of which is composed of many elementary data items. For a module to have ideal cohesion from the data perspective it must operate on one logical data item that is composed of one elementary data item.

3.4.1.3 Time Elements

The time elements of a module refers to the dependency of functional and data elements with respect to the order that they must be computed. For both the functional elements and the data elements, there are three types of time dependencies (listed below in reverse order of desirability):

- a) The elements are independant. This means that each element (segment, logical operation, data element, etc.) may be moved at random within the module without affecting the outcome. For example, suppose that a module must initialize a series of data items. The data elements may be initialized in any order as long as they are all initialized. In turn, the functional elements that perform the initialization may be executed in any order. Such modules can be

thought of as being executed in "N" logical time units, one time unit per executable element or data element.

- b) The elements are sequentially dependant. This means that the elements must be executed in a specific predetermined order. The elements may be functional segments, logical operations, or a series of data elements that must be computed in a specific order. For example, suppose that a module initializes a series of data elements. The computations are sequentially dependant if some of the data elements must be initialized before other elements for the outcome to be correct. Such modules can be thought of as being executed in "N" logical time units, one time unit for each executable element or data element.
- c) The elements are totally dependant. This classification differs from the previous classification because only one segment, data element, etc. can be identified and therefore the elements are indivisible. Modules that are indivisible can be thought of being executed in one logical time unit as opposed to "N" logical time units as in the previous two classifications.

It should be noted that the functional dependency of a module and the data dependency of a module are tightly related. For example, consider a module that is composed of

a number of initialization statements. When the functional elements are rearranged so is the dependency of the data elements and vice versa. There is therefore a one to one relationship between functional time dependencies and data time dependencies. For this reason such relationships shall be referred to collectively as **time dependencies**.

Another timing aspect that is involved in cohesion is **exclusivism**. A module exhibits exclusivism when

- The module is composed of more than one logical operation and not all of the logical operations are executed on each invocation. This is called **functional exclusion**.
- The module performs operations on more than one logical datum and not all of the data are manipulated on each invocation. This is called **data exclusion**.

Notice that functional exclusion applies to logical operations but not to segments. If functional exclusion applied to segments then most modules would exhibit functional exclusivism. For example, consider a module that performs error checking on the parameters. Such a module would not be functionally exclusive simply because certain segments would not be executed if errors were detected. Rather, the logical operations that are performed would have to be distinct and mutually exclusive. Modules will usually exhibit the exclusivism property when they are control

coupled and the control is being passed into the module rather than out of the module.

3.4.2 Identifying The Factors That Influence Cohesion

Now that the behaviour of the elements internal to a module have been identified, the factors that influence module cohesion may be discussed. The factors that influence cohesion are:

1. Number of logical data elements.
2. Number of logical operations.
3. Number of executable segments.
4. Time dependency of elements.
5. Exclusivism.

The factors listed above are ordered from those factors that have the most influence on module cohesion to those factors that have the least influence on module cohesion. The description of each factor and their corresponding metrics are described below.

3.4.2.1 Number of Data Elements

This measure counts the number of distinct logical data items the module operates on. For example, if the module accepted a queue and a stack then the count for this item would be two.

3.4.2.2 Number of Logical Operations

This measure counts the number of logical operations performed by the module in question. For example, if the name of the module is "READ AND EDIT TRANSACTION" then there are two logical operations.

3.4.2.3 Number of Functional Segments

This measure counts the number of identifiable functional segments in the module. The segments may come from one or more logical operations. The count is the total number of segments from all logical operations.

3.4.2.4 Dependency of Time Elements

This measure quantifies the time dependencies between elements in a module. The time dependency increases according to the number of time dependent segments and the nature of each of the time dependencies. To more precisely quantify the time dependency let us define

- Let I = The number of independent segments in the module.
- Let S = The number of sequentially dependent segments in the module.
- Let D = The number of separate dependent segments in the module.

Given the above definitions, the time dependencies for a module may be measured by the formula:

$$\text{Time dependency} = W[I] I + W[S] S + W[D] D$$

where $W[i]$ is the weight for dependency i . The weight for each dependency should be assigned such that it reflects the relative importance of the item. The constraints placed upon the weights are:

- a) $W[I] > W[S] > W[D]$.
- b) Each of the weights must be positive and non-zero.

In normal situations $W[D]$ should be set to 1. The exact values must be determined by statistical experimentation and are beyond the scope of this chapter.

3.4.2.5 Exclusivism

This measure counts the number of different functions that can be executed exclusively in the module. The count for this is always at least one.

3.4.3 Module Cohesion Coefficient

So far the behaviour of elements internal to a module has been identified, as well as the factors that influence module cohesion. Rather than evaluating the cohesion of a module as falling into one of "N" categories, it would be advantageous if there was a numerical measure of a module's quality. In this section a numerical value called the **module cohesion coefficient** will be described. Let us first of all assign labels to each of the factors described above.

- Let A = The number of logical data elements the module effects.
- Let B = The number of logical operations performed by the module.
- Let C = The number of identifiable segments in the module.
- Let D = The time dependency of the module elements.
- Let E = The number of exclusive operations.

Given the above definitions let us also define

- Let $A' = 1/A$
- Let $B' = 1/B$
- Let $C' = 1/C$
- Let $D' = 1/D$
- Let $E' = 1/E$
- Let $W[i]$ = The weighting for factor "i".

The module cohesion coefficient may be given by the formula:

$$\frac{W[A] A' + W[B] B' + W[C] C' + W[D] D' + W[E] E'}{\text{sum of } W[i]}$$

The above formula yields a value that lies between zero and one. A module cohesion coefficient of 1.0 represents the highest possible cohesion that a module can exhibit -- functional cohesion under the old scheme. A module cohesion coefficient near 0 represents the lowest possible cohesion

that a module can exhibit -- coincidental cohesion under the old scheme. A module's cohesion coefficient may approach zero but it will never equal zero; this is indicative of the fact that all modules are at least temporally cohesive because all statements are executed consecutively (more on this below).

3.4.3.1 Weight Factors

The weights for each of the cohesion factors represents the relative importance of each factor. The exact values assigned to each weight are subject only to the constraints:

- a) $W[A] > W[B] > W[C] > W[D] > W[E]$.
- b) Each of the weights must be positive and non-zero.

Appropriate values should be chosen for each of the weights that reflect the importance of each factor. The exact values that should be assigned to each of the weights could be found by applying five dimensional surface fitting techniques to a set of captured test data. Some additional empirical research is required which is beyond the scope of this thesis.

3.4.4 Comparing The Factors to The Accepted Classifications

This chapter has created a new way to measure the cohesion of a module. It would be beneficial to compare the old classification scheme to the new factor scheme to

demonstrate its correctness. For illustration purposes then, let us compute the module cohesion coefficient for all the accepted cohesion classifications.

In this simplistic model the module cohesion coefficient will be expressed using a five digit number, each digit of which corresponds directly to a measure of one of the five factors that influence module cohesion. The leftmost digit will represent A, the number of logical data elements. The second from the leftmost digit will represent B, the number of logical operations, and so on down to the rightmost digit representing E, the number of exclusive elements. Therefore, the five digit number that will be used is simply the concatenation of the factors found to influence module cohesion.

For the purposes of this illustration only the value 1 shall be used to mean 1 and 1 only when measuring A, B, C, and E; the value 0 will be used to mean 1 or more (recall the inverse relationship A', B', etc.) For measuring D, the time dependencies, the value 2 will be used for totally dependent segments, 1 for sequentially dependent segments, and 0 for independent segments.

If the computation of the module cohesion coefficient and the accepted cohesion classifications correspond in a meaningful way, we would expect functional cohesion to have the highest module cohesion coefficient. The remaining

classifications, each in turn, should be less than or equal to the previous one.

Using the measures described above, figure 7 gives a crude but enlightening comparison between the cohesion classifications developed by Myers and Yourdon and

MODULE COHESION CLASSIFICATION	COHESION FACTORS					PSEUDO COHESION COEFFICIENT
	A	B	C	D	E	
Functional	1	1	1	2	1	11121
Informational	1	0	0	0	0	10000
Logical Function	1	0	0	0	0	10000
Sequential	0	1	1	1	1	01111
Communicational	0	1	1	0	1	01101
Procedural	0	1	0	0	1	01001
Logical Data	0	1	0	0	0	01000
Temporal	0	0	0	0	1	00001
Coincidental	0	0	0	0	1	00001

Figure 7: Comparing Cohesion Classifications and Factors

Constantine and the factors introduced in this chapter. Notice that in figure 7 the classification of logical cohesion appears twice, once called "logical function cohesion" and once called "logical data cohesion". The reason for this shall be explained below. The interesting

point to observe with regards to figure 7 is the correlation of the accepted cohesion classifications to the new cohesion factors. All of the seven accepted cohesion classifications can be expressed using the five cohesion factors introduced. The five cohesion factors are necessary and complete to quantify module cohesion to a higher degree of accuracy than the seven accepted classifications.

3.4.5 Modifying The Accepted Cohesion Classifications

3.4.5.1 Decomposing Logical Cohesion

Notice that figure 7 comparing cohesion factors and classifications above contains all of the accepted cohesion classifications with the exception of logical cohesion. As well, notice the addition in the list of "Logical function cohesion" and "Logical data cohesion".

Recall that in logically cohesive modules the internal elements were related because:

- a) They performed the same logical operation on a number of different data elements.

or

- b) They performed a number of varied operations on one data element.

The operation or the data element in question is not determined by the module, rather, it is specified in some manner by the caller.

According to both accepted classifications, logical cohesion is less desirable than temporal cohesion. However, logical data cohesion is more desirable than temporal cohesion because some segments are performed at the same time and to some extent are functionally bound; segments in a logical data cohesion module are certainly more functionally bound than temporally cohesive modules. While it is true that all of the different data items are mutually exclusive, the code for each individual item is tightly bound. This cannot be said of segments in a temporally cohesive module.

The classification of logical function cohesion is placed immediately after that of information cohesion because they are very similar. In fact, the only difference between logical function cohesion and informational cohesion is the manner in which the caller is coupled to the module; this is a matter of coupling and not an aspect of cohesion.

Given this, the classification of logical cohesion should be removed and replaced with logical data cohesion and logical function cohesion. The classification of logical function cohesion is actually the same as informational cohesion and may be named as such.

3.4.5.2 Removing Coincidental Cohesion

The classification coincidental cohesion should be removed because it is not intrinsically different from temporal

cohesion. Elements of a temporally cohesive module are executed at the same time. Because control implicitly flows from one statement to the next in any classical Von Neumann type machine, this can be said for almost any group of statements that are encapsulated in the same module. If the statements are modified by use of control structures then the module probably exhibits procedural cohesion. Coincidental cohesion is useful as an academic classification but in practice is redundant.

3.5 COMMENTS ON GATHERING EMPIRICAL DATA

Although gathering empirical data in support of the theoretical arguments presented in this chapter is beyond the scope of this thesis, a few points on the subject will be discussed on page 92 in section 3.5.

3.6 CONCLUSION

3.6.1 Summary of Results

In this chapter several important new ideas have been introduced relating to module cohesion. First, the factors that influence module cohesion have been identified and ranked. They are (ordered from most influence to least influence):

- a) The number of logical data elements that the module operates on.

- b) The number of logical operations that the module performs.
- c) The number of identifiable executable segments that the module contains.
- d) The time dependency of the module's elements.
- e) The exclusivism of the the module's elements.

The five factors above are necessary and complete to determine a module's cohesion. Using the above five factors the module cohesion coefficient which is a real value that lies between zero and one can be computed. The higher the module cohesion coefficient, the higher the cohesion between the elements of the module are. The factors that influence module cohesion are of greater utility than the accepted cohesion classifications because:

- There are fewer factors than classifications and therefore fewer facts to remember.
- Using the factors, the module cohesion coefficient can be calculated. The module cohesion coefficient more accurately describes the cohesion of a module than the cohesion classifications. One no longer has to apply the "weakest link" theory when considering various circumstances internal to a module.
- Knowledge of the factors that influence module cohesion may lead to better quality software being developed because its fundamental nature will be better understood. The previous classification scheme was

used as an "after the fact" evaluation of a design which aided very little in the actual development.

Secondly, the factors that influence module cohesion and the module cohesion coefficient have been related to the accepted cohesion classifications developed by Myers and Yourdon and Constantine. The result of the comparison is a slightly modified list of cohesion classifications. Namely, the new list of cohesion classifications are (ordered from highest cohesion to lowest cohesion):

- a) Functional cohesion.
- b) Logical function (or informational) cohesion.
- c) Sequential cohesion.
- d) Communicational cohesion.
- e) Procedural cohesion.
- f) Logical data cohesion.
- g) Temporal cohesion.

3.6.2 Critique of Results

Although this chapter has reduced the complexity of evaluating cohesion by decomposing the problem into five areas, it has not entirely eliminated the judgemental aspects of measuring cohesion. Each of the five influencing factors must be counted before the module's cohesion coefficient can be computed. Identifying the number of the factors in each of the areas will require some judgement on the part of the observer.

It must be emphasized that less judgement is required to compute the module cohesion coefficient than is required to classify the module using the old scheme. Under the old scheme, the observer had to judge module cohesion based on experience and intuition; there were no other criteria. Under the new scheme, the observer is given five criteria upon which he has to make judgements. The sum of these judgements yield the module cohesion coefficient. The new scheme is preferred to the old scheme because the judgements made under the new scheme are a series of small independent judgements based on specific criteria.

Additional work is required to determine appropriate weights for the module cohesion coefficient formula. Once this was done a limit could be placed on the module cohesion coefficient to separate acceptable cohesion from unacceptable cohesion. As well, more work is required to externalize the identification process of each of the five factors.

Chapter IV

COUPLING

"When you can measure what you are speaking about, and express it in numbers, you know something about it; when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."

- William Thomson, Lord Kelvin

4.1 ABSTRACT

Module coupling is a measure of the inter-relatedness of a module or a set of modules. The current coupling classifications are a crude qualitative measure that do not provide a program designer with a deterministic design tool. At best, the current knowledge on coupling provides only supportive influence on program design. In this chapter, the current coupling classifications are enumerated and summarized. The factors that influence module coupling are asserted. Using the underlying factors of coupling as a basis, a numerical representation of a module's coupling called the module coupling coefficient is developed. As well, a representation of a system's coupling called the system coupling coefficient is developed. Using the module coupling factor and the system coupling factor a program designer can evaluate in a scientific manner the quality of a particular program design.

4.2 INTRODUCTION

It is generally accepted that a program's complexity can be reduced by decomposing the program into a series of modules, each of which models a particular aspect of the problem. Given that this is true, there are many possible decompositions of the same problem. Two of the techniques used to measure the quality of a design are coupling and cohesion, the latter having been discussed in the previous chapter. Coupling is a measure of the external relationships between modules while cohesion is a measure of the relationships within a module. The goal of a program designer is to yield a design with high cohesion and low coupling; the elements of an individual module should be highly dependant of each other while individual modules should be highly independant of each other. Along with cohesion, coupling provides a qualitative metric for software designs.

4.3 REVIEW OF COUPLING

The concept of module coupling was pioneered by Glenford Myers [40],[41],[42],[43],[44], and [45]. The subject was then later touched on by Edward Yourdon and Larry Constantine [65]. To some extent, Harlan Mills had a contributing influence as well. This section reviews the history and current thinking about module coupling.

4.3.1 Myers' Work on Coupling

The result of Myers' work was the realization of seven distinct types of module coupling. Myers' coupling classifications are generally accepted to be the norm by the software engineering community. Listed in the order of the most desirable form of coupling to the least desirable form of coupling, the seven types are:

1. No direct coupling.
2. Data coupling.
3. Stamp coupling.
4. Control coupling.
5. External coupling.
6. Common coupling.
7. Content coupling.

