สู่มาตรวัดความซับซ้อนปริชานซอฟต์แวร์ด้วยโครงสร้างแตกชั้นของปัจจัยรวม

นาย เบญจพล ออประเสริฐ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2552

TOWARDS SOFTWARE COGNITIVE COMPLEXITY MEASURE WITH GRANULAR STRUCTURES OF UNIFIED FACTORS

Mr. Benjapol Auprasert

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Engineering Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2009

Thesis Title	TOWARDS SOFTWARE COGNITIVE COMPLEXITY MEASURE WITH GRANULAR STRUCTURES OF UNIFIED FACTORS
By	Benjapol Auprasert
Field of Study	Computer Engineering
Advisor	Assistant Professor Yachai Limpiyakorn, Ph.D

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

..... Dean of the Faculty of Engineering

(Associate Professor Boonsom Lerdhirunwong, Dr.Ing.)

THESIS COMMITTEE

..... Chairman

(Professor Boonserm Kijsirikul, D.Eng.)

..... Advisor

(Assistant Professor Yachai Limpiyakorn, Ph.D)

..... Examiner

(Assistant Professor Vishnu Kotrajaras, Ph.D)

..... External Examiner

(Paskorn Apirukvorapinit, Ph.D)

เบญจพล ออประเสริฐ : สู่มาตรวัดความซับซ้อนปริชานซอฟต์แวร์ด้วยโครงสร้างแตกชั้นของปัจจัยรวม. (Towards Software Cognitive Complexity Measure with Granular Structures of Unified Factors) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ผศ.ดร. ญาใจ ลิ่มปิยะกรณ์ :103 หน้า.

มาตรวัดความซับซ้อนปริชานวัดความยากง่ายสำหรับสมองมนุษย์ในการทำความเข้าใจซอฟต์แวร์โดยอาศัยหลักการพื้นฐานทาง Cognitive Informatics ซึ่งวัดความซับซ้อนปริชานจากปัจจัยพื้นฐานของซอฟต์แวร์ ได้แก่ อินพุต เอาต์พุต และ โครงสร้างประมวลผลภายใน ภายหลังจากการที่ได้มีการนำเสนอ Cognitive Functional Size (CFS) ขึ้น งานวิจัยหลายงานได้พยายามเพิ่มเติมและดัดแปลง CFS ให้คำนึงถึงปัจจัยพื้นฐานให้ครบถ้วนสมบูรณ์ขึ้น เช่นคำนึงถึงปริมาณสารสนเทศในรูปของ identifiers และ operators งานวิจัยเหล่านี้ พยายามที่จะประเมินความซับซ้อนจากหลายปัจจัย แต่กลับประเมินค่าความซับซ้อนจากแต่ละปัจจัยแยกจากกัน โดยไม่คำนึงถึงความสัมพันธ์เกี่ยวเนื่องกันของปัจจัย วิทยานิพนธ์นี้จึงนำเสนอวิธีการใหม่โดยนำหลักการจาก Granular Computing เข้ามาประยุกต์ใช้และนำเสนอมาตรวัดใหม่เรียกว่า Structured Cognitive Information Measure (SCIM) ซึ่งรวมปัจจัยและจัดโครงสร้างใหม่ให้สอดคล้องกับกระบวนการปริชานของมนุษย์ มีการทดลองเบื้องต้นเพื่อประเมินวิธีที่นำเสนอนี้ รวมถึงมีการประเมินผ่านคุณสมบัติของ Weyuker ทั้ง 9 ข้อ วิทยานิพนธ์ยังได้สนับสนุนความเป็นสากลกับโดเมนทั่วไปของหลักการ Granular Computing นอกจากนี้ยังได้มีการเสนอกรอบงานอุปนัยเพื่อแก้ไขข้อบกพร่องของคุณสมบัติของ Weyuker และนำมาใช้ประเมินและวิเคราะห์มาตรวัดความซับซ้อนปริชานซอฟต์แวร์ต่างๆ เพื่อหาจุดอ่อน จุดแข็ง และแนวทางในการพัฒนามาตรวัดต่างๆต่อไป

ภาควิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่อนิสิต เบญจพล ออประเสริฐ
สาขาวิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่ออ.ที่ปรึกษาวิทยานิพนธ์หลัก
ปีการศึกษา 2552

5170365021 : MAJOR COMPUTER ENGINEERING

KEYWORDS : COGNITIVE COMPLEXITY MEASURE / GRANULAR STRUCTURE / INDUCTIVE FRAMEWORK / UNIFIED AND STRUCTURED FACTORS / WEYUKER'S PROPERTIES

BENJAPOL AUPRASERT : TOWARDS SOFTWARE COGNITIVE COMPLEXITY MEASURE WITH GRANULAR STRUCTURES OF UNIFIED FACTORS. ADVISOR : ASST.PROF YACHAI LIMPIYAKORN, Ph.D, 103 pp.