4.3.1.1 Content Coupling

Modules that are content coupled exhibit the worst form of coupling under Myers' classifications. Content coupling is hard to find on most programs, not because programmers realize it is bad and therefore do not employ it, but because there are very few instances where such a connection might be allowed by the translator⁵. Two modules are content coupled if one module makes a reference to the contents of the other module. Most, if not all of the instances where content coupling may occur are limited to

⁵ The term translator is used generically in place of the terms compiler, assembler, interpreter, etc.

the assembly language level; most high level language compilers do not permit violations of the following sort:

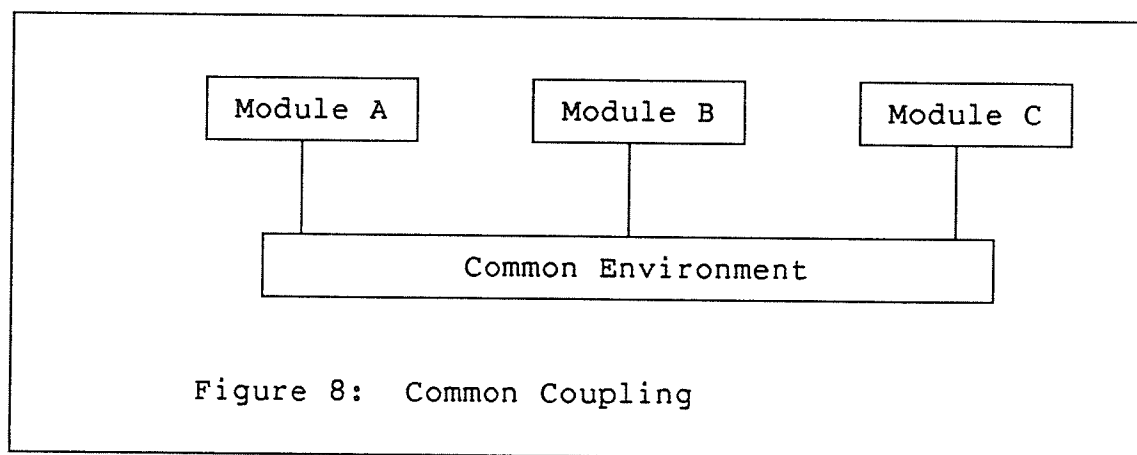
1. One module modifies a program instruction in another module. Any module that modifies an instruction is at best difficult to comprehend, let alone if the instruction exists in another module.
2. One module transfers control to a point inside another module. The point inside the second module is not known as an external entry.
3. Two modules share the same contents. This statement implies that the two modules comprise a single compilation unit. Given this point, one might argue that procedures in a package are content coupled; a case is made later to show that procedures in a package that make references to data defined in the package are actually common environment coupled.
4. One module refers nonexternally to a data item declared in another module. This is possible by knowing the expected offset of a variable in another module.

In all of the above cases the module making the connection must rely on a known offset into the object module. It is then clear why such modules are not independant. If the offset in the object module is altered as a result of a recompile the offset stored in the reference module must be altered accordingly. Since there is no program construct to

automatically enforce such a change, the onus is on the programmer to mentally keep track of references of this nature. As was mentioned, although content coupling is the worst manner in which to make a connection, it does not occur often since only low level languages provide the conditions under which it may flourish.

4.3.1.2 Common Coupling

Common coupling is named after the FORTRAN COMMON statement. Modules are common coupled if first, they refer to the same central data repository which is externally defined, and second, the elements being exchanged are heterogeneous. The elements that are being exchanged could be control elements or data elements. The most obvious example of this type of coupling is exhibited by the use of the FORTRAN COMMON statement. Many modules may reference the elements of a



common block in the manner depicted in figure 8. Figure 8 is misleading because while it accurately depicts that there is only one common copy of the data, it does not show that each module that references the common environment must define within its own scope the layout of the common environment. This means that there exist multiple copies of the layout of the common environment. Should the format of the common block change for some reason, all "N" copies of the layout must change as well. Naturally it is possible for some of the layouts to become out of synchronization with the actual format of the common environment, leading to errors.

Another example of a language feature that exhibits common coupling is the EXTERNAL attribute of PL/I. Although the FORTRAN common block is syntactically different from a PL/I EXTERNAL variable, they are in fact handled in a similar manner by the linkage editor. Both the FORTRAN common block and PL/I EXTERNAL variables are capable of defining a data aggregate in addition to a simple scalar variable. It was Myers' intention that modules coupled through a common environment would be referencing heterogeneous data items. In this thesis such an entity is called a data aggregate.

A problem with modules that are common coupled is that they are difficult to reuse in another context. Since all communication with the module is accomplished through a

common environment, another module wishing to invoke said module would also have to duplicate the environment. In a normal situation this can be a tedious procedure. An application in a different context may not require all data elements of the aggregate, yet the entire environment must be duplicated. Another problem with modules that are common coupled is that they tend to get access to more data than is minimally required to perform their function. Since the common environment is a data aggregate it is likely that not all items in the aggregate are essential for the function of the module. In this sense, common coupling exhibits some of the same problems associated with stamp coupling, which is described below.

An empirical study [13] conducted by Card, Church, and Agresti discovered evidence that suggests that common coupling actually reduces SDLC costs. It should be pointed out that this evidence contradicts Myers' theoretical predictions.

4.3.1.3 External Coupling

Modules are externally coupled if they reference the same externally defined homogeneous data item. The item may be for the purpose of control or it may be data. The following are examples of external coupling:

- Functions that reference the same global variable in an APL workspace are externally coupled.

- Modules that reference the same item via a "V" constant (an external reference) in 360 assembly code are externally coupled.
- Two BASIC subprograms that reference the same global data item are externally coupled.
- Two PL/I modules that reference the same EXTERNAL variable are externally coupled if the variable is not a data structure. If the variable is a data structure then the modules are common coupled.
- A reference within a nested scope of a PL/I procedure is made to a variable not declared within the local scope. The variable is defined within a superior scope that encapsulates both the variable and the procedure in which the reference is made.

There are a few subtle differences between common coupled and external coupled connections:

- a) In common coupling the data in question is restricted to a subset of the modules in the program. Modules that are common coupled must specifically include the name of the common environment that they wish to reference in the module. With external coupling the data in question is available to all modules in the program without restriction. Essentially common coupling and external coupling are very similar except that the "environment" associated with external coupling is a global one.

- b) Common coupling implies a reference to a heterogeneous data item whereas external coupling implies a reference to a homogeneous data item.

It should be noted that the distinctions between common coupling and external coupling described above are Myers' definition, not the author's. The author disagrees with some aspects of Myers' conclusions with respect to these differences. These disagreements will be discussed in a section below.

4.3.1.4 Control Coupling

Myers refers to the situation where modules exchange elements of control (as opposed to data) through the formal parameter mechanism as control coupling. It is important to emphasize the difference between what Myers considered to be control coupling and exchanging elements of control. In every computer program that contains more than one subprogram there is an exchange of the elements of control. When subprogram A invokes subprogram B there is an exchange of control. Depending on how the subprograms are organized, the invocation is a reference that must be resolved either by the compiler, the linker, or by the run time system. While it is true that under such conditions modules do exchange elements of control, it is not considered to be control coupling by Myers. In order to be considered control coupling, the element of control must be exchanged

via the formal parameter mechanism. The control can include information that is passed as a hybrid data item or as the name of a procedure. Elements of control that are exchanged globally are not considered to be control coupling.

Control is an abstract concept. Data is an easier concept to understand than control; data resides in a specific memory location of the computer and can be manipulated by the CPU. Data has a specific representation for each possible state. Control on the other hand has no analogous representation and this is one of the reasons that Myers considered control coupling to be undesirable. Of course control itself cannot be passed through a parameter since control and data are different mediums. Essentially, control coupling is the passing of data between modules such that the value of the data will directly influence the flow of control of the receiving module. It is possible to have control passed as either a withdrawal connection or as a deposit connection.

Here is an example of passing control down to a module. Consider a module that performs two different functions, say read or write. The module READ_OR_WRITE accepts two formal parameters:

- An operation code which indicates whether the operation to be performed is read or write. The code may be a boolean flag or something more elaborate such as a character string.

- A buffer. The buffer will contain the contents of the last record read if the operation was read, otherwise the buffer contains the information to be written. Notice that whether the buffer is an input or an output parameter depends on the contents of the other parameter.

Inside module READ_OR_WRITE there is likely to be a segment of program code that reads:

```
IF OPCODE = READ_CODE THEN
  CALL READ_RECORD(BUFFER)
ELSE
  CALL WRITE_RECORD(BUFFER)
ENDIF
```

Clearly, the manner in which the calling module sets the operation code affects the flow of execution in module READ_OR_WRITE. Surely this can be said about many types of parameters. For example, consider a module that computes the square root of an argument. Should the caller supply a negative argument some sort of error handling code will be invoked. Is this control coupling? The answer is no. Parameters are only control coupled if the module sending the data perceives that the flow of control in the receiving module can be changed by altering the value of the parameter. If there is no perception of control then there is no dependence. Modules often perform error checking on parameters as a matter of course and such instances are not control coupling.

The problem with control coupling is that modules coupled in this fashion are not independent of each other. A much better way to handle the READ_OR_WRITE problem above would be to have the calling module itself invoke either READ_RECORD or WRITE_RECORD directly. In this way the action of invoking the proper routine is physically near the decision of which routine to invoke. Control coupling divorces decisions from their corresponding actions which leads to increased program complexity.

It is also possible to pass control up, as in the following example. Consider a program that analyzes the frequency of words within lines of text. The program might consist of

- A procedure GET_LINE that reads in one line of text from the input file.
- A procedure GET_WORD that extracts words, one at a time, from the line of text that was read in by GET_LINE. (It is assumed that words are not split across lines.)
- A main program that calls the subprograms.

The source for a portion of the program might look like this:

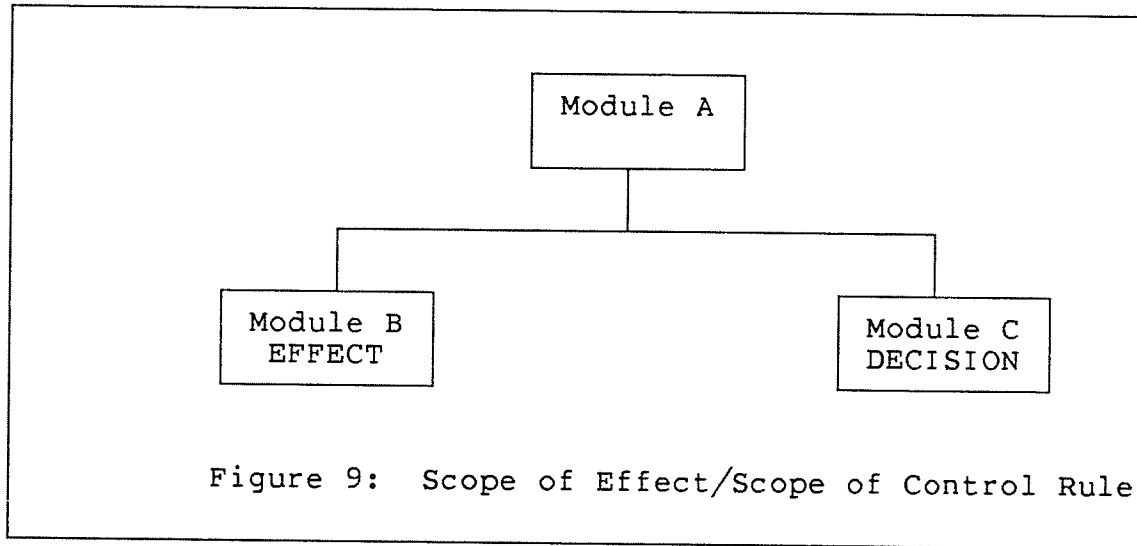
```

CALL GET_LINE(LINE,EOF)
WHILE NOT EOF DO
    CALL GET_WORD(LINE,WORD,EOL)
    WHILE NOT EOL DO
        PROCESS WORD
        CALL GET_WORD(LINE,WORD,EOL)
    END WHILE
    CALL GET_LINE(LINE,EOF)
END WHILE

```

Based on the setting of the flag EOL (which stands for end of line), procedure GET_WORD can direct the mainline to invoke GET_LINE to read in another line of text. As well, one may argue that procedure GET_LINE can direct the main program to finish by the setting of the flag EOF (which stands for end of file).

Of the two forms of control coupling, Myers rates passing control up to be less desirable than passing control down. His argument is that by passing control up it is possible to violate the scope of effect/scope of control rule. The **scope of control** of a module is the module itself and all of its subordinate modules in the hierarchy. The **scope of effect** of a module is all the modules that are ultimately affected by a decision made in the module in question. The scope of effect/scope of control rule is a software engineering rule of thumb that states that any given module should only be able to affect itself and its subordinate modules; essentially, the scope of effect should be contained within the scope of control. Figure 9 demonstrates a violation of the scope of effect/scope of control principle. In figure 9 module C makes a decision



that affects module B. This is a violation of the rule because module B is not within module C's scope of control. This situation is possible because control is being passed up from C to A and ultimately affecting B.

It is also possible to create the situation by passing the offending control element through a COMMON block that joins modules B and C. Myers did not consider such an exchange to be control coupling. This author disagrees with this conclusion. It is the opinion of this author that the important aspect of control coupling is the exchange of control elements, rather than the method of exchange. The method of exchange is a separate issue from that of the usage of the information. This concept will be refined in a later section of this chapter.

4.3.1.5 Stamp Coupling

Myers refers to the situation where modules exchange a heterogeneous data element via a formal parameter mechanism as stamp coupling. Although stamp coupling is not totally undesirable it is not the ideal manner in which modules may be coupled. Consider two modules, module A and module B that accept as a formal parameter the non-homogeneous data structure R. For the purposes of this illustration let us presume that R is defined as:

```
DECLARE 1 R,  
        2 F1    INTEGER,  
        2 F2    REAL;
```

Therefore, the procedure headers for A and B look like:

```
A: PROCEDURE(R);    and    B: PROCEDURE(R);
```

It might occur at some point in time that module A requires another field of information, say F3, to be passed down. Further suppose that module B does not require the additional item. Data structure R could be modified to include the new field:

```
DECLARE 1 R,  
        2 F1    INTEGER,  
        2 F2    REAL,  
        2 F3    CHARACTER;
```

In this case not only would module A have to be recompiled, but every module that references R (in this case module B) would have to be recompiled. Modules that are stamp coupled are not ideally coupled because:

1. If the data structure should change then every module that uses the data structure must be recompiled. It may be difficult to track down all of the modules that reference a particular data structure without the use of programming aids such as cross references or a data dictionary.
2. By passing a data structure a module may have access to more data than is actually required for the module to function. In the case above, even though module B may only needs F1 and F2 it also gets F3, which it may inadvertently change.
3. A module that uses a data structure may be difficult to reuse in another context. Even though the individual data items are all that is required, the user of a stamp coupled module would have to duplicate the data structure and pass it down.

Modules that exhibit stamp coupling are disadvantaged for many of the same reasons that common coupled modules are disadvantaged.

4.3.1.6 Data Coupling

Myers refers to the situation where modules exchange homogeneous data items through a formal parameter mechanism as data coupling. Data coupling represents the most desirable level of coupling that modules may exhibit and still perform useful work in conjunction. In considering

data coupling let us refer to the example used to illustrate stamp coupling above. Instead of using a data structure to pass the information to modules A and B, each field of the data structure R will be passed individually, as in:

```
A: PROC(R.F1,R.F2); and B: PROC(R.F1,R.F2);
```

If module A should require another field it can be added as an additional parameter, as in:

```
A:PROC(R.F1,R.F2,R.F3); and B:PROC(R.F1,R.F2);
```

In this instance only module A need be recompiled. As well, both modules A and B only get access to the minimum amount of data that they require to perform their function. This represents an advantage over stamp coupling and for these reasons modules coupled in this fashion are highly desirable.

4.3.1.7 No Direct Coupling

Myers does not give a name to the lowest level of coupling between modules. In this form of coupling modules do not communicate any data and are therefore totally independent of each other. While a high degree of independence between modules is desirable, modules that do not share data cannot perform any useful work in conjunction. Coupling is a generalization of the different types of connections between modules, and if there is no connection then there is no coupling. In this thesis this form of coupling is defined

as **null coupling**. This category of coupling is necessary to complete the enumeration but is otherwise uninteresting.

4.3.1.8 Summary of Myers' Coupling Classifications

According to Myers, the coupling classification that is assigned to a module is the classification of the module's lowest quality connection. This means that if a module had 1000 data coupled connections and 1 common coupled connection the classification assigned to the module would be common coupling. This is the "weakest link in the chain" theory, or more colloquial terms, "one bad apple spoils the barrel". Myers considers that the levels of coupling above control coupling are reasonable ways to couple modules.

In summary, Myers' coupling classifications are:

- 1. Null coupling.**

No external references implies no coupling. However, no data exchange implies no work can be done in conjunction. This classification is useful for enumeration purposes only.

- 2. Data coupling.**

All exchanges involve homogeneous data items through the formal parameter mechanism.

- 3. Stamp coupling.**

At least one exchange involves heterogeneous data items through the formal parameter mechanism. All other exchanges are data coupled.

4. Control coupling.

At least one exchange involves homogeneous elements of control through the formal parameter mechanism. All other exchanges are stamp and data coupled. The element of control being exchanged can either be a subprogram name or a hybrid data item.

5. External coupling.

At least one exchange is made through a global homogeneous data or control item. All other exchanges are control, stamp, or data coupled.

6. Common coupling.

At least one exchange is made through a global heterogeneous data or control item. All other exchanges are external, control, stamp, or data coupled.

7. Content coupling.

Modules bypass normal linkage conventions for either data references or control transfers. Occurs when external references are satisfied by the programmer at coding time rather than by a piece of software such as the compiler or the linker.

4.3.2 Deficiencies in the Coupling Classifications

There are several reasons why Myers' coupling classifications do not have the utility that one would like. In this section some of the reasons are outlined.

Firstly, Myers' coupling classifications provide a qualitative measure of a module's coupling. It would be advantageous if a scheme could be worked out whereby the

MODULE COUPLING CLASSIFICATION	FIRST ORDER DEPENDENCY COEFFICIENT
Content	0.95
Common	0.70
External	0.60
Control	0.50
Stamp	0.35
Data	0.20

Figure 10: Myers' Qualitative Coupling Values

coupling of a module could be assigned a quantitative value. Myers did develop quantitative values for each coupling classification which he called the **first order dependencies**. These values were intended to be used to compute the probability that a module would change given that it is coupled in a certain way to another module. The first order dependency values are listed in figure 10. For example, consider the scenario depicted by figure 11. In this case module A is coupled to module B in any one of the six meaningful ways. Suppose it is control coupled. Using the dependency values Myers postulates that if a change were to



Figure 11: Coupling Dependency

be made to Module A, then the probability that Module B will have to be changed is 50%. Likewise, if the two modules were data coupled and Module A were changed, there would be only a 20% chance that Module B would have to be changed. These values turn out to be unproven because they were not empirically derived; they were "guesstimated" based on Myers' own intuition and experience. More importantly, these values fail to take into consideration the case where all but one of the connections to a module are data coupled; because of this one bad connection the module receives a low coupling rating. Intuitively, there must be a better way to account for the coupling of a module.

The classifications are not linear. They are quantized by reference type. The coupling classifications give no insight into the tradeoffs that are involved in the various connection types. For example, consider a module that accepts as a parameter a complex number. It is known that a complex number consists of two separate and distinct parts, a real part and an imaginary part. Based on the coupling classifications, it would be best to pass such an item as

two separate parameters rather than placing the parts into a record and passing the record. In this way stamp coupling, which rates lower than data coupling, would be avoided (at least according to Myers' definition). A contrary example to that one involves a personal experience of the author. The author once attended a third year undergraduate class where the instructor preached that the number of parameters to a subprogram should be kept at an absolute minimum; it was said that the best way to reduce parameters to a subprogram was to place all the individual items into a record and pass the one record. In this way there would only be the one parameter, internally consisting of all the other parameters. While this is technically possible it is not always in the best interests of reducing program complexity. Exactly how much better is data coupling than stamp coupling and when does one choose a particular one?

The coupling classification assigned to a module is the classification of the module's lowest quality connection. This is the "weakest link in the chain" approach and although it may apply to other situations, intuitively it should not be used for module coupling. This theory does not take different combinations of module connections into account. For example, does a module that has 1000 data coupled connections exhibit lower coupling than a module that has one common coupled connection? Well, if one follows the classifications religiously it does, but which

module is really less dependant on the other modules of the system?

There are gaps in the classifications where control coupling is concerned. The subject of data and hybrid usage is dealt with but pure control connections seem to be almost totally ignored. Recall that control coupling was the name that Myers assigned to the passing of hybrid elements through the formal parameter mechanism. These elements of control could either be subprogram names used as a procedure parameters or hybrid data like a boolean flag. There is no mention made of hybrid data exchanged through a common block. The classifications do not consider such an exchange to be control coupling even though there is a non-standard exchange of control taking place. It should be pointed out that exchanging data elements through a common block is quite different from exchanging pure control or hybrid elements through a common block. Indeed, exchanging control is always more complex and obscure in human terms than exchanging data under any circumstances. The classifications fail to take this into account. There should be a difference between passing control elements globally and passing other data globally.

Finally, the coupling classifications cannot be used to compare two rival program designs, because it is an "after the fact" evaluation. By the time the coupling of the system can be measured the system has already been built, or

at least this is what typically happens. If this is so, then measuring the coupling amounts to an academic exercise only. An analogous situation exists when attempting to predict the strength of concrete for some building project. Although the required strength of the concrete is known when the ingredients are mixed, it is impossible to measure the actual strength of the batch until the concrete has cured, which is 24 days later. By this time of course the building project has already proceeded under the faith that the concrete is strong enough. The person mixing the batch must use his previous knowledge of good and bad recipes and prepare the batch to have greater strength than that actually required.

The current state of coupling is much the same. Coupling is more likely used as software engineering "rule of thumb" rather than as a direct measure of design quality. That is to say, that they may be kept in mind during the design stage, but they certainly cannot be used to compare two designs. Myers himself admits

"...its major purpose is to provide an understanding of desirable and undesirable module relationships, thus guiding the program designer in the establishment of module interfaces."

4.3.3 Motivation for Developing Factors

In an effort to improve our knowledge of coupling it is advantageous to identify the individual factors that influence module coupling. This is desirable for several reasons.

Firstly, if coupling is at all a meaningful attribute of module behavior, identification of the factors that contribute to coupling could lead to a better understanding of coupling, and therefore higher quality software.

One has to question how Myers' coupling classifications were derived. Normally, one identifies factors and then forms general classifications based on grouping sets of factors. The coupling classifications were not formed in this way. The classifications were "proven" with a combination of practical experience, theoretical argument, and a little hand waving. Once the contributing factors were known, they could allow the verification and enhancement of Myers' coupling classifications.

The factors may be easier to remember and apply than Myers' classifications. For example, it may be that there are fewer factors than classifications. It could also turn out that factors are more representative of module behavior than the classifications.

Given that the individual factors are known, and the factors can be measured individually, it may be possible to

measure a module's coupling quantitatively rather than qualitatively. This would be desirable because the classification system is so judgmental that only an experienced software engineer can perform the evaluation. A knowledge of the factors might permit the construction of automated tools for comparing module coupling.

4.3.4 Yourdon and Constantine's Work on Coupling

While remaining purely qualitative in nature, Yourdon and Constantine delved deeper into the nature of coupling than did Myers. Rather than identifying a series of broad coupling classifications, they attempted to identify the individual factors that influence module coupling; as well, they identified different connection types as having different influences on module coupling.

Yourdon and Constantine identified the factors that influence module coupling as (in random order):

- a) Type of connection.
- b) Complexity of interface.
- c) Type of information passing through the connection.
- d) Binding time of connection.

Factor (a), the type of the connection, refers to the manner in which the module in question acquires the data. For example, data that is passed via a parameter is considered to be of a different coupling type from data that is referenced from a common block.

Factor (b), the complexity of the interface, refers to the number of connections which the module in question has to objects not defined internally. Basically, the greater the number of objects, the greater the complexity of the interface. For example, a module referencing ten externally defined items has an interface that is more complex than a module that references two externally defined items.

Factor (c), the type of information passing through the connection, refers to whether the information passing through the connection is being used as data or control. It is considered that modules that pass control are more highly coupled than modules that pass data.

Factor (d), the binding time of the connection, refers to the time at which the connection is satisfied. For example, connections that are satisfied at execution time are more independent than connections that are satisfied at compile time, which in turn are more independent than connections that are satisfied at link-edit time, etc. The philosophy is that if a connection is satisfied at execution time it is easier to change than a connection made at an earlier stage in the development. Some authors, such as Jensen and Tonies [30], do not consider the binding time of the connection to be one of the factors involved in coupling.

Yourdon and Constantine thought it worthy of consideration to examine each connection to a module more

closely. They categorized a module's collective interface as belonging to one of these three classes:

- i) Minimally connected.
- ii) Normally connected.
- iii) Pathologically connected.

Under their scheme, a module is minimally connected if all connections to the module are fully parameterized and all transfers of control are via a single entry to the module. This means that the only type of connection permitted according to Myers' classifications would be data, stamp, and control coupling. Any other type of connection to the module implies that the module is not minimally connected.

Yourdon and Constantine define a module to be normally connected if the module is minimally connected with the exception of one or more of the following conditions:

- There is more than one entry point to the module, each of which must be minimally connected.
- After invocation of the module, control returns to a statement other than the next sequential statement in the calling module.
- The module is invoked by some mechanism other than a structured transfer of control.

If the module is not minimally connected and it is not normally connected then it is defined to be pathologically connected. A pathologically connected module is one that exhibits connections such as content, common, and external coupling under Myers' classifications and violates one or more of the conditions necessary for normally connected modules.

4.4 THE FACTORS THAT INFLUENCE MODULE COUPLING

Yourdon and Constantine introduced the concept of coupling factors, but they did not develop the concept fully. In this section the factors that influence module coupling are asserted and rigorously examined. The factors are developed to the point that the factors rather than the classifications can be used to evaluate the coupling of a module.

In this thesis the factors that influence coupling for a given module (given in random order) are:

1. The number of connections.
2. The interface type of each connection.
3. The complexity of information flowing through each connection.
4. The usage type of information flowing through each connection.
5. The direction of information flow through each connection.

6. The binding time of each connection.
7. The number of distinct connected modules.

The factors are explained in more detail below.

4.4.1 Number of Connections

As the number of connections emanating from a module increases, the coupling of the module increases, regardless of the quality of each of the connections. This follows from the fact that a module with no connections has no coupling; therefore as the number of connections increases the dependency increases. Basically, Myers did not focus on a module's interface, but Yourdon and Constantine did. This thesis takes the concept of a module interface one step further than Yourdon and Constantine who thought that one bad connection spoiled the interface. They had three ratings for a modules interface as a whole: excellent, good, and bad. In this thesis each module connection is evaluated individually instead of as a group.

It should also be noted that all connections emanating from a module have to be enumerated. This includes pure control connections as well as data and hybrid connections.

The empirical study in [13] found that the more connections a module had the more fault prone the module was. This study gives empirical evidence which supports the theoretical basis for this factor.

4.4.2 Interface Type of Connection

In electronics a connection can be made between two modular units by

- Removing the covers from each unit and then soldering an adjoining wire to the back of the printed circuit board of each unit.

This is a tightly coupled connection.

- Running a cable with connectors on each end that plug into sockets on each of the units.

This is a loosely coupled connection.

- Transmitting information over a medium that is not physically connected, such as the wireless remote control for a television.

This connection is made dynamically and is therefore extremely loosely coupled.

In software the number of different types of connections depends on the facilities provided by the host programming language; the connection types are closely related to the scope rules of the host programming language.

There are two basic categories of connection types, arranged here from best to worst for data connections and arranged from worst to best for pure control connections:

a) Normal Connection.

The connection is run through the formal parameter mechanism of the host programming language.

b) Common Environment Connection.

The connection is not run through the parameter mechanism. The connection could be a global one or known to a subset of the modules. The important aspect is that it is not through the formal parameter mechanism.

There are many different types of common environment connections. Here are a few examples.

- FORTRAN COMMON or PL/I External.

These connections are resolved by the linker.

- Package Import.

A variable is defined in the public area of another package and then imported from that package into the host.

These connections are resolved first by the compiler, to obtain the correct offset into the package, and then by the linker, to obtain the location of the package. This is permitted in the language Mesa [39].

- Transcendental Import.

Same as package import except that the package that the item is imported from itself imports the item from another package. This recursive import can be carried on to an arbitrary depth. This is permitted in the language PROTEL [9].

- Global.

The variable is defined outside of the current module but within the same compilation unit. These types of references are resolved by the compiler.

- Object reference.

Reference to an object in an Object Oriented Programming Language [20]. Resolved at run time by the run time environment.

Notice that other than syntax, the main difference between these common environment connection types is the resolution of the reference; specifically, who resolves the reference and when.

4.4.3 Complexity of Information

In electronics, a connection can carry either something as simple as B+ or something as complex as a time division multiplexed signal. The difference is the amount of intelligence that is conveyed through the connection as perceived by the observer.

In software, a connection can convey either something simple like an atomic integer data item or something as complex as a stack. As described earlier in this thesis, there are several categories, arranged from least complex to most complex:

- Atomic data items.

Simple variables of any base type. For example, integer, real, boolean, etc.

- An aggregate of data items.

For example, a PL/I AREA variable.

- Linear complexity.

For example, an array or a record with fields of same type.

- Molecular items.

For example, an array of records.

- Abstract items.

For example, any variable reference with a pointer.

It is possible to decrease the number of connections and at the same time increase the complexity of the connections. For example, instead of passing two parameters, a programmer could pass the two parameters in one connection as the elements of one record. This would decrease the interface complexity in terms of the number of connections but increase the complexity of the one connection. This is really discussing how cohesive the data in any one connection is. For the purposes of this discussion it is assumed that the data in any one connection exhibits an acceptable level of cohesion.

4.4.4 Usage of Information

The usage of information refers to whether the intelligence is being used as data or for control. Most programming languages constrain the use of information as control or data via the type mechanism (assembly languages are an exception, because in general names are not typed). For example, two procedure names could not be added together in Pascal because the compiler will not permit it.

There are three typical usages of intelligence, arranged here from lowest coupling dependency to highest coupling dependency:

- Data.

For example, a variable name refers to location in memory containing a value.

- Control.

For example, a procedure or entry name stores the address of executable code.

- Hybrid.

Information that is declared as data but is used to exchange control between modules. For example, flags, opcodes, and return codes. Hybrid data tends to create higher inter module dependencies than either data or control.