Cognitive complexity measures quantify human difficulty in understanding the source code based on cognitive informatics foundation. The discipline derives cognitive complexity on a basis of fundamental software factors i.e. inputs, outputs, and internal processing architecture. The invention of Cognitive Functional Size (CFS) stands out as the breakthrough to software complexity measures. Several subsequent researches have tried to enhance CFS to fully consider more factors, such as information contents in the form of identifiers and operators. However, these existing approaches quantify the factors separately without considering the relationships among them. This thesis presents an approach to integrating Granular Computing into the new measure called Structured Cognitive Information Measure or SCIM. The proposed measure unifies and re-organizes complexity factors analogous to human cognitive process. Empirical studies were conducted to evaluate the virtue of SCIM, including theoretical validation through nine Weyuker's properties. The universal applicability of granular computing concepts is also demonstrated. Additionally, the new inductive framework has been proposed to patch the holes of Weyuker's properties, and used in the assessment of the cognitive complexity measures to analyze and guide directions for future improvement of the measures.

Department : Computer Engineering Student's Signature Benjapol Auprasert

Field of Study : Computer Engineering Advisor's Signature

Academic Year : 2009

Acknowledgements

There are so many people I would like to heartily say thank you for the ongoing and enthusiastic support throughout this whole experience of thesis writing and beyond. Completing this thesis is such a moment, and the moment is so much bigger than me. This is for Asst.Prof Yachai Limpiyakorn, my advisor, who always believes in me and has guided me through this rollercoaster-ride journey, despite some up and down times, in pursuing the goals and dreams until we have so far achieved publishing papers in four international conferences proceedings. It is for Prof Yinxu Wang, who started this whole cognitive complexity measure idea and organized such grand and ongoing conferences like ICCIs. It is for Prof Mark Burgin, who supported the idea and gave me some good advices at the CSIE conference. It is for Prof Boonserm Kijsirikul, Asst.Prof Vishnu Kotrajaras, and Ajarn Paskorn Apirukvorapinit, whose advices, criticisms, and questions during the proposal exam and beyond got me to re-think and improve the work in this thesis. It is for SIPA, who funded this Software Quality Research and Development Project. And it is for anyone trying to pursue some great researches that now the door has opened for you to achieve your dreams if you try hard enough and believe in what you are doing.

This thesis would not have been a success without the amazing support structures surrounding me by my family and friends, my father, Suparat Auprasert, and my mother, Assoc.Prof Kanya Auprasert, who encouraged me to pursue the degree and taught me to be strong and believe in what I was doing. My friend, Jenjira Wongboonsin, who has been through this amazing experience like me and helped me all along, my ex-supervisor, Ekraj Wongkiatkachorn, who made it possible for me to pursue my degree, and all my colleagues, and the friends I met during the conferences and the trips, all the people are so wonderful. I never thought that I would have met people who are so great, cool, and so nice to me during this whole journey. This thesis is dedicated to all the people who came into my life, who I cannot say all their names. They all gave me the faith and courage to hew down the mountain of despair into stepping stones of hope, until I have finally completed this thesis. I really appreciate them and would like to say thank again for everything they gave to me. This whole experience, all the people that I met, all the places that I have been through, all the memories that I have got, will always stay with me. Thank you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Software complexity measurement has been an age-long problem in software engineering as the effort used to develop, comprehend, or maintain the software depends on so many complicated factors. In the past, the measures tended to consider only some perspectives particular to each research approach. For example, McCabe's Cyclomatic Number [1] only considers the control flows of the program, while Halstead's Metrics [2] only take into account the number of operators and operands.

As the software industry grows more complex, trends in software engineering are turning from empirical studies towards schema-based studies [3]. Cognitive Informatics [4],[5],[6] is a multidisciplinary study of the internal information processing mechanisms of the brain and the processes involved in perception and cognition. With Cognitive Informatics emerging as a promising discipline during the past few years, cognitive complexity of software has opened up the potential for a new research area of software metrics. Cognitive complexity measures attempt to quantify the degree of difficulty or effort spent on comprehending software from all the perspectives e.g. loops and branches, data objects such as inputs, outputs, and variables, so that the complexity value more precisely reflects the difficulty for human brain to process and comprehend the software. The measures can thus be used to predict quality dimensions such as testability, reliability, and maintainability of software systems. However, such metrics are still in a very nascent stage, as evaluating complexity from many factors may not be convincing if the factors are not carefully thought and organized.

Previously proposed cognitive complexity measures [7],[8],[9],[10],[11], [12] evaluate the complexity of software based on factors e.g. basic control structures (BCS's), inputs/outputs, and information in the form of identifiers and operators, separately, and then assemble the complexity value by the weights derived from each

factors. This process of calculation overlooked the dependency among factors, while in fact, these factors have relationships with each other and should not be evaluated separately. Therefore, we suggest that the factors need be unified and organized into a structure, and then the complexity of the 'whole' software program should be evaluated by complexity of its 'parts' and the interrelationships among 'parts' and 'whole'. This seems to fit the concept of 'Granular Structures' [13] used in granular computing, which has recently been suggested as a generic method that can be extended to broad areas of problem-solving [13].