4.4.5 Binding Time

Binding time refers to the point in time at which a connection is established, or when an external reference is resolved. The later the connection is established, the more independent the module, thus lower coupling. Examples of binding times arranged from highest coupling to lowest coupling are:

- Coding Time.

For example, the reference is satisfied by the programmer coding a constant or a label which refers to an element inside another module. All content coupling is done in this way.

- Compile Time.

For example, the programmer used a constant which was resolved by the compiler.

- Link Time.

For example, The programmer called an externally defined procedure which was resolved by the linker.

- Load Time.

For example, the programmer calls a system resident procedure which gets resolved by the loader.

- Dynamically.

Object Oriented Programming Languages implement dynamic binding for all procedure calls.

4.4.6 Direction of Information Flow

The direction of the information flow affects the coupling of a module where hybrid control elements are concerned. The direction of information flow of data intelligence or pure control intelligence has no effect on the coupling dependency. Arranged from most desirable to least desirable the coupling dependency of a module is affected by the direction of information flow as follows:

- Out.

Control information coming out of a module is often unavoidable and does not violate the "black box" concept of a module. The problem can be solved by making the calling module larger by including in it the code from the called module, which is undesirable because of reuseability problems. The tradeoff is to pass control up or cope with large poor cohesive modules.

- In.

Passing control into a module is avoidable by moving the decision into the caller; for example, opcodes, or device codes. Passing control into a module also degrades the cohesion of the module.

- In-Out.

Bad for same reasons as IN, but complicated by the two way movement.

4.4.7 Number of Distinct Destinations

The greater the number of distinct modules that module X is connected to, the greater the system depends on module X and

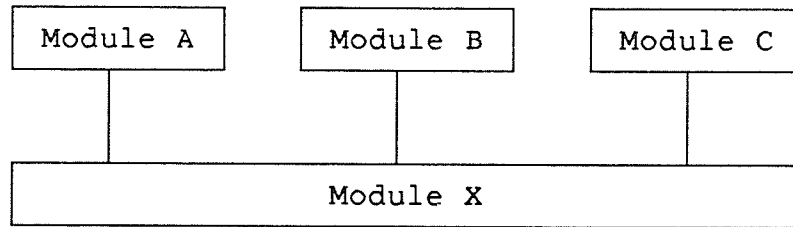


Figure 12: Connection Density: Scenario 1

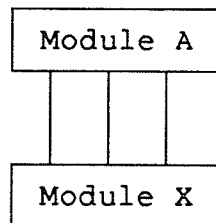


Figure 13: Connection Density: Scenario 2

the higher the coupling of module X.

Consider the scenario posed by figures 12 and 13. The number of connections emanating from module X is the same in

both scenarios. It may also be assumed that all things other than the number of modules that module X is connected to are the same, for example, each connection carries information of the same complexity and usage, etc. In figure 12 there are three modules connected to X as opposed to only one module connected to X in figure 13. If module X were to be radically changed in some fashion then it would be the responsibility of the individual performing the change to ensure that no other parts of the system were affected by the change. In figure 12 there are three places to check but in figure 13 there is only one place to check, at the first level. Clearly, if the number of distinct connected modules can be minimized then changes affecting one module are more localized and do not tend to spread through the system as readily. Reducing the number of connected modules therefore minimizes the amount of work required to enact a serious change.

The empirical study in [13] found that the greater the number of descendants of a module there were, the more fault prone the module was. This empirical evidence supports the theoretical basis for this factor.

4.5 DESCRIBING CLASSIFICATIONS WITH THE FACTORS

Now that the factors that influence module coupling have been asserted, it is a worthwhile exercise to highlight the factors that are necessary to describe each particular coupling classification.

4.5.1 Describing Null Coupling with Factors

The classification of null coupling is highlighted by the fact that there are no connections between the stated modules, and therefore there is no coupling. The factor that is necessary and complete to describe this classification is simply the number of connections. Specifically, the number of connections is zero. No other coupling factors influence this coupling classification.

4.5.2 Describing Data Coupling with Factors

With data coupling the factors that are necessary and complete to describe the classification are as follows: There are one or more connections; the interface type of each connection in the module interface is parameter and the complexity of the data being exchanged through each connection is atomic; further, the usage of the information through each connection is data and can be neither hybrid nor pure control; the direction of the information flow through each connection, as well as the distinct number of connected modules, is inconsequential; the binding of each connection must be completed any time after coding time.

4.5.3 Describing Stamp Coupling with Factors

In the classification stamp coupling the factors that are both necessary and complete to describe the classification are as follows: There are one or more connections to the module; the interface type of each connection in the interface is parameter; there exists at least one connection in the module interface that has information of greater than atomic complexity flowing through it; other connections, if any exist, may have information of any complexity flowing through them; the usage type of all information exchanged through the module interface must be data and can be neither hybrid nor control; the direction of the information flow through the connections as well as the number of distinct destination modules is inconsequential; the binding time of each connection in the interface must be later than coding time.

4.5.4 Describing Control Coupling with Factors

In the classification control coupling the factors that are necessary and complete to describe the classification are: There are one or more connections to the module; the interface type of each connection through the module interface is parameter; the usage of the information in at least one of the connections is either hybrid or control; the usage of the information in the other connections, if any exist, can be either data, hybrid, or control; the connections can have information of any complexity flowing

through them; the direction of the information flow through the connections, as well as the number of distinct connected modules, is inconsequential; the binding time of each connection must be later than coding time.

4.5.5 Describing External Coupling with Factors

In the classification external coupling the factors that are necessary and complete to describe the classification are: There are one or more connections; at least one connection is not made through the formal parameter mechanism, that is to say that it is made through a common environment; any other connections, if they exist, can either be made through the formal parameter mechanism or through a common environment; the complexity of the information flowing through the common environment connections is of type atomic; the complexity of the information flowing through the parameterized connections, if any, can be of any complexity; the usage type of the information flowing through any of the connections can be any of data, hybrid, or control; the direction that the information flows through any of the connections as well as the number of distinct connected modules is inconsequential; the binding time of each connection is later than coding time.

4.5.6 Describing Common Coupling with Factors

In the classification common coupling the factors that are necessary and complete to describe the classification are: There are one or more connections; at least one connection is made through a common environment; the other connections, if any exist, are made through either a common environment or through the formal parameter mechanism; the complexity of the information flowing through at least one of the common environment connections is greater than atomic; the complexity of information flowing through the other connections, if any exist, can be of any complexity; the usage of information flowing through any of the connections can be either data, hybrid, or control; the direction of information flow through any of the connections as well as the number of distinct connected modules is inconsequential; the binding time of all of the connections is greater than coding time.

4.5.7 Describing Content Coupling with Factors

In the classification content coupling the factors that are necessary and complete to describe the classification are: There are one or more connections; the binding time of one or more of the connections is coding time; all other factors are inconsequential.

4.6 DISAGREEMENTS BETWEEN FACTORS AND CLASSIFICATIONS

Many serious software engineering textbooks contain a section of software engineering heuristics, or "rules of thumb". These rules of thumb are a set of principles obtained through experience rather than rigorous formalized methods. Since they do not fit elegantly into any other section of a book and are well worth publishing [30],[52],[65], they are usually collected together in a chapter unto themselves. The coupling classifications contradict some of these heuristics. Presented below are a few instances where the coupling classifications and the coupling factors disagree. It is interesting to compare the outcome of these disputes to the principles outlined by the heuristics.

The classifications portray the number of connections emanating from a module as being inconsequential, as long as they are all good connections. The factors disagree with this. The factors state that as the number of connections increases the coupling increases. The heuristics state that the number of direct descendants of a module should be minimized, with the upper limit being 7 plus or minus two [37]. Obviously, the greater the number of descendants the greater the number of connections (each descendant is one control connection). Empirical studies [13] support this position.

The classifications suggest that all common environment or pathological connections are bad. The factors disagree. Sometimes a common environment connection is advantageous because it decreases the total number of connections and/or the total number of connected modules. This theory agrees with a heuristic that states that information should not be passed through a module that does not act upon it. This is because such information is an implementation detail that should be hidden from modules that do not directly manipulate it. Recall that the goal of modular design was to reduce program complexity, which is sometimes accomplished by reducing the number of connections. Empirical studies [13] support this position.

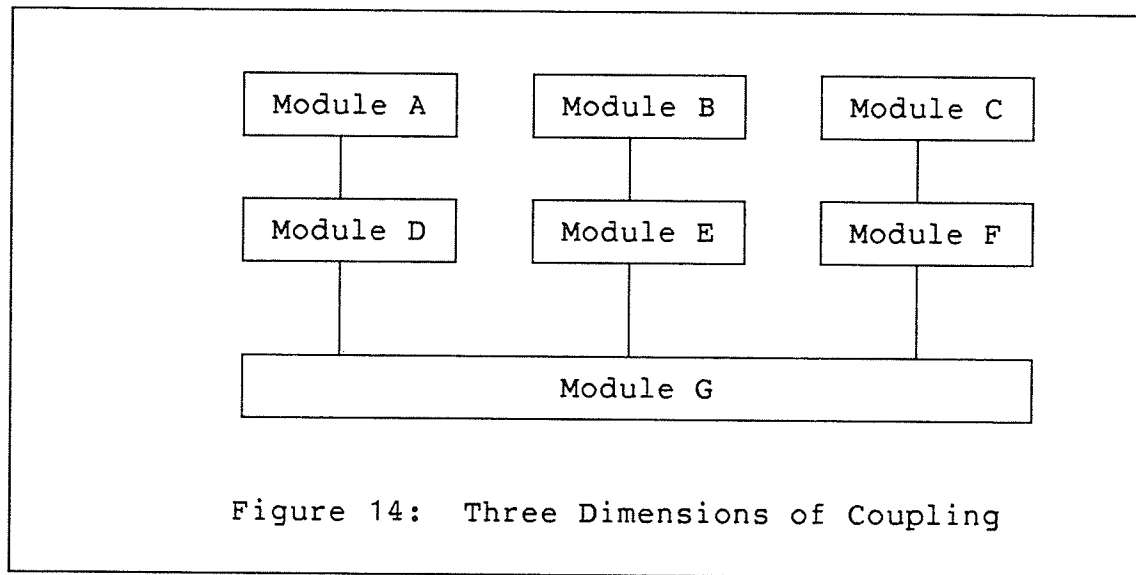
The classifications state that control coupling is bad. While it is true that control coupling is not an ideal way to couple modules, often it is unavoidable. The factors state that the direction of the control coupling is important. Passing control down (opcodes, function codes, item codes, etc.) leads to higher coupling than passing control up. When control is passed down, the called module is not a "black box" because the caller has specific knowledge about its function. On the other hand, passing control up (return codes, flags, condition codes, etc.) does not expose the contents of the called module to the caller and is part of the normal reporting process for a subordinate.

The coupling classification assigned to a module is dictated by the lowest quality connection. In other words, the connections are considered as a group rather than being accounted for individually. Essentially, "one bad apple spoils the barrel", or more precisely, one bad connection tremendously increases the coupling of a module. The factors maintain that this theory has some element of truth to it but not to the extent that it is portrayed. The classifications give far too much weight to the type of the connection without concern for the other factors such as number of connections, usage, etc. In module coupling one poor connection does not degrade module coupling with a quantum leap, it degrades it proportionately in accordance with the weight assigned to the pertinent factors.

4.7 DIMENSIONS OF COUPLING

Before continuing to develop a quantitative coupling metric it is important to emphasize the three distinct types of coupling measurements that can be taken. For ease of explanation consider the system of modules in figure 14. Let us presume for the purposes of illustration that the weight of each connection in figure 14 is one. There are three types of coupling measurement that are of interest in this system:

- a) The direct coupling dependency between any two given modules. For example, in figure 14 the coupling



between module D and module G could be measured. To do this all the connections between module D and module G, and only the connections between module D and module G would be taken into account. No other connections emanating from either D or G are counted. In this example there is only one connection between modules D and G, therefore the total of the weights is 1. This is a measure of the direct dependency between modules D and G. This is represented by the function

$$CD(D:G)$$

for the coupling dependency between modules D and G.

- b) The coupling dependency between the system and an individual module. For this measure all the connections to a given module are accounted for. For example, in figure 14 the direct coupling dependency

between the system and module G could be measured by taking into account all connections to/from module G. In this case module G is connected to modules D, E, and F by one connection each, giving a total weight of 3. This is represented by the function

CD(G:)

- c) The coupling dependency of the system. In this case all connections in the system are taken into account. In this example the total weight would come to 6. The weight of 6 emphasizes that the connections are being counted, not the number of module interfaces. If the number of interfaces were counted the total would be 12, not 6. This is an important distinction because not all connections have an interface on each side of the connection. Take for example a connection between a module and an external variable. The module referencing the external variable has an interface but the external variable is global and therefore has no interface to speak of. This is represented by the function

CD(:)

For any two modules, X and Y, it is expected by definition that

$$CD(X:Y) \leq CD(X:) \leq CD(:)$$

4.8 A THEORETICAL QUANTITATIVE COUPLING METRIC

4.8.1 Bounds of the Metric

In chapter 3 a theoretical quantitative value representing module cohesion was developed. The value amounted to a ratio that yielded a non-negative number between 0 and 1 inclusive. A number that approached 1 indicated that the module exhibited good cohesive properties whereas as the value approached 0 it was an indication that the cohesive properties of the module were less than desirable. The coupling metric that will be developed below is slightly different from the cohesion metric. The coupling dependency value yielded from the computation is a non-negative number between zero and infinity. If, for example, $CD(A:B)$ is being measured, a value of zero indicates that there is no direct coupling dependency between the modules. As the coupling dependency value increases the actual coupling dependency between the modules increases. There is no absolute upper bound to the value derived for the coupling dependency.

4.8.2 Factor Weight and Factor Precedence

To apply the coupling metric to $CD(X:)$ for instance, all connections emanating from module X must be enumerated. All of the coupling factors outlined in section 4.4 on page number 124 must be taken into account. When performing the calculation there are two levels of "weights" to be considered.

The **factor weight** or **FW** is the first type of weight. The FW takes into account the relative importance of one factor to another. For example, under the existing classification scheme the type of the connection carries far more importance than the number of connections. All of the factors with the exception of the number of connections would be assigned appropriate FWs. The number of connections is not assigned a weight per se because the fact that each connection is taken into account essentially assigns it a weight.

The **factor precedence** or **FP** is the second type of weight. The FPs represent the relative weights of the different levels of quality in each factor. For example, it is generally agreed that a binding time of coding time is less desirable than a binding time of compile time, which is less desirable than a binding time of link time, etc. The FPs are only associated with factors 2 through 6 inclusive.

In order to simplify the number of entities to be considered and to increase the metric flexibility the FWs and FPs shall be compressed into the **FWPM**. Let FWPM be the factor weight-precedence matrix which is a matrix that has bounds $[2..7, 1..N]$, where N is the upper bound on the number of FPs. So, FWPM[3,3] contains the factor precedence of the third best type of complexity, namely heterogeneous.

Row 7 of the FWPM is a special case since it represents the number of distinct connected modules, not the precedence of something that happens inside a connection. Column 1 of row 7 contains the precedence of having 1 connected module, column 2 of row 7 contains the precedence of having 2 connected modules, etc. Therefore, using the FWPM the relationships between the factor precedences need not be linear. As well, the precedences of one factor need not necessarily be greater or less as a group than the precedences of some other factor.

4.8.3 An Algorithm for Coupling Dependency

Given the above definitions, the following pseudo-code algorithm will compute CD(X:).

```

ND is the number of destinations.
NC is the number of connections.
CD is the coupling dependency.
F is the factor number.
P is the precedence number.

CD := FWPM[7,ND]
LOOP FOR C := 1 to NC
  LOOP FOR F := 2 to 6
    P := identified precedence of factor[F]
    IF the factor is relevant AND
       the factor is in context THEN
      CD := CD + FWPM[F,P]
    ENDIF
  ENDLLOOP
ENDLOOP

```

The above algorithm for CD(X:) will also work for CD(X:Y) if the variable ND is set to 1.

The above algorithm for CD(X:) will also work for CD(:) if the variable ND is set to the total number of modules in the system.

4.8.4 Normalization of CD(X:)

Since the weights and precedences in the FWPM are to be determined by empirical methods it is possible to scale the values in the matrix up and down by an arbitrary constant value. Although the empirical data will not be collected as an aspect of this presentation, criteria necessary for such a study will be discussed in section 3.5 on page 92 of this dissertation. The outcome of the empirical study should yield a FWPM which contains values such that when CD(X:Y) is calculated a value of 1.0 will result if and only if:

- There is one connection between the modules.
- The usage of the information flowing through the connection is data, it can be neither hybrid nor pure control.
- The complexity of the information is atomic.
- The interface type of the connection is formal parameter.
- The connection is resolved at run time.

4.9 NOTES ON GATHERING EMPIRICAL DATA

Although gathering empirical data in support of the theoretical arguments presented in chapters 3 and 4 are beyond the scope of this thesis, a few points on the subject are important enough to be mentioned here. A methodology for collecting software engineering empirical data in general has already been developed in [2]. In addition to the general criteria that have already been established, the following criteria should be kept in mind for this particular problem:

- The data should be captured from one large project rather than from many smaller projects. This ensures that the same data scheme is not measured more than once. It is better to measure ten different things than to measure one thing ten times. Measuring one large project will ensure that this is not done, to any great extent at least. As well, many of the problems which module coupling and module cohesion address do not occur in small systems.
- The project to be measured should be implemented in one programming language only. Crossing language boundaries may introduce more modules and associated complexities than are intrinsically required to solve the problem, thereby impairing the results.
- The programming language used in the project should be one of the state of the art programming languages with

package oriented or object oriented features. While it is true that many projects are still implemented in first generation high level languages such as FORTRAN and COBOL, these programming languages do not offer many features that the newer languages do. For example, neither FORTRAN nor COBOL offer any sort of nested block structuring. The author does not advocate that block structuring is either good or bad. Block structuring is a feature that could affect the outcome of the analysis and therefore should be measured.

The empirical data should be gathered at all of the following SDLC stages:

- During programming.

The amount of development effort should be measured for different module types. The process could be automated such that the system would report the exact amount of time spent coding each module.

- During testing.

The number of errors discovered in each module should be logged. The number of errors versus module coupling and cohesion should be correlated.

- During deployment.

The customer's satisfaction, or lack thereof, such be recorded. The changes made to the system before an acceptance is issued should be correlated to module coupling and cohesion.

- During maintenance.

When an enhancement is made to the system the number and type of modules affected should be recorded. The locality of such changes should be correlated with module coupling and cohesion.

Clearly, collecting data from a project would have to be identified as one of the goals of the project from the onset. As well, the type of systems eligible for the study would be limited to a very small subset of the available systems.

4.10 CONCLUSION

In this chapter several important concepts dealing with module coupling have been introduced. Firstly, the factors that influence module coupling have been identified. While the original concept of coupling was a brilliant discovery, it can only be used by an individual who has substantial software engineering experience. Its practical usefulness is limited in that sense. This chapter has externalized the underlying factors of module coupling. The identification of the factors leads to a better understanding of module coupling, whether the factors are accounted for qualitatively or quantitatively. The factors can be identified by individuals who possess mediocre experience, and in fact are straightforward enough to be identified by a computer program; it would be feasible to construct a

computer program that would evaluate the coupling of a module or set of modules. Secondly, the idea of rigorously enumerating the connections and assigning a weight to them has been introduced. The concept of assigning a weight to each type of factor to obtain a numerical representation means that the coupling dependency can be represented in a more precise and meaningful way.

Chapter V

SOFTWARE SYSTEM ENGINEERING

"Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away."

- A. Saint-Exupery

5.1 ABSTRACT

The design documentation prepared for a software system is often obsolete before the system is put into production. This is because the design phase is not integrated with the other SDLC phases, and hence the design documentation is too expensive to maintain. Given that a design for a software system has been documented, it is not obvious that the quality of the resulting program can be predicted solely from the design documentation. This chapter presents the specifications for a software tool called SPAM, an acronym for System Planning Model, that addresses these problems. A software system is modelled with a system schema, which is basically a detailed machine representation of a transformation schema. The underlying programming language contains control elements only; all data elements, which are managed by SPAM, are part of the system schema and must be specified during the design stage. Because program data elements are divorced from program control elements,

features such as flexible scoping, system wide cross referencing, separate compilation consistency checking, SDLC phase integration, and a host of others can be provided by SPAM. Since the system schema contains a mapping of all connections in the system, coupling metrics can be applied to the schema to predict program quality before any of the control elements have been detailed. The benefits derived from SPAM are two-fold. Firstly, the SDLC enjoys a higher degree of integration and automation. Secondly, system designers view the workings of the system from a different philosophical perspective than traditional systems because the data elements are divorced from the control elements.

5.2 INTRODUCTION

This chapter presents the specifications for SPAM, a software tool for automating limited aspects of the software development lifecycle. The name SPAM is an acronym for System Planning Model. Although "system planning model" suggests that SPAM's functions are limited to the design phase of the SDLC, SPAM provides an integrated environment for the design, programming, testing, deployment, and maintenance phases of the SDLC. The focus of this chapter is on the facilities that SPAM provides for the design phase of the SDLC. The facilities provided by SPAM for the other SDLC phases are outlined but are not described in any great amount of detail.

It should be noted that only the theoretical aspects of SPAM are discussed in this chapter. The implementation of such a tool is not addressed.

5.2.1 Overview of This Chapter

This chapter describes the need for a software system development tool that follows the entire life cycle of the system. The results from the previous chapters are used to define the problems and shape the tool. The concept of a software system model is discussed, along with a new scope mechanism that is an integral part of the model. The remainder of the chapter is used to describe the individual components of the tool: how the component is used, how it integrates into the rest of the tool, and what advantage the component provides over conventional approaches.

5.2.2 Design Documentation

One of the problem areas in the design phase of the SDLC is the documenting of software system designs. The problem is not the lack of a documentation technique as there are plenty of them [63]. Design documentation is either not attempted or quickly becomes obsolete because of the amount of labour that is required to keep it up to date. In practice very few software engineering shops regularly use and place weight on the results of formalized software design techniques [4]. It is interesting to observe that the design stage is not emphasized by many software

engineers. An empirical study in [57] concluded the following, among other things:

- Syntax type error corrections aside, the most frequent type of change in the SDLC is an unplanned design modification.
- The most common type of error is one made in the design or implementation of a single component of the system.
- Design changes tend to be nonlocalized with respect to individual components but localized with respect to sub-systems.

Design documentation is typically kept on paper. When a change is made to the design either during or after the design phase, the original paper design becomes obsolete and must be redone. These design changes not only surface during the design stage itself but also appear during other SDLC phases. For example, if a design error is discovered during the testing stage, any resulting design change must ripple up through the SDLC phases affecting the source code and the design documentation. The number of design changes encountered during SDLC phases other than the design phase are numerous; sometimes a design error is not discovered until the system is delivered to the customer. Quite often design changes are required during the maintenance phase of the SDLC, not because some sort of an error has been detected, but because the customer requests that new features be added to the system. A software system is

rarely a static entity. When a design change is made during the maintenance stage of the SDLC the following are affected:

- The documentation supplied to the customer may have to be changed.
- The source code of the program must be changed.
- The test data for all or a part of the system must be changed.
- The design documentation must be changed.

Basically, if a change is made at phase "N" of the SDLC, then quite likely changes also have to be made to phases N-1, N-2, ..., 1. An article in IEEE COMPUTER [46] reported on the findings of the **Strategic Defense Initiative Organization (SDIO)** with respect to the SDLC as follows:

The panel regarded the "waterfall diagram" paradigm, "the portrayal of a software development project proceeding smoothly and uni-directionally from requirements specification to design specification to coding, testing, and delivery", as a misconception of the reality. "The truth is that these steps can, and often do, feed back into any of their predecessors."

Management tends to view keeping design documentation up to date as a fruitless exercise, hence it is either abandoned after the first few iterations or not attempted at all.

As well as the cost aspects of keeping design documentation up to date, many software professionals view it as a boring exercise beneath their talents.

SPAM provides facilities that ease the burden of updating design documentation by integrating the design phase into the other SDLC phases. SPAM stores the design of a software system in a machine representable form that must be kept up to date because it is used by successive SDLC phases.

5.2.3 Software Design Metrics

There are many methodologies for deriving a program design based on a set of program specifications [63]. Different design methodologies place the emphasis of the design on different aspects of program behavior. Some design techniques base the design on functional aspects while others base the design on the behavior of the data. While many of the design methodologies are quite specific and detailed regarding how to produce the design, they lack a measure of the resulting design's quality. When presented with two different designs for the same program, it is difficult to identify the superior design. The identification of the superior design is left to the intuition of the observer.

In general, there are two types of metrics, **result metrics** and **predictor metrics**[15]. A result metric can be used to determine the quality of a completed program. A predictor metric forecasts the quality of a program before the program is built. Most of the metrics in software engineering are result metrics, which implies that a program

is written and then a metric is applied to the completed product to determine the quality. In [11] Fred Brooks comments that the authors of a software system should "plan to throw one away". Brooks states that the experience gained from constructing the first version can be used to construct a higher quality second version. Essentially, the outcome of the first version is used as a result metric. Result metrics are of limited usefulness for very large software systems.

A predictor metric is analogous to a medical prognosis whereas a result metric is analogous to a medical post-mortem; the results of a medical post-mortem may be conclusive but it is impossible to convince the patient that this is an indication of success. Very large software systems such as BNR's DMS [7] and America's SDI involve thousands of man years of development, and hence are simply too large to "build one and throw it away". The cost to throw very large systems away and start over is prohibitive.

It would be desirable to have a predictor metric that could be applied at an early stage of the SDLC that would determine program quality before the program was written; at an early stage of the SDLC back tracking is relatively cheap. This chapter describes how SPAM can compare the quality of rival program designs before any source code has been written by applying metrics to a model of the software system.

5.3 MODELS

It is desirable to be able to construct a model for a software system. Having a model of a software system is advantageous for the same reasons that models are desirable in general, namely:

1. A model is relatively inexpensive to construct when compared to the product itself.
2. Models have smaller resource requirements than the product itself.
3. Models exhibit the same behavior as the product. Many models do not exhibit all the properties of the product as the model is constructed to exhibit only the properties that the observer is interested in.
4. Models can be constructed in less time than the product can be constructed.
5. The model can be studied in isolation to observe weaknesses.

The problem with modelling a software system is that software is an intangible entity; while software is often used to construct models of other things it is not clear how software itself can be modelled.

In order to determine how a model of a software system could be constructed, the definition of "system" must be brought to light. A system is "a group of interacting objects" [56]. For the purpose of a software system the

"objects" are modules and the "interaction" between the objects is the exchange of data and control. The essence of any system is the interacting parts. The objects themselves are of less importance from the system perspective as they could be coalesced into one huge object that performs the same function, but then there would be no system. As was described earlier in this thesis, such a program could be made to work but would be counter-productive. Since program data elements and program control elements are conveyed over hypothetical paths called connections, the "interaction" in a software system is represented by connections. Therefore, to model a software system a representation of all of the connections between the modules is required.

SPAM models a software system by requiring all the inter-module connections to be fully detailed during the design phase of the SDLC. The quantitative coupling metric described in chapter 4 can be applied to the model to determine the coupling dependency of the software system. The model can be altered or alternative models proposed and the coupling dependency of the alternative models can be compared to the original model. The process can be iterated until a design with the lowest coupling dependency is yielded. It should be noted that the cohesion of the model cannot be rigorously computed until the program is completed, which will be discussed in further detail below. The benefits of a model in general are realized for such a software system model because:

1. The system model is inexpensive to construct when compared to constructing the entire system. The model can be constructed by a small group of designers, thereby assuring conceptual integrity [11].
2. The system model has smaller resource requirements than a working version of the system because the model contains no executable code.
3. The system model exhibits the same behavioural properties as the working system because all of the interactions are the same.
4. The system model can be constructed in a relatively short amount of time since there is no executable code to write.
5. The system model can be studied by applying quantitative coupling metrics to it thereby exposing any weaknesses.

5.4 THE PHILOSOPHICAL FOUNDATIONS OF SPAM

The philosophical foundations of SPAM are somewhat different than the philosophical foundations found in other design methodologies. Consider figure 1 on page 28. Before the advent of modular programming, programs were constructed as one large monolithic block. The complexity of such a program is high because one large monolithic block of code exhibits extremely poor cohesion. (This is represented by "point B" on figure 1.) Such a large program is difficult

to comprehend unless an individual has a great deal of experience with that particular program. When modular programming was introduced, an attempt was made to reduce the size of the monolithic blocks in an effort to make programs more comprehensible. The problem that is encountered with doing this is that as the module size decreases the number of external connections running between the modules increases. Because human beings have a limited capacity for processing information [37], as the number of connections increase, the perceived program complexity increases. Because humans cannot cope with large amounts of information, the correct size for a program's modules is a tradeoff between the size of the modules and the number of external connections. (This compromise is represented by point C on the graph.) This is the theory that figure 1 is attempting to convey.

The ideal scenario for a program would be one under which all the modules were very small and there were very few external connections. In this way the modules could be understood because they would be small and the number of external connections would be humanly manageable. Unfortunately, the realities of the situation are such that this rarely, if ever, occurs. If humans are to compose, manage, and understand complex software systems, these tradeoffs are imposed because of our limited information processing capability.

The number of external connections in a program falls into the realm of coupling. The size of a module falls into the realm of cohesion. Therefore, the correct size for a module is a compromise between module coupling and module cohesion. In chapter 3 the factors that influence module cohesion were discussed. One does not have to understand the factors described in chapter 3 to know intuitively that modules with a large number of source lines have a greater potential for poor cohesion than modules that have very few source lines. This is fundamentally true because as a side effect to the number of lines increasing, the numbers of each cohesion factor will also increase.

It is not known how to compute the cohesion of a module before the statements in the module have been coded. Once the module has been coded, the cohesion can be rigorously evaluated, but not before. Given the system model, an estimate of the cohesion can be made based on the factors that the model exhibits with respect to cohesion, but some of the factors are not determined until the module is coded.

On the other hand, if a complete and detailed mapping of all the external connections in the program is produced, then the module coupling can be calculated for each module and for the system. This can be done because the model exhibits all of the factors that influence module coupling. Given that such a mapping exists for a software system, and that the mapping is stored in an internal representation on

a computer, the perceived coupling complexity of the system diminishes. This is because the machine is managing the coupling rather than the coupling being mentally managed by a human. No longer is the complexity of the coupling directly linked to the human capacity for processing information. Because coupling is managed by the computer, the other important aspect, cohesion, becomes the prime ingredient to program complexity.

Recall that the size of a module was moderated by the coupling-cohesion tradeoff. Since the coupling aspect is managed by a machine, the cohesion aspect can be forced to its near ideal limit. Essentially, this means that the size of each module in the system can be very small without concern that program complexity will escalate due to the high number of connections. The aspects of the program that humans deal with, namely the module size, are kept small and mentally manageable, while the aspects that the machine deals with, namely the external connections, are unconstrained.