This research has reviewed and revealed the flaws of existing cognitive complexity measures. Another approach for cognitive complexity measurement is proposed by arranging complexity attributes into granular structures analogous to when human comprehends the software, in order to make the blooming discipline of cognitive complexity measurement [4],[5],[7] a more mature and sound discipline.

1.2 Objectives

The objective of this thesis is to make the emerging discipline of software cognitive complexity measurement more mature and justified by:

1) Deriving a new measure that is theoretically not inferior to the existing ones.

2) Proposing a framework for evaluating and improving the assessment of cognitive complexity measures.

1.3 Scopes and Initial Agreements

The scopes of this thesis as agreed in the proposal are as follows:

- Derive a new cognitive complexity measure that is not inferior to the existing ones (i.e. CFS, MCCM, CPCM), at least in terms of Weyuker's properties.

- Invent a new sound framework for evaluating cognitive complexity measures, and use it to analyze and find logical strengths and weaknesses of each measure.

- Empirical study of the proposed cognitive measure is conducted on at least 5 programs of which the size is not less than 100 lines of code.

1.4 Limitations

- As current studies in Cognitive Informatics disciplines tend towards schema-based studies rather than empirical studies, and studies of software cognitive complexity measurement in Cognitive Informatics are in a nascent stage, this thesis focuses more on schema-based or theory-based studies, rather than empirical or practical studies. Therefore the proposal is rather conceptual than implementation-focused. The validation of the proposed measures in this thesis also focuses more on theoretical than empirical.

- The measure proposed works better on logical programs, and is not suitable for evaluating programs with some specific characteristics, e.g. file-processing programs. This is because some cognitive weights of basic control structures (BCS's) proposed in [7] are still not defined and calibrated well when it comes to BCS's with more complex processing.

1.5 Acronyms

BCS	Basic Control Structure
CC	McCabe's Cyclomatic Number
CFS	Cognitive Functional Size
CICM	Cognitive Information Complexity Measure
CPCM	Cognitive Program Complexity Measure

DF	Data Flow Complexity
GUI	Graphic User Interface
HE	Halstead's Effort
ICN	Informatics Complexity Number
I(L)	Information contained in leaf node granule 'L'
I/O	Input / Output
LOC	Line of Codes
MCCM	Modified Cognitive Complexity Measure
SCIM	Structured Cognitive Information Measure

1.6 Expected Benefits

- The emerging discipline of cognitive complexity measure becomes more mature and justified.

- The direction of how to evolve the cognitive complexity measures into a more precise ones become clearer.

1.7 Research Methodology

This research aims to review and point out the flaws of existing cognitive complexity measures, and then propose a new solution for cognitive complexity measurement by arranging complexity attributes into granular structures analogous to when human comprehends the software, in order to make the blooming discipline of cognitive complexity measurement a more mature and sound discipline.

Instead of evaluating each factors e.g. basic control structures, number of identifiers/ operators separately, the principle of granular structures for problem-solving [13],[14],[15],[16] is applied to the domain of software cognitive complexity.

Based on this approach, the newly proposed measure unifies the factors and re-organizes them into granular hierarchical structures analogous to how human comprehends the program code. In other words, we unify and group factors by their relationships and dependencies instead of grouping them by types (e.g. basic control structures, identifiers, operators, inputs, or outputs)

The proposed measure solves three major problems of existing cognitive complexity measures. One is the lack of consideration of information content as regarded as CFS's drawback. Another is the ignorance of the detailed relationships between some factors, e.g. the data objects and the basic control structures as appeared in CICM, MCCM, and CPCM. The other is the irrational use of addition (+) and multiplication (*) in the computation of Wc, as rationaled in [17], which also suggested that this problem leads to the lack of Property 6 (($\exists$ P)($\exists$ Q)($\exists$ R)((|P|=|Q|) & (|P;R| $\neq$ |Q;R|))) of Weyuker's.

The newly proposed cognitive complexity measure is theoretically evaluated against Weyuker's properties [18], as well as the Framework for Evaluating Metrics [19]. The new inductive framework for evaluating cognitive complexity measures is also proposed as guidelines for improving the assessment of the measures. The proposed measure is also evaluated with this new framework, which focuses on what happens to complexity values when two programs are concatenated in various conditions. Finally, the empirical study of the effectiveness of the proposed measure compared to the selected cognitive complexity measures is conducted on the case studies.