The net effect of all of these measures is lower program complexity as perceived by a human observer. SPAM provides a discipline under which these philosophical beliefs can be enforced and measured.

5.5 SCOPE

5.5.1 History of Scoping

The "scope" aspects of modularity were discussed in chapter 2. Recall that the scope rules of a language impose restrictions on the lifespan and visibility of data items. The evolution of scope in programming languages has developed in three generations:

1. First generation scoping.

The scope of an identifier is the subprogram in which it is declared. Typical examples are the programming languages FORTRAN and COBOL.

2. Second generation scoping.

The scope of an identifier is the subprogram in which it is declared and all subprograms nested within the subprogram. These are the "block structured" languages. Typical examples are PL/I and Pascal.

3. Third generation scoping.

A program entity encapsulates data and subprograms within one scope [1], [9], [54], [39], [47], [60], [61], [12], [50], [20], [48]. These are the package and object oriented languages. Typical examples are PROTEL and Smalltalk.

The three generations of scope rules presented above are a generalization of the progression of program scoping. Obviously there are exceptions and extensions to each generation.

Many of the languages from generation "N+1" introduce new scope facilities, but no other new features over the languages from generation "N". One can surmise from the evolution of programming languages that scope is very important in representing and organizing module work.

Connections that are "normal" do not violate scope rules. This is not true of all "pathological" connections. For example, consider a variable in a FORTRAN named common block that is shared by a subset of a system's modules. The lifespan and visibility of such a variable does not coincide with the normal workings of the scope mechanism. While shared variables are sometimes the most elegant way to solve a particular programming problem, it is not clear why they do not fit into the "normal" scope mechanism. Is the scope mechanism inadequate or is the solution to the programming problem the wrong one?

The author suggests that the first, second, and third generation scope mechanisms unduely constrain the lifespan and visibility of data. Consider two approaches to software design, data flow diagrams [19] and structure diagrams [65]. A data flow diagram (DFD) bases the system design on the individual transitions of data as they flow through the system. A structure diagram (SD) bases the system design on the hierarchy of control within the system. Essentially, the DFD models the way that data behaves in the system and the SD models the way that control behaves in the system.

There is a process known as **transform analysis** whereby a DFD can be transformed into a SD [52]. One has to wonder why such a process is necessary. Why is only one of these diagrams insufficient to express the design of a software system?

The reason that two distinct techniques are required to convey the internal aspects of a system is because there are two distinct and separate aspects to the workings of a system, namely control and data. The scope mechanisms of programming languages attempt to constrain the behaviour of data to force the data to behave in the same manner as control. Whenever the data behaves in a manner different from the control, a "pathological" connection is created. Recall that pathological connections create greater program complexity than normal connections. In order to properly specify the aspects of a software system it is necessary to handle both the behaviour of data and the behaviour of control, each in a natural and convenient way. This can be accomplished by having a scope mechanism that is more or less separate from the control aspects of the programming language. In this thesis such a scope mechanism is called **flexible scoping**.

5.5.2 Flexible Scoping

Flexible scoping gives the designer the ability to specify the properties and behaviour of each data entity in the system independent of the control aspects of the system. This means that the lifespan and visibility of a data item are no longer derived from the physical location of the declaration in the program source. Rather than the data being defined in the programming language, the definition of the data, its lifespan, and its areas of visibility are all part of the system schema. The system schema is manipulated through an interactive program called the **Interactive Schema Editor (ISE)**, which is discussed below. It should be noted at this point that the editor for changing the system schema, which is referred to as the ISE, is a different editor from the one used to change program source.

The advantages of flexible scoping are:

- The lifespan of the data and the visibility of the data can be specified separately. They are not tied together as closely as they would be under scope rules.
- The lifespan and visibility of the data are not dependent upon the lexical organization of the executable elements.
- Via the ISE, the ability to alter the data can be restricted to authorized personnel only. This ensures the conceptual integrity of the design.

- By definition, there are no pathological connections. A pathological connection is a connection that is not formally specified and may violate normal scope rules. Since all connections are mapped by the system schema, there is never any doubt about a connection's source and destination. Because the lifespan and visibility of all data items can be specified in the manner that is natural to the behaviour of the data, there are no violations of the "scope" rules.

5.6 COMPONENTS

There are several components that comprise SPAM. These components are:

- a) The system schema. The system schema is created and changed via the ISE, which is an acronym for interactive schema editor.
- b) The language that the source code is programmed in. This language is called UPL, which is an acronym for "Underlying Programming Language".
- c) The editor that is used to create and change the program source. This editor is called the INE, which is an acronym for "Interactive Nodule Editor".
- d) The compiler that is used to generate object code using the system schema and UPL source as input. The compiler is called SIC, which is an acronym for "SPAM Integrated Compiler".

These components are discussed below.

5.7 SYSTEM SCHEMA

A **system schema** is a model of a software system. The concept of a system schema is based on the "transformation schema" devised by Ward [55], which is an extension of the data flow diagram to represent control and timing. The differences between a DFD on paper and the system schema are:

- The system schema is stored on a computer in a machine readable form.
- Any changes to the system schema are made through the interactive schema editor and take effect immediately.
- A human readable form of the system schema, which would look like a DFD, can be generated automatically by the ISE. The subject of an algorithm for the automated drawing of DFD's has been addressed in detail in [3]. Other than mentioning that SPAM must have this capability, the discussion of the drawing of DFD's is beyond the scope of this thesis.
- The design information is either entered into the ISE via an interactive command language or via a mouse driven, graphically oriented, windowed, what-you-see-is-what-you-get interface. The method of input is an implementation detail and hence is beyond the scope of this chapter.
- Since the system schema is stored in a machine readable format, a number of metrics may be applied to the

schema without the need to transport the data from paper to machine.

- The system schema is not a mirror-image of the connections in the system as the DFD is. The system schema is forced on the programming language because the language must inherit the data definitions from the system schema. The underlying programming language is a specially tailored language that does not contain any data definition facilities. It is therefore not possible to change the definition of a data item without changing the system schema because they are one and the same.

In order to develop a system schema, the system designer must specify all of the following at some point, not necessarily in the order given below:

- The processing entities that perform the transitions on the data. These are referred to as **nodules**.
- The data entities. With each data entity a description of the data's purpose and function must be given; this description constitutes the data dictionary.
- The connections between the nodules, both data and control.
- The lifespan of each data entity.
- The visibility of each data entity with respect to the nodules.

- Sub-system information. The designer has the capability of defining a subset of the nodules in a system as belonging to a sub-system. The purpose and advantages of sub-systems are discussed below.
- Test data for each nodule and test data for each sub-system. For each nodule and sub-system in the system the designer must specify a set of possible inputs and the expected outputs for each input set. The test data are used to perform automatic unit testing on the nodules and sub-systems.

5.7.1 Nodules

A nodule is an entity that performs a functional transition on one or more data entities. The term nodule is used in this chapter instead of the term module to emphasize the different philosophy employed by SPAM. A nodule differs from a module in the following ways:

- Ideally, a nodule performs only one transition on one data entity. Although there is no mechanism to enforce this, the name nodule implies such a processing entity.
- Ideally, a module is a self contained unit that contains both the executable elements and the data elements that are necessary to model one aspect of a problem. (The ideal module rarely occurs because of the introduction of pathological connections which violate the black-box concept.) While a module

contains the data and executable code necessary to model one aspect of a problem, a nodule does not. A nodule contains only the executable elements necessary to perform one transition on one data item. The nodules do not contain any data. The data is harboured by the system schema.

Basically the differences between a module and a nodule are philosophical in nature. The term nodule is used to emphasize the philosophical differences.

It should be noted that the nodules must be defined when the system schema is built, but are not necessarily filled in until some later time.

5.7.2 Sub-Systems

The system designer has the capability of defining a subset of the nodules to be a **sub-system**. Essentially the sub-system boundary forms a hypothetical shell that the nodules are encapsulated in. The only thing that penetrates the shell are connections that enter and leave the sub-system. The designer is permitted to define as many sub-systems as desired providing that sub-systems at the same level do not intersect. Sub-systems are also allowed to encapsulate other sub-systems. Therefore, a sub-system is a hypothetical boundary surrounding a set of nodules and/or other sub-systems. At the lowest level an individual nodule is considered to be a sub-system. Essentially the sub-

systems can be considered to form a hierarchy, the lowest level of which can be at the nodule level, and the highest level of which can be considered to encapsulate the entire system. In the hierarchy sense the sub-systems represent a "block structured" sort of approach to the system. The key difference between a block structured hierarchy and the sub-system concept of SPAM is that the lines of the sub-systems are imposed by the system designer rather than by the programming language.

Since there are many levels of sub-systems, it is advantageous to develop some sort of notation that distinguishes the depth of one sub-system to another. The lowest level sub-system, which is an individual nodule, shall be called **SS(0)**. There are many **SS(0)**'s in the system, one for each nodule. As one goes higher in the hierarchy the number of the sub-system increases by one. For example, the sub-system that encapsulates a **SS(0)** would be **SS(1)**. As the notation indicates, there is no limit to the depth of sub-system nesting. By definition, the sub-system that represents the system itself will be called **SS(top)**. The sub-systems that are enclosed in **SS(top)** are all at level **SS(top-1)**.

Other than forming an hypothetical shell that encapsulates nodules, connections, and sub-systems, the properties of a sub-system are:

- The sub-system level provides a boundary that the lifespan of a data item can be specified for. When control enters the sub-system, all variables that are defined as being born on entry to the sub-system are created. When control leaves the sub-system the variables that are defined for that sub-system are put to death.
- The visibility of a variable can be specified by sub-system.
- The sub-system forms a compilation unit. That is to say that a particular sub-system can be compiled in its entirety.
- The sub-system forms an entity for unit testing and integration testing. When defining a sub-system the system designer must specify a set of test data for the connections running into the sub-system and the expected results for the connections running out of the sub-system. Using the test data provided during the design phase, SPAM can perform unit testing on the individual nodules in the system. Providing that all SS(0)'s are validated SPAM can move to the next sub-system level, and perform integration testing on SS(1).
- The sub-system is a unit of packaging [65]. In particular, the sub-system is a unit for **feature** and **image** packaging. The notion of packaging is an important one in the deployment stage of the SDLC which is discussed further below.

- The sub-system is a unit of change. Since very large software systems must be allowed to evolve, there must be accommodation for implementing and tracking changes to the system. This is done at the sub-system level. The problem of tracking changes is discussed further below.
- The sub-system provides a level of programmer team isolation. For example, one group of programmers may be assigned the responsibility of a particular sub-system. Their access and change privileges can be restricted to a sub-system only. Essentially this allows management to control the project by delegating responsibility and ensuring that unsanctioned changes are not made to parts of the system.
- The sub-system provides a logical level for cross reference purposes. Since the system schema stores all data definitions, SPAM provides cross referencing facilities. The search space for the cross reference can be restricted to a particular sub-system. Normally, system wide cross referencing is performed by a separate software tool. The advantage of using the schema for cross referencing is that the cross reference is always up to date. There is no need to run a cross referencing program against the entire system after every change. Usually such cross referencing programs are run only periodically, perhaps once per week. This means that the cross reference is almost always out of date when it is needed.

5.7.3 Data Entities

When composing the system schema, the system designer must completely detail every data entity in the system. For each data entity, the following must be specified:

- The type of the data, as in REAL, INTEGER, etc.
- The complexity of information. This includes the dimensions and the structure of the information, if such information is applicable.
- The usage of the information.
- The lifespan of the data item. The lifespan of the data is synchronized to the entry and exit of control from either an individual nodule or a sub-system.
- The visibility of the data item. The data can be selectively visible in a subset of the nodules that the lifespan is defined for.

5.7.4 Connections

The connections that run between the nodules are completely detailed as part of the system schema. Each connection in the system must be described with the following information:

- The source of the connection.
- The destination of the connection.
- The lifespan of the connection. This information includes the birth and death of the connection. Essentially this gives the binding time of the connection.

- The direction of information flow through the connection.
- The identification of the information that will flow through the connection. This can be described by giving the name of the data entity that will flow through the connection.

It should be mentioned that connections can run between the following entities:

- One nodule and another nodule.
This sort of a connection either transfers a data reference from one nodule to another or transfers control from one nodule to another.
- A data entity and a nodule.
This is either a uni-directional withdrawal connection or a bi-directional connection.
- A nodule and a data entity.
This is either a uni-directional deposit connection or a bi-directional connection.
- A data entity and another data entity.
This is a connection between data items. This is similar to a pointer reference in standard programming languages. Although it is the exception rather than the rule, such connections can be established dynamically. The bounds of such connections are statically defined.

The advantage of enumerating all the connections through the system schema are:

- Since every connection is formally specified, the connections or each connection can be easily traced through the ISE.
- Should a data entity change, the ISE can inform the designer of all the connections and nodules that are affected by the change.
- Since the ISE has knowledge of all connections, it is possible to produce a detailed cross reference listing for the entire system, or for a given sub-system.
- All connections in the system are formal and external. Since the nodules do not contain any data there are no internal connections.
- It is relatively simple to produce a DFD from the connection information.
- The consistency of separate compilation is always ensured because the interface information is stored in one location.
- It is relatively easy to have SPAM conduct automatic unit testing since it has a detailed knowledge of all inputs and all outputs. Since testing shows the absence of bugs, not their presence, this aspect of SPAM merely assists the human process of testing and does not replace it.

5.7.5 Data Lifespan and Visibility

The lifespan of a data item is a separate consideration from the visibility of the data item in SPAM. The only constraint is that the visibility of the data item must be restricted to a subset of the nodules that the lifespan of the data is defined for. Recall that variable instances that obeyed this constraint were defined to be "properly synchronized".

The lifespan of a data item is restricted to a particular sub-system.

5.7.6 Test Data

One of the problems with any program, regardless of the size, is maintaining a set of test cases for the program as the program evolves. Typically, after a program is tested the test cases are discarded. This is unfortunate because the test cases represent a great deal of labour and most often can be reused at a later time. SPAM permits the designer and/or programmer and/or tester to develop a set of test cases for each nodule and sub-system that stay with the nodule for the lifetime of the system. The tester can specify a series of possible inputs for each nodule and the expected outputs for those given inputs. Upon command, SPAM will automatically unit test each nodule against all the given test cases. SPAM reports any anomalies that are found. Given that all of the nodules within a given sub-

system are unit tested without error, SPAM then tests the sub-system that the nodules reside in. In this way SPAM will methodically unit test and then integration test every nodule and sub-system in the system. Because the test cases are part of the system schema they stay with the system. When a problem is reported from the field the test cases for the problem area are augmented to include the new problem.

Not all nodules have their output in the form of outgoing connections. For instance, a given nodule may output information to the printer or other device. When SPAM is instructed to perform unit testing, it recognizes all non-standard output connections in a nodule. SPAM will compile the nodule in such a way that the device calls are written to buffers; the buffers can be interrogated by the test system. By changing certain statements in a nodule when preparing for unit testing, SPAM can apply all of the predetermined test cases to a nodule and validate the results.

The advantages of the testing capabilities of SPAM are:

- Test cases are always retained and augmented as the system ages.
- The entire system can be methodically tested by SPAM. The need for testing can be forced upon anyone making a change to the system. For example, SPAM will not permit a load to be built if an untested change will be

incorporated in the load. SPAM forces the entire system to be tested if a change is made to any one component. This is referred to as **regression testing**. The thoroughness of the test cases is a matter that is left up to the testing staff. SPAM can ensure that each nodule has at least one test case, but the quality of the test cases is the responsibility of humans. The best way to ensure that test cases are adequate is to employ a separate test team for the project. The testing team and the programming team should work as adversaries.

5.7.7 Reuseability

The reuseability of nodules and sub-systems is addressed by SPAM. The term **reuseability** refers to the ability to use a piece of a system in another context in the same system without redefining the portion of the system that is being reused. The designer has the capability of physically defining a nodule or sub-system once but using it logically many times in the schema. In this way traditional reuseability of system parts is achieved.

5.7.8 Portability

One of the traditional virtues of modular programming that SPAM does not support is the portability of individual system components. The term **portability** refers to the ability to lift a segment of source from one system and use

it in another context in a different system. Portability is normally not possible with SPAM because the information for one particular component of the system does not all reside in one place. In a traditional programming environment, the program source contains both the executable elements and the data elements necessary to implement some function. By copying or moving a section of the source code, all the information is transferred. In SPAM, the information describing the data elements is kept separate from the information describing the executable elements. By moving program source written for SPAM, only the executable elements are transferred. The information describing the structure of the program and the data information is part of the system schema. Because the information is disjoint, portability is sacrificed. Although it would be possible to have an export function in SPAM that would merge the necessary information in order to ready it for transportation, it does not fit elegantly into the overall design of SPAM.

5.8 THE UNDERLYING PROGRAMMING LANGUAGE

The programming language source contained in the individual nodules of the system is a language called UPL, which is an acronym for **Underlying Programming Language**. There are three components that must be discussed with regards to UPL:

- The language UPL itself.

- The editor that is used to create and change UPL source.
- The compiler that converts the UPL source and schema information to object code.

These components are discussed below.

5.8.1 Language

It is not the purpose of this section to provide a complete and detailed description of the syntax and semantics of UPL. Indeed, it is not the mandate of this chapter to describe any of the features of SPAM in any great amount of detail, but instead to give a general overview of the important concepts that distinguish SPAM from other programming environments. The purpose of this section is to outline the facilities of UPL that are different from those of some of the other state of the art programming languages.

SPAM is based on the premise that advantages can be realized from handling the data elements of a program separately from the executable elements. This means that some of the traditional facilities of a programming language are removed from the programming language and handled by some other tool. In this case the other tool is the ISE, which is part of SPAM. The facilities that are normally part of a programming language but do not exist in UPL include:

- Declaration of variables, types, files, etc.

UPL does not have any declaration facilities for defining variables, files, and types; all definitions are part of the system schema and are handled by the ISE rather than UPL.

- Specification of the structure of the program.

In a traditional programming language the structure of the program is defined by facilities in the source language such as packages, objects, procedures, subroutines, functions, etc. UPL has no such facilities because the structure of the program is defined in the system schema.

- Definition of the interfaces that ensure consistency during separate compilation.

UPL does not contain facilities for defining module interfaces because such information is defined through the ISE.

- Facilities for defining compilation units.

Some traditional programming languages have facilities for defining program segments which physically decompose the program. UPL has no such facilities because compilation units are specified at the subsystem level of the system schema.

- Facilities for defining logical program units.

Most traditional programming languages have facilities for defining subprograms which logically decompose the program. UPL has no such facilities because the

structure of the program along with the interface of the individual parts are part of the system schema.

- Facilities for defining the visibility, also known as scope, of a variable or other data entity. UPL has no such facilities because variable lifespan and visibility are defined through the ISE.

With the above items excluded from UPL, the programming language basically consists of control type statements. The following list is not intended to be an exhaustive inventory of UPL statements, rather, it is intended to give the reader the flavour of the language:

- Data manipulation statements. Although data is not defined through UPL, it can be manipulated with the basic set of arithmetic operations including assignment, addition, subtraction, multiplication, division, etc.
- Control structures. UPL provides a standard complement of control structures for altering the normal sequential flow of control. The control structures include the traditional IF-THEN-ELSE, FOR, WHILE, REPEAT, CASE, etc.
- Transfer of control. Although the control connections themselves are specified through the ISE, UPL contains statements for transferring the flow of control to another module in a structured manner.

- Other control transfers. UPL contains facilities for handling exceptions and multi-tasking.

5.8.2 The Interactive Nodule Editor

The interactive nodule editor (INE) is the editor that is used to create, modify, and delete UPL source code. The INE is not a program editor in the normal sense. A normal editor considers the source to be a series of lines of text. Although any aspect of the program can be altered via the editor, the editor has no knowledge of the programming language. The editor can permit the programmer to alter any aspect of the program because all aspects are part of the program source. In this sense a syntax directed editor (SDE) is like any other program editor. A SDE has a knowledge of the syntax of the target programming language, and hence is able to detect program syntax errors as the program is typed in. The INE is not an SDE. A programmer cannot change all of the aspects of a program through the INE because not all of the aspects are specified in the program source. Only the executable aspects are part of the program source in UPL.

Through the INE, a programmer is only permitted to edit the contents of one nodule at a time. It is not possible to change the overall structure of the program through the INE. The structure of the program is part of the system schema, hence it must be changed through the ISE. The INE is

integrated with the system schema in such a way that the INE can read the system schema to determine the connections that emanate from the nodule being edited. When a programmer opens a nodule to edit it, the INE presents the programmer with a description of the nodule's interface as well as the source code that is in the nodule. The programmer is only permitted to change aspects of the program associated with UPL via the INE.

5.8.3 The SPAM Integrated Compiler

The **SPAM integrated compiler (SIC)** is the software tool that produces object code based on the information provided in the nodules and in the system schema. A few aspects of SIC are discussed below.

5.8.3.1 Compilation Units

As was stated above, the unit of compilation in SPAM is a sub-system. Recall that a nodule is considered to be the lowest level of sub-system, which was written SS(0). It is possible to change the compilation units in the program without altering the program source. This would be done through the ISE.

Sometimes the order of compilation of the individual units in a program is important. The order of compilation can be specified through the ISE.

5.8.3.2 Separate Compilation Consistency

In order to produce object code from the source program the SIC must have access to the contents of nodules as well as the system schema. Since the interface information for each nodule is stored in the system schema there is no problem ensuring the consistency of units during separate compilation.

5.9 CHANGES TO THE SYSTEM

As well as organizing the effort required to construct a large software system from the onset, SPAM provides facilities for implementing changes as the system evolves. Although it is not the purpose of this thesis to address changes to software systems in general, there are several important aspects of system evolution that should be described.

There are two important ingredients to system change:

- Flexibility.

The system must be flexible enough to accommodate a number of different types of changes. Without flexibility the system cannot evolve.

- Trackability.

Every change to the system must be documented and related back to problem reports.

SPAM provides flexibility for the following types of changes:

- Images.
- Schema Versions.
- Features.
- Fixes.

5.9.1 Images

SPAM provides the capability for defining a system **image**. An image is a subset of a software system that is tailored for a particular target environment. For example, if the software system were a compiler then an image would be required for each machine language that object code was to be generated for. Most of the compiler would be the same in every image. The differences from image to image would be the low level code generation routines. It is expected that from image to image there would be mutually exclusive system parts, but the core of the system would be the same.

Hardware differences are just one reason why a system image might be created. An image may be created for reasons other than hardware differences. For example, in every compiler there are many common sub-systems such as parsing, symbol table manipulation, etc. It would be possible to create an image for every different language that was to have a compiler generated for it. The common sub-system of the images would be the symbol table manipulation routines,

the parser routines, etc. The secondary parts of the system would be the parts that are unique to compiling a particular language. Each different compiler would compile a different language but they would all generate code for the same low level machine language.

5.9.2 Schema Versions

SPAM provides the flexibility of **schema versions**. Each image can have many versions. Every time a change is required to the system schema of a particular image, a new version of that image is produced.

5.9.3 Features

Often a software system is sold to a customer who may choose whether or not to purchase one or more optional **features**. A feature is an optional part of an image. There are two approaches to selling software with optional features:

- Compose each image with all possible features built into it. Before the software is sent to the customer turn a software switch for each feature either on or off, depending on whether the customer has purchased or not purchased the feature respectively. There are two problems with this. First, customers have a way of finding software switches. Second, the resource requirements of most customer systems will be higher than necessary because of all the memory being taken up by features that never get used.

- Construct the load for each customer such that only the features that the customer has bought and paid for are included in the load.

SPAM's ISE maintains a list of each customer and his requirements in terms of image and optional features. Nodules that are not required for a particular customer are not built into the load. This in itself is not always enough to implement an optional feature scheme. Conditional compilation is also required.

5.9.4 Conditional Compilation

Often a nodule that is in the core of the system will have to check for the presence of an optional feature at run time. Based on the absence or presence of the optional feature some special processing may or may not have to occur at run time. SPAM knows when the load is being compiled which features have to be built into the load. The feature information is made available to affected nodules as part of the connection information at compile time. Each nodule can interrogate the connections that are passed to it at compile time with preprocessor type statements to determine which features are being built into the load. The nodule uses conditional compilation to avoid generating code that is unnecessary. The check for the feature at run time is not required since the code gets optimized out at compile time. It is the responsibility of the programmer to ensure

that the conditional compilation facility is used where it is necessary.

5.9.5 Fixes

Individual nodules in the system will often undergo changes. The changes that an individual nodule undergoes is called a **fix**.

5.9.6 Summary of Flexibility Provisions

The SPAM system provides for system evolution with the following flexibility aspects:

- The system can have many images.
- Each image can accommodate many optional features.
- Each image can have many schema versions.
- Each nodule in each schema version of each image can have many fixes.

5.9.7 Provisions for Trackability

SPAM provides for the trackability of changes as the system evolves with the following scheme:

- Each image is numbered.
- Each feature is numbered.
- Each schema version of each image is numbered.
- Each nodule fix is numbered.
- Each change to the system is numbered and related back to the numbers of the image, schema, and nodule.

Each nodule is assigned a fix number. The fix number increases every time a change to the nodule is made. A nodule may be assigned a new fix number for one or more of the following reasons:

- An error was detected in the UPL code inside of the nodule. The change is isolated to the nodule itself.
- A change is made to the nodule to accommodate a new feature. In order to have the nodule accommodate the new feature a new fix number is assigned to the nodule.
- A change is made to the schema of the sub-system that the nodule resides in. In this case all nodules in the sub-system are assigned new fix numbers for the new version of the schema.
- A change is made to the image of which the nodule is a part. In this case the nodules within the image all get assigned new fix numbers, and the current schema version for the image is assigned a new schema version number.

The above list is not intended as a complete description of the trackability provisions of SPAM, rather, it is intended to give the reader the flavour of the trackability features. A structure that could be used to support this system organization is the MD-NT space, developed by Kimelman [32].

5.10 CONCLUSION

5.10.1 Summary

SPAM is a software system programming environment that provides a number of advantages over the traditional approaches to software system construction, namely:

- The ability to model a software system.
- The ability to evaluate a software design before the system has been implemented.
- The ability to specify the lifespan and visibility of data separately from the lexical organization of a program. This was called flexible scoping.
- The ability to integrate many software development lifecycle phases through automation.
- The ability to provide automatic unit and integration testing for a software system.
- The ability to impose management control and enforce conceptual integrity over the system by controlling access to the system schema.
- The flexibility to change the system as it evolves over time. As the system changes the design documentation and test case documentation are kept up to date.
- The ability to track individual changes to the system and relate them back to problem reports from the field.
- The ability to cross reference all data elements in the system without the use of a separate cross referencing tool. The cross reference is always up to date.

- Last, the ability to enforce a discipline which maintains that the perceived program complexity of a software system can be lowered by divorcing program data elements from program executable elements.

5.10.2 Analysis

This chapter has outlined the basic specifications and philosophical background of the software tool called SPAM. As well, the advantages of such a tool, were one to exist, have been enumerated. Obviously, there is a lot more work required before SPAM can be moved from the drawing board to the machine. The advantages of SPAM seem promising. Indeed, new techniques will have to be created if very large software projects are to be seriously attempted; this is the sort of research that America's SDI would encourage. This subject area is, and shall continue to be for some time, worthy of further research.

Chapter VI

CONCLUSION

"I pass with relief from the tossing sea of cause and theory to the firm ground of result and fact."

- Sir Winston Spencer Churchill

6.1 REVIEWING THE ORIGINAL OBJECTIVES

The original objective of this dissertation was to increase the productivity of the design stage of the software development lifecycle by increasing the degree of automation with tools and by refining human thought processes. Both of these objectives have been met. The human thought processes have been refined by the introduction of new terminology and heuristics for identifying module coupling and cohesion. Both module coupling and module cohesion can be represented precisely by the use of a quantitative metric rather than using the previous qualitative metrics. The degree of automation has been increased by the introduction of SPAM, a tool for modelling software system designs. The main advantage of SPAM is that the design phase of the SDLC is intrinsically integrated with other SDLC phases through automation. As well as providing integration for the SDLC, SPAM also changes the way that software engineers think about the workings of a software system since the programming language used deals with control elements only.

6.2 REVIEW OF RESULTS

In this section the results of each chapter of the dissertation are reviewed and examined with respect to meeting the original objectives.

6.2.1 Improved Terminology

In chapter 2 of this thesis the terminology associated with modular design was articulated. This is very important for improving the human thought processes associated with modular design. Often, the definition creates the meaning. In some instances new concepts were defined and new terminology was created to express the concepts. Some of the important concepts clarified in chapter 2 are:

- The definition of "module" and "modular design".
- The concept of a module interface and the associated data interfaces and control interfaces. Most importantly, a "connection" is defined to be a hypothetical path between modules over which information flows.
- The concepts of variable timing and variable visibility have been precisely defined and described as separate issues.
- The attributes that affect the complexity of data information have been identified and ordered.

6.2.2 Cohesion Factors

In chapter 3 of this thesis the factors that influence module cohesion were asserted and ordered. As a result of identifying and ordering the factors that influence module cohesion, the accepted levels of cohesion classification have been changed. These results affect the priority of different types of module cohesion, and hence the way that modules are coded.

6.2.3 Coupling Factors

In chapter 4 of this thesis the factors that influence module coupling were asserted and rigorously examined. As a result of identifying the factors that influence module coupling, a better understanding of good and bad module interfaces is gained. This affects the way that large systems are created and joined.

6.2.4 Software System Engineering

In chapter 5 of this thesis the ideas and concepts presented in the previous chapters are brought together into an integrated tool. The tool, called SPAM, enforces a discipline of the desirable concepts from the previous chapters. The result of bringing all of these ideas together in such an automated tool is a higher degree of automation and integrity in the software development lifecycle.

6.3 ADDITIONAL WORK

There are a number of distinct components that comprise SPAM, they are:

- The system schema, which is created and changed via the interactive schema editor (ISE).
- The underlying programming language (UPL).
- The interactive nodule editor (INE).
- The SPAM integrated compiler (SIC).

There would be a great deal of work involved in building a SPAM. It is advantageous to point out the components that could be adapted from existing systems; this would allow a working version of SPAM to be constructed expediently in order to demonstrate its feasibility.

There is a significant amount of work required to create a new programming language. Not only does the language grammar have to be defined, but a compiler for the language has to be written, debugged, and tested. There is no conceptual advantage to developing a unique programming language for the first version of SPAM. An existing programming language could be employed, by using only a subset of the language and by modifying the existing compiler. Indeed, since an existing language would be used, the advantages of SPAM would be highlighted.

The interactive nodule editor is basically a text editor that is capable of reading the system schema. In order to

get a SPAM running quickly a new INE would not have to be created. An existing text editor could easily be altered to include the presentation of all variable declarations. Once again, since an existing tool would be used, the conceptual advantage of SPAM would be highlighted.

The heart of SPAM is the ISE. By implementing an ISE and reusing all of the other components from existing systems, the advantages of SPAM can be demonstrated as expediently as possible. As well, the amount of time required to learn to use SPAM is reduced since users will already be familiar with many aspects of the tool.