1.8 Research Publication Progress

Research papers which are parts of this thesis have been published in four international conference proceedings. The details of the papers and the conferences are as follows:

- The full paper that covers the whole content of the thesis, “Towards Structured Software Cognitive Complexity Measurement with Granular Computing Strategies” [20], was presented at the 8th IEEE International Conference on Cognitive Informatics (ICCI 2009) at Hong Kong Polytechnic University on June 16th, 2009, and was published in the Proceedings of the 8th IEEE International Conference on Cognitive Informatics (ICCI 2009), Hong Kong, June 15 - 17, 2009, pp 365-370.

- The paper “Representing source code with granular hierarchical structures” [21], which is part of the thesis, was accepted for the poster track and published in the Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC 2009), Vancouver, Canada, May 17 - 19, 2009, pp 319-320.

- The paper “Structuring Cognitive Information for Software Complexity Measurement” [22], which is part of the thesis, was presented at the 2009 World Congress on Computer Science and Information Engineering (CSIE 2009) at Wilshire Grand Hotel, Los Angeles, on April 2nd, 2009, and was published in the Proceedings of 2009 World Congress on Computer Science and Information Engineering (CSIE 2009), IEEE CPS, Los Angeles, USA, March 31 - April 2, 2009, pp 830-834.

- The paper “Underlying Cognitive Complexity Measure with Combinatorial Rules” [17], which is part of the thesis, was presented at WCSET 2008: World Congress on Science, Engineering and Technology, at Holiday Inn Paris, France, on November 23rd , 2008, and was published in the Proceedings of World Academy of Science, Engineering and Technology, Volume 45, November 2008, ISSN: 2070-3740, pp 432-437.

In addition, the paper “Towards Inductive Framework for Evaluating Software Cognitive Complexity Measures”, which is part of the thesis, is currently under the process of submitting to publication.

Chapter 2

Literature Reviews

2.1 Classical Software Complexity Measures

Since the late 1960's, several complexity metrics have been proposed to measure the complexity of software. Each of them dealt with different aspects and problems in evaluating complexity. Very soon, many metrics were found not useful and faded from the industry. Still, the metrics that remain widely accepted and used in the industry, namely LOC (Lines of Codes), McCabe's cyclomatic complexity, Halstead's metrics, Data flow complexity, and Function Point, are, however, found to be far from perfection, nor satisfy the needs in the industry to rigorously evaluate and compare the complexity of the software.

2.1.1 The Physical Size (Lines of Code - LOC)

The physical size, or lines of code (LOC), is the simplest and most-used software measure. It simply counts the number of lines in the source code. The recommended counting standard is to count any lines but blank lines, section separations, and comments [7].

LOC counter is easy to implement. However, it may not reflect the true complexity, nor indicate the effort used in a software project, as it does not take into account some complicated details of software, like the difficulty of algorithms and program structures. It is also dependent on languages and programmers' skills. Moreover, it seems to cause inefficient implementation in coding

2.1.2 McCabe's Cyclomatic Complexity (CC)

In 1976, McCabe [1] developed the cyclomatic complexity that counts the number of linearly-independent paths through a software component which is transformed into a connected graph showing the topology of control flow within the program. The measurement value is calculated from:

Cyclomatic complexity (CC) = E – N + p

where E = the number of edges of the graph, N = the number of nodes of the graph, and p = the number of connected components.

CC is independent of languages and language format. It provides a single ordinal number that can be compared to the complexity of other programs. However, it considers only the loops and branches, without the consideration of other factors such as the length of codes or number of statements that could affect the ability of program comprehension.

2.1.3 Halstead's Software Metrics

In 1977, Halstead [2] proposed a set of software metrics for measuring the algorithmic complexity by counting operators and operands from software codes.

Let n_1 = number of distinct operators

n_2 = number of distinct operands

N_1 = total number of operator occurrences

N_2 = total number of operand occurrences

Halstead's measures are then defined as shown in Table 1.

Table 1. Definition of the Derived Measures of Halstead's Software Metrics

Measure	Symbol	Formula
Program length	N	$N = N_1 + N_2$
Program vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N*(\log_2 n)$
Estimated abstraction level	L	$L= (2 n_2) / (n_1*N_2)$
Difficulty	D	D= 1 / L

Effort	E	E = V * D
Time	T	T = E / 18
Remaining bugs	B	$B = E^{2/3} / 3000$

Halstead's measures can be considered as an all-in-one tool for estimating project parameters, as they provide the formulae for many important attributes of a software project. However, they are language dependent, and they do not take into account the factors that intensify the complexity of the software, such as loops, branches, function calls, interrupts, etc. Moreover, some of the constants, e.g. T = E / 18, $B = E^{2/3} / 3000$, are based on pure observations and experiments, while lacking the rational support. The physical meanings of the measures are not clear either.

2.1.4 Oviedo's data flow complexity measure (DF)

Oveido's measure [23] is a different approach based on the data flow characteristics of the program. The measure decomposes a program into a set of disjoint blocks of ordered statements having the property that the first statement of the block is the only statement which can be executed directly after the execution of a statement in another block and whenever it is executed, the other statements in the block are executed in the given order. In other words, the block is a chunk of code that is always executed as a unit. The blocks are then used to construct a program flow graph, a directed graph in which each node corresponds to a block of the program and there exists the edge from node i to node j if and only if it is possible for control to transfer directly from block i to block j in the program. DF is defined as followed:

A variable definition takes place when the variable is defined.

A variable reference takes place when the variable is used in the program.

A locally available variable definition for a program block is a definition of the variable in the block. A locally exposed variable reference in a block is a reference to a variable which is not preceded in the block by a definition of that variable.

A variable definition in block i is said to reach block j if the definition is locally available in block i and there is a path from i to j along which the variable is not locally available in any block on the path, i.e. the variable is not redefined along that path.

Data flow complexity (DF) of a program is defined as the sum of data flow complexity of all blocks in the program. Data flow complexity of block i (DF_i) is computed from all variable v_j in the set of variables whose references are locally exposed in block i:

$$DF_i = \sum DEF(v_j)$$

where $DEF(v_j)$ is the number of available definition of variables v_j in the set of variable definitions that reach block i .

DF solves the problem of other measures that the relationships between blocks of software code are not taken into account when computing complexity size. However, it is still a rough measure based on the assumptions that programmers can determine the definition-reference relationships within blocks more easily than between blocks, and number of different variables locally exposed in each block is more important than the total occurrences of that variable references. Hence, it ignores the fact that the frequent occurrences of variables can also make the program more complex. Because it only considers the complexity transferred between blocks, not considering the complexity within the blocks, it lacks an important fundamental property of software measure that the additional code into any program code should not decrease the complexity of that program code as proved by Weyuker [10].

2.1.5 Function Points (FP)

In 1983, Albrecht [24] developed the function point metric as a measurement to express the amount of business functionality that an information system provides to users. Function points (FPs) can be determined by “a function of the number of inputs, outputs, data objects, and internal processes.” [7]

FP = UFP * TCF

where UFP (Unadjusted function points) is a weighted sum of numbers of function items: (Note that the average weights are used here.)

UFP = 4(#external inputs) + 5(#external outputs) + 10(#internal logic files) + 7(#interface files) + 4(#external inquiries)

and TCF (Technical correction factors) is a weighted sum of 14 affective degrees of General System Characteristics (GSKs) including data communications, distributed data processing, performance, heavily used configuration, transaction rate, on-line data entry, end -user efficiency, on-line update, complex processing, reusability, installation ease, operational ease, multiple sites, and modifiability :

TCF = 0.65 + 0.01 $\sum$(affective degree of each GSK)

where the affective degree ranges from 0 (none) to 5 (essential). As a result, TCF falls in the range of 0.65-1.35.

“Function Points” is a measure that takes into account many factors affecting software complexity, making it a metric that can estimate the effort and productivity quite accurately. However, due to many factors, Function Points has been criticized as adding little value relative to the cost and complexity of the effort used to conduct the measurement. The method also requires subjectively weighing of the GSKs.

2.2 Cognitive Complexity Measures of Software

Cognitive Informatics is the multidisciplinary study of the internal information processing mechanisms of the brain and the processes involved in perception and cognition [4]. Main ideas in Cognitive Informatics Research that is associated with software can be summarized as follows:

- Software is a mathematical entity, a coded solution, and a special type of information at a certain abstract level, i.e. level 3 – special notation systems or level 4 – mathematics [4],[5],[6].

- Program comprehension is a cognitive process to understand a software system in terms of architecture, static behaviors and dynamic behaviors [6].

- Cognitive complexity of software is dependent on three fundamental factors: internal processing and its input and output [5].

- Cognitive informatics process takes into account the information contained in software [5],[9].

Cognitive complexity of software is based on these principles. The research in this field, including this research is based on the fundamental principles that software is information and cognitive complexity of software is dependent on inputs, outputs, and internal processing architecture.

2.2.1 Cognitive Functional Size (CFS)

Referring to the Cognitive Informatics foundation that cognitive complexity of software depends on three fundamental factors: inputs, outputs, and internal processing, Cognitive Functional Size (CFS) [7],[8] was proposed by Wang in 2003 as:

$$CFS = (N_i + N_o) * W_c$$

where N_i = #inputs, N_o = #outputs, and W_c is the total cognitive weight of basic control structures (BCS's), representing internal processing, defined as the total sum of cognitive weights of its q linear blocks composed in individual BCS's. Since each block may consist of 'm' layers of nesting BCS's, and each layer with 'n' linear BCS's, then

$$W_c = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right]$$

where weights W_c (j,k,i) of BCS's were defined as presented in Table 2.

Table 2. Cognitive Weights (W_c) of BCS's

Category	BCS	W_c
Sequence	Sequence (SEQ)	1
Branch	If–Then-Else (ITE)	2
	Case (CASE)	3
Iteration	For-do (R_j)	3
	Repeat-until (R_1)	3
	While-do (R_0)	3
Embedded Component	Function call (FC)	2
	Recursion (REC)	3
Concurrency	Parallel (PAR)	4
	Interrupt (INT)	4

Although CFS is easy to implement and independent from implementation technologies, it excludes some details of potential cognitive complexity factors like information contents in the forms of identifiers and operators contained in the internal architecture as suggested by the informatics laws of software [9],[10],[11]. This drawback of CFS leads to the proposal of many subsequent measures [9],[10],[11],[12] attempting to solve this problem.

This thesis relies on the CFS's concept and uses its originally proposal of the cognitive weights of BCS. CFS is selected to be compared with the proposed measure in this work.

2.2.2 Cognitive Information Complexity Measure (CICM)

Early 2006, Kushwaha and A.K. Misra [9],[10] modified Wang's CFS to measure the information contained in software. They referred to the law of informatics [6] that software ≈ information, thus the difficulty in understanding software ≈ the difficulty in understanding information. Moreover, since software is a mathematical entity that represents computational information, the amount of information contained in software is a function of identifiers that hold the information and operators that perform the operations on the information. Hence:

Information = f (Identifiers, Operators).

The Cognitive Information Complexity Measure (CICM) was then defined as:

$$CICM = WICS * W_c$$

where W_c is defined the same as in CFS, and WICS is the weighted information count of the software derived from:

$$WICS = \sum_{k=1}^{LOCS} \{ n(k) / (LOCS-k) \}$$

where n(k) is the number of identifiers and operators in the $k^{th.}$ line.

This thesis exploits the CICM's concept that the amount of information contained in software is the function of identifiers and operators. Although the idea to count identifiers and operators as the information contained in software, as in CICM, is reasonable, the weight function is meaningless and requires more rationales. The weighing technique was just a bogus function to make it conform to the fact that identifiers and operators become more and more difficult to understand as we advance into the later lines of the program.

Furthermore, for CFS, it is quite acceptable to multiply the number of inputs and outputs by W_c, because inputs can be considered as external objects that are processed through the whole control architecture that is derived into W_c, resulting in outputs from the system, as shown in Figure 1. In contrast, for CICM, identifiers and operators are not contained in all BCS's in the architecture, but inside some specific layers of the BCS, as shown in Figure 2. Therefore, the weighted count of identifiers and operators should not be simply multiplied by the W_c derived from the whole BCS's architecture without considering the specific position of each identifier or operator in the BCS's

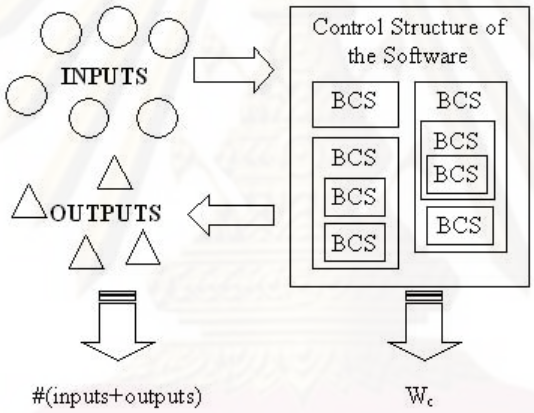


Figure 1. Relationships between I/Os and the whole BCS's as in CFS

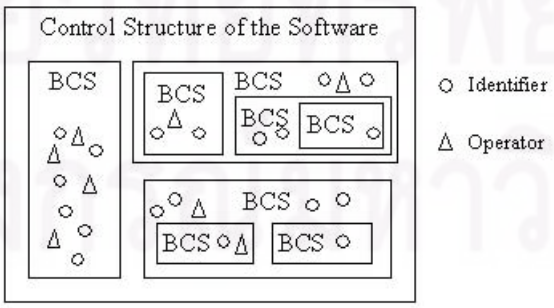


Figure 2. Identifiers/operators not affected by all BCS's in the structure

2.2.3 Modified Cognitive Complexity Measure (MCCM)

Later in 2006, S. Misra [11] enhanced CFS into Modified Cognitive Complexity Measure (MCCM), which was formulated as:

$$MCCM = (N_{i1} + N_{i2}) * W_c$$

where N_{i1} is the total number of occurrences of operators, N_{i2} is the total number of occurrences of operands, and W_c is defined the same as in CFS.

MCCM simplifies the complex weighted information count of each line in CICM by quantifying information contents in terms of number of occurrences of operators and operands. However, the multiplication of information content with the weight W_c derived from the whole BCS's structure remains the approach's drawback.

Due to the too complicated calculation of CICM, this thesis considers MCCM as the measure of the same type as CICM, and uses MCCM as the representative of both approaches in the comparative case studies, to enable more practical evaluation.

2.2.4 Cognitive Program Complexity Measure (CPCM)

In 2007, S. Misra proposed the Cognitive Program Complexity Measure (CPCM) [12] based on the arguments that the occurrences of inputs and outputs in the program directly affect the internal architecture and they can be considered as the forms of information contents. He also criticized the computation of CFS that the multiplication of distinct number of inputs and outputs with the total cognitive weights was not justified as there was no reason why using multiplication. Furthermore, he claimed that operators are run time attributes and cannot be regarded as information contained in the software as proposed by CICM. Based on these arguments, CPCM was thus defined as

$$CPCM = S_{io} + W_c$$

where S_{io} is the total occurrences of input and output variables, and W_c is defined as in CFS.

In CPCM, the assumption that operators should not be considered as information contained in software is reasonable, as operators only perform operations on information stored in identifiers such as variables. However, similar to CICM and MCCM, CPCM does not consider dependencies between variables and their positions in the BCS's architecture. Additionally the use of addition (+) instead of multiplication (*) is not fully justified, as it implies that S_{io} and W_c are distinct factors of complexity.

This thesis applies CPCM ideas that operators are disregarded as information contained in software. CPCM is selected to be compared with the proposed measure in this work.

2.3 Other Associated Theories and Researches

2.3.1 Unified Framework of Granular Computing

Granular computing is an interdisciplinary study of human-inspired computing [13]. Recently, studies in granular computing previously dominated by set-theoretic models have been re-casted in a wider context outside data mining, which is their original domain. Studies in granular computing include studying: the contexts of structured writing, structured proof, structured programming, or information processing [13],[16]. The unified framework of granular computing [13] extracts high-level common principles from a wide range of scientific disciplines involving human problem solving methodologies, and studies them in a uniform way [13],[14],[15], by partitioning the universe into multilevel and multi-view granules (class or group that contains common features) to allow solving the problem at appropriate level of granularity by ignoring unimportant and irrelevant details.

A primitive notion of granular computing is a granule representing part of a whole. A granule may be an element of another granule and is considered to be a part forming the other granule. It may also consist of a family of granules and is

considered to be a whole. Granular computing paradigm explores the composition of parts, their interrelationships, and connections to the whole. Though real-world problems may consist of a web of interacting and interrelated parts, granular computing exploits structures in terms of granules, levels, and hierarchies based on multilevel and multi-view representations, as hierarchical structures make a complex problem more easily understandable, leading to efficient approximate solutions [13],[14],[15],[16].

This thesis applies basic ideas, principles, and strategies of granular computing into the software complexity evaluation, in order to present the measure that more thoughtfully considers the complexity factors by structuring or grouping them, so that the complexity calculation process can be made analogous to when human comprehends the software.

2.3.2 Combinatorial Counting Rules

In Combinatorics [25],[26],[27],[28], there are two basic counting rules: the rule of sum, and the rule of product. Counting rules are the foundation of problems involving counting.

The rule of product [25] states that “the number of ways to do a procedure that consists of two subtasks is the product of the number of ways to do the first task and the number of ways to do the second task after the first has been completed”.

This rule indicates that “multiplication” is used when the two sets we are counting, are dependent on each other. Applying this rule to counting the cognitive complexity implies that the total cognitive complexity of two blocks of software code should be calculated from the product of the amount of the cognitive complexity of each block if and only if the understanding of a particular block of code requires the preceding comprehension of the other block.

The rule of sum [25] states that “the number of ways to do a task in one of the two ways is the sum of the number of ways to do these tasks if they can not be done simultaneously”.

This rule reflects a fact about set theory. It states that “addition” is used when the two sets we are counting, are disjoint. Applying this rule to counting the cognitive complexity implies that the total cognitive complexity of two blocks of software code should be computed from the sum of the amount of the cognitive complexity of each block if and only if to comprehend each block does not require the understanding of the other block at all.

As the computation of total cognitive weights of basic control structures resembles the counting rules [17], this thesis uses counting rules to derive the combinatorial meanings of the calculation, in order to find the weaknesses and the direction for improvement.

2.3.3 Weyuker’s Properties

Weyuker’s properties [18] consist of nine properties of syntactic software complexity measures widely used as criteria for evaluating software measures. However, many classical complexity measures, such as LOC, McCabe Cyclomatic number, Halstead’s effort, fail to satisfy some of these properties as shown in Table 3 [18],[29],[30].

As having been proved in [29],[30], and shown in TABLE 3, the Cognitive Functional Size (CFS) mostly satisfies eight of totally 9 properties, that are:

Let P and Q be a program body.

Property 1. ($\exists$ P) ($\exists$ Q) (|P| $\neq$ |Q|)

Property 2. Let c be a non negative number, then there are only finitely many programs of complexity c.

Property 3. There are distinct program P and Q such that |P| = |Q|

Property 4. ($\exists$ P) ($\exists$ Q) (P $\equiv$ Q & |P| $\neq$ |Q|)

Property 5. ($\forall$ P) ($\forall$ Q) (|P| $\leq$ |P;Q| & |Q| $\leq$ |P;Q|)

Property 7. There are some program bodies P and Q such that Q is formed by permuting the order of statements of P, and |P| $\neq$ |Q|

Property 8. If P is renaming of Q, then |P| = |Q|

Property 9. ($\exists$ P) ($\exists$ Q) ((|P| + |Q|) < |P;Q|)

However, CFS fails to satisfy Property 6, which states that:

Property 6a. ($\exists$ P)($\exists$ Q)($\exists$ R)((|P|=|Q|) & (|P;R| $\neq$|Q;R|))

Property 6b. ($\exists$ P)($\exists$ Q)($\exists$ R)((|P|=|Q|) & (|R;P| $\neq$|R;Q|))

Table 3. Evaluation of Complexity Measures against Weyuker's Properties

Property	LOC	McCabe's Cyclomatic	Halstead's Effort	Dataflow Complexity	CFS
1	/	/	/	/	/
2	/	X	/	X	/
3	/	/	/	/	/
4	/	/	/	/	/
5	/	/	X	X	/
6	X	X	/	/	X
7	X	X	X	/	/
8	/	/	/	/	/
9	X	X	/	/	/

This thesis uses Weyuker's properties as the evaluation criteria for comparing the proposed measure with the selected measures. However, as we found

that the true intent behind Weyuker's Properties is to check whether complexity value of a program is suitable with complexity values of its parts, but the nature of the definitions leaves some rooms for improvement to make it complies with its true intent, we also propose a new inductive framework for evaluating cognitive complexity measures to patch the holes in Weyuker's properties.

2.3.4 Framework for Evaluating Metrics

Kaner and Bond's framework for evaluating metrics [19] is a series of questions for evaluating the meaningful measures based on how much they could capture the essence of what they are supposed to measure. The main purpose of the framework is to help answer the question "How do you know that you are measuring what you think you are measuring?".

Kaner and Bonds defined measurement as "the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them. [19]" The framework's key issues of practical measurement can be summarized as follows:

"1) What is the purpose of this measure? Examples of purposes include:

- facilitating private self-assessment and improvement.
- evaluating project status (to facilitate management of the project or related projects)
- evaluating staff performance

2) What is the scope of this measure? A few examples of scope:

- a single method from one person
- one project done by one workgroup
- a year's work from that workgroup

3) What attribute are we trying to measure?

4) What is the natural scale of the attribute we are trying to measure?

5) What is the natural variability of the attribute? What are the inherent sources and degrees of variation of the attribute we are trying to measure?

6) What is the metric (the function that assigns a value to the attribute)? What measuring instrument do we use to perform the measurement? For the attribute length, we can use a ruler (the instrument) and read the number from it. Here are a few other examples of instruments:

• Counting (by a human or by a machine). For example, count bugs, reported hours, branches, and lines of code.

• Matching (by a human, an algorithm or some other device). For example, a person might estimate the difficulty or complexity of a product by matching it to one of several products already completed. ("In my judgment, this one is just like that one.")

• Comparing (by a human, an algorithm or some other device). For example, a person might say that one specification item is more clearly written than another.

• Timing (by computer, by stopwatch, or by some external automated device, or by calculating a difference between two timestamps). For example, measure the time until a specified event (time to first failure), time between events, or time required to complete a task.

7) What is the natural scale for this metric?

8) What is the natural variability of readings from this instrument?

9) What is the relationship of the attribute to the metric value? What model relates the value of the attribute to the value of the metric?

10) What are the natural and foreseeable side effects of using this instrument?" [19]

This thesis uses Kaner and Bond's framework to evaluate the proposed measure to confirm that it is one practically valid measure.

Chapter 3

The Structured Cognitive Complexity Measure with Granular Computing Strategies

3.1 Decomposition of Software into Granular Hierarchical Structure

According to Laue [31], “When the code grows beyond a subroutine or module, its complexity to the programmer is better assessed by measuring constructs other then the number of lines of code. The reason for this is that a program is understood by a programmer in small pieces, not as a whole." Representing software with proper and efficient representations is essential to unproblematic source code analysis. Control flow graph (CFG), or flowchart, is a technique commonly used to represent program algorithms or procedures. The representation can be used for various purposes, e.g. McCabe [1] used control flow graphs for computing the complexity of software. However, flowcharts only capture the ‘control flow’ aspect, whereas complexity depends on many other factors, such as size of software, numbers of objects or operators. Besides, the ‘web’ structure of the graph could cause difficulty in analysis. Therefore, representing source code with ‘tree’ structure rather than ‘web’ may be an interesting solution