Not all of the features need to be implemented in the first version of the ISE. For example, a skeleton ISE could be constructed that would not have the capability to:

- Draw data flow diagrams.
- Provide cross referencing capabilities.
- Provide automated unit testing of nodules.
- Provide support for features, images, versions, and fixes.

This would significantly reduce the time required to get the first version operational.

The author estimates that a skeleton version of SPAM could be constructed with the above limitations in less than two man years. The author estimates that it would take:

- Six man months of further analysis, requirements, and specifications.
- One man month for INE alterations.
- Two man months for SIC alterations.
- Nine man months to construct the skeleton ISE.

The above items total 18 man months. Allowing for things to go wrong, and they always do, the first version could be working within 24 man months. This is a very conservative estimate.

REFERENCES

1. APPLE Computer Inc. **Apple Pascal Language Reference Manual**, APPLE Computer Inc., 1980, Pages 72-82 (chapter 5). (Referenced: pp. 22, 165)
2. Basili, V.R., Weiss, D.M. "A Methodology for Collecting Valid Software Engineering Data", **IEEE Transactions on Software Engineering**, Volume SE-10, No. 6, November 1984. (Referenced: p. 148)
3. Batini, C., Nardelli, E., Tamassia, R. "A Layout Algorithm for Data Flow Diagrams", **IEEE Transactions on Software Engineering**, Volume SE-12, No. 4, April 1986. (Referenced: p. 170)
4. Beck, L.L., Perkins, T.E. "A Survey of Software Engineering Practice: Tools, Methods, and Results", **IEEE Transactions on Software Engineering**, Volume SE-9, No. 5, September 1983. (Referenced: p. 154)
5. **The Holy Bible**, Book of JOHN, Verse 3.3. (Referenced: p. 12)
6. Bell-Northern Research Technical Education Department. **Digital Technology in Telecommunications**, Bell-Northern Research Limited, 1977. (Referenced: p. 13)
7. Bell-Northern Research Technical Education Department. **Digital Switching Systems**, Bell-Northern Research Limited, 1978. (Referenced: pp. 1, 13, 158)
8. Bell-Northern Research Technical Education Department. **Telephony**, Bell-Northern Research Limited, 1979. (Referenced: pp. 1, 13)
9. Bell-Northern Research Software Tools Department. **PROTEL: A Procedure Oriented Type Enforcing Language for Telephony**, Bell-Northern Research, Division 7, Ottawa, 1980. (Referenced: pp. 13, 22, 127, 165)
10. Boehm, B. W. **Software Engineering Economics**, Prentice-Hall. ISBN 0-13-822122-7. 1981. (Referenced: pp. 3, 10)
11. Brooks, F.P. **The Mythical Man-Month**, Addison-Wesley, ISBN 0-201-00650-2, 1972. (Referenced: pp. 158, 161)

12. Buhr, P. E. **A Programming System**, Phd Thesis, University of Manitoba, 1985. (Referenced: pp. 22, 165)
13. Card, D. N., Church, V. E., Agresti, W. W. "An Empirical Study of Software Design Practices", **IEEE Transactions on Software Engineering**, Volume SE-12, No. 2, Feb. 1986. (Referenced: pp. 28, 102, 125, 134, 139, 140)
14. DeMarco, T. **Structured Analysis and System Specification**. 47th Proceedings of Guide International, 1978. (Referenced: p. 10)
15. DeMarco, T. **Controlling Software Projects: Management, Measurement and Estimation**, Yourdon Press, ISBN 0-917072-32-4, 1982. (Referenced: pp. 8, 157)
16. Dijkstra, E. "Go To Statement Considered Harmful [1968]". **Classics in Software Engineering**, Yourdon Press, ISBN 0-917072-14-6, 1979, Paper No. 3. (Referenced: p. 63)
17. Dijkstra, E. "Structured Programming [1969]". **Classics in Software Engineering**, Yourdon Press, ISBN 0-917072-14-6, 1979, Paper No. 5. (Referenced: p. 63)
18. Dijkstra, E.W. **A Discipline of Programming**, Prentice-Hall, ISBN 0-13-215871-X, 1976. (Referenced: p. 63)
19. Gane, C., Sarson, T. **Structured Systems Analysis: Tools and Techniques**, Mcdonnell Douglas, ISBN 0-930196-00-7, 1981. (Referenced: pp. 10, 32, 33, 166)
20. Goldberg, A., Robson, D. **Smalltalk 80: The Language and its Implementation**. Addison-Wesley, May 1983. (Referenced: pp. 22, 128, 165)
21. Habermann, A. N., Perry, D.E. **Ada for Experienced Programmers**, Addison-Wesley, ISBN 0-201-11481-X, 1983.
22. Harris, J. M. "The Factors That Influence Module Cohesion", **Congressus Numerantium Volume 51**, Utilitas Mathematica, Winnipeg, March, 1986.
23. Harris, J. M. "Pascal as a Teaching Language: Some Experiences From The Classroom", **Congressus Numerantium Volume 52**, Utilitas Mathematica, Winnipeg, May 1986.
24. Harris, J. M. "The Factors That Influence Module Coupling", **Congressus Numerantium Volume 58**, Utilitas Mathematica, Winnipeg, (accepted) December 1986.

25. Harris, J. M. "Elucidating The Definition of a Module", **Congressus Numerantium Volume 58**, Utilitas Mathematica, Winnipeg, (accepted) December 1986.
26. Harris, J. M. "SPAM: A Tool for Software System Engineering", **Congressus Numerantium Volume 58**, Utilitas Mathematica, Winnipeg, (accepted) December 1986.
27. Hopkins, M. E. "A Case For The GOTO [1972]". **Classics in Software Engineering**, Yourdon Press, ISBN 0-917072-14-6, 1979, Paper No. 9. (Referenced: p. 63)
28. Hornbach, K. **Real World use of Software Development Tools.**, Lear Siegler, Inc., 1985, Page 65. (Referenced: p. 12)
29. International Business Machines Corporation, **OS PL/I Checkout and Optimizing Compilers: Language Reference Manual**, International Business Machines Corporation, GC 33-0009-4, 1976. (Referenced: p. 52)
30. Jensen, R. W., Tonies, C. C. **Software Engineering**, Prentice-Hall, ISBN 0-13-822130-8, 1979. (Referenced: pp. 65, 122, 139)
31. Jones, C. B. **Software Development: A Rigorous Approach**, Prentice-Hall, ISBN 0-13-821884-6, 1980.
32. Kimelman, D. **Multidimensional Tree-Structured File Spaces**, Phd Thesis, University of Manitoba, 1986. (Referenced: p. 194)
33. Knuth, D. "Structured Programming with go to Statements [1974]". **Classics in Software Engineering**, Yourdon Press, ISBN 0-917072-14-6, 1979, Paper No. 20. (Referenced: p. 63)
34. Martin, J., McClure, C. **Software Maintenance: The Problem and its Solutions**, Prentice-Hall, ISBN 0-13-822361-0, 1983. (Referenced: p. 6)
35. Martin, J. **System Design from Provably Correct Constructs**, Prentice-Hall, ISBN 0-13-8811483-X, 1985 (Referenced: p. 5)
36. Martin, J., McClure, C. **Diagramming Techniques for Analysts and Programmers**, Prentice-Hall, ISBN 0-13-208794-4, 1985. (Referenced: p. 33)
37. Miller, G. A. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information [1956]". **Writings of The Revolution**, Yourdon Press, ISBN 0-917072-25-1, 1982, Paper No. 25. (Referenced: pp. 139, 162)

38. Mills, H. D. "Mathematical Foundations for Structured Programming[1972]". **Writings of the Revolution**, Yourdon Press, ISBN 0-917072-25-1, 1982, Paper No. 14.
39. Mitchell, J., et al. **Mesa Language Manual**, Xerox, Palo Alto, 1979. (Referenced: pp. 22, 127, 165)
40. Myers, G. J. "Composite Design: The Design of Modular Programs", **Technical Report No. TR00.2406**. IBM Corporation, Jan. 29, 1973. (Referenced: pp. 7, 97)
41. Myers, G. J. "Characteristics of Composite Design". **Datamation, Vol. 19, No. 9**, September 1973, pages 100-102. (Referenced: pp. 7, 97)
42. Stevens, W., Myers, G., Constantine, L. "Structured Design". **IBM Systems Journal, Vol. 13, No. 2**, International Business Machines Corporation, May 1974, Pages 115-139. (Referenced: pp. 7, 97)
43. Myers, G. J. **Reliable Software Through Composite Design**, Petrocelli-Charter, New York, ISBN 0-442-25620-5, 1975. (Referenced: pp. 2, 7, 19, 27, 30, 50, 59, 64, 65, 66, 66, 75, 97)
44. Myers, G. J. **Software Reliability Principles and Practice**, Wiley, New York, ISBN 0-471-62765-8, 1976. (Referenced: pp. 2, 7, 30, 63, 65, 75, 97)
45. Myers, G. J. **Composite Structure Design**, Van Nos Reinhold, ISBN 0-442-80584-5, 1978. (Referenced: pp. 2, 7, 19, 27, 27, 30, 50, 59, 64, 65, 65, 66, 66, 75, 97)
46. Myers, W., "Can Software for the Strategic Defense Initiative ever be error-free?", **IEEE COMPUTER** November 1986, page 66. (Referenced: pp. 7, 7, 156)
47. Olsen, E.W., Whitehill, S.B. **Ada for Programmers**, Reston Publishing Company Inc., ISBN 0-8359-0149-1, 1983. (Referenced: pp. 22, 165)
48. Pascoe, G.A. "Elements of Object-oriented Programming", **BYTE Magazine, August 1986, Volume 11, Number 8**, McGraw Hill, 1986, Pages 139 to 144. (Referenced: pp. 22, 165)
49. Parnas, D. L. "On the Criteria to Be Used in Decomposing Systems into Modules". **Communications of the ACM**, volume 15, no. 12, pages 1053-1058, Dec. 1972. (Referenced: p. 25)
50. Paul, R. J. "An Introduction to Modula-2". **Byte Magazine**, McGraw-Hill, August 1984, pages. 195-210. (Referenced: pp. 22, 165)

51. Peters, L.J. **Software Design: Methods and Techniques**, Yourdon Press, ISBN 0-917072-19-7, 1981. (Referenced: p. 3)
52. Shooman, M. L. **Software Engineering: Design, Reliability, and Management**, McGraw-Hill Company, ISBN 0-07-057021-3, 1983. (Referenced: pp. 18, 139, 166)
53. Swanson, E. "The Dimensions of Maintenance". **2nd International Conference on Software Engineering, Proceedings, October 13-15, 1976, Pages 492-471.**
54. United States Department of Defense. **The programming Language Ada**, Springer-Verlag, ISBN 3-540-10693-6, 1980. (Referenced: pp. 22, 165)
55. Ward, P.T., "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing", **IEEE Transactions on Software Engineering**, Volume SE-12, No. 2, Feb. 1986. (Referenced: p. 170)
56. **Webster's Ninth New Collegiate Dictionary**. ISBN 0-919028-66-7, Merriam-Webster, 1983. (Referenced: pp. 2, 8, 50, 159)
57. Weiss, D. M., Basili, V. R. "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory". **IEEE Transactions on Software Engineering, Volume SE-11 Number 2**, IEEE Computer Society, ISSN 0098-5589, 1985, Pages 157-168. (Referenced: p. 154)
58. Whitehead, A.N. **An Introduction to Mathematics**, Oxford University Press, 1959.
59. Wirth, N., Jensen K. **Pascal User Manual And Report (second edition)**, Springer-Verlag, ISBN 0-387-90144-2, 1974.
60. Wirth, N. **Programming in MODULA-2**, Springer-Verlag , ISBN 3-540-11674-5, 1982. (Referenced: pp. 22, 165)
61. Wirth, N. "History and Goals of Modula-2". **BYTE Magazine, August 1984, Volume 9, Number 8**, McGraw-Hill, 1984, Pages 145 to 152. (Referenced: pp. 22, 165)
62. Wulf, W. A. "A Case Against The GOTO [1972]". **Classics in Software Engineering**, Yourdon Press, ISBN 0-917072-14-6, 1979, Paper No. 8. (Referenced: p. 63)
63. Yau, S., Tsai, J.J.P. "A survey of Software Design Techniques" **IEEE Transactions on Software Engineering**, Volume SE-12, No. 6, June 1986. (Referenced: pp. 154, 157)

64. Yourdon, E. **Techniques of Program Structure and Design**, PRENTICE-Hall, ISBN 0-13-901702-X, 1975.
(Referenced: p. 21)
65. Yourdon E., Constantine, L.L. **Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design**, Prentice-Hall, ISBN 0-13-854471-9, 1979. (Referenced: pp. 20, 30, 44, 46, 64, 65, 66, 75, 75, 75, 97, 139, 166, 175)

INDEX OF DEFINITIONS

abstract, 59
aggregate, 47
atomic, 51

bi-directional connections,
 31
birth, 35
BNR, 12

CD(:), 143
CD(D:G), 142
CD(G:), 143
complete formal data
 interface mechanism, 42
completely synchronized, 40
complex, 50
composite data interface, 44
connection, 31
Control Information, 41
control interface, 45

data aggregate, 47
data exclusion, 82
Data Information, 41
data interface, 42
death, 36
Deployment, 4
deposit, 33
Design, 4
DFD, 166
DMS (Digital Multiplexed
 Switch), 12

exclusivism, 82
executable range, 39
explicit connections, 31

factor precedence, 145
factor weight, 144
feature, 175
features, 191
first order dependencies,
 115
fix, 193
flexible scoping, 167
formal interface mechanism,
 42

FP, 145
functional exclusion, 82
FW, 144
FWPM, 145

heterogeneous complexity, 51
homogeneous complexity, 51
Hybrid Information, 41

ideal data interface, 43
image, 175, 190
implicit connections, 31
improperly synchronized, 40
incomplete formal data
 interface mechanism, 43
INE, 187
informal data interface
 mechanism, 43
interactive nodule editor,
 187
Interactive Schema Editor,
 168
interface, 42
ISE, 168

lifespan, 36
linear complexity, 58

Maintenance, 4
Modular design, 29
module, 26
module cohesion coefficient,
 85
molecular complexity, 58

nodules, 171
null coupling, 112

order dependency, 54

parameters, 79
pathological data interface,
 44
portability, 182
predictor metrics, 157
Programming, 4
properly synchronized, 40

- pure control information, 41
- qualitative metrics, 8
- quantitative metrics, 8
- receiving module, 33-34
- regression testing, 182
- Requirements, 3
- result metrics, 157
- reuseability, 182
- schema versions, 191
- scope of control, 108
- scope of effect, 108
- scope rules, 38
- SD, 166
- SDIO, 156
- SDLC, 3
- segment, 78
- sending module, 33-34
- SIC, 188
- simple, 50
- Software Development Lifecycle, 3
- SPAM integrated compiler, 188
- Specifications, 4
- SS(top), 174
- SS(0), 174
- Strategic Defense Initiative Organization, 156
- structure information, 55
- sub-system, 173
- system schema, 10, 170
- temporally coincidental, 36
- temporally permanent, 37
- temporally persistent, 37
- temporally synchronized, 36
- Testing, 4
- time dependencies, 82
- transform analysis, 166
- Underlying Programming Language, 183
- uni-directional connections, 32
- uni-directional deposit connection, 33
- uni-directional withdrawal connection, 33
- UPL, 183
- usage, 41
- visibility rules, 38
- visibly coincidental, 40
- visibly contained, 39
- visibly synchronized, 39
- Waterfall Model, 3
- withdrawal, 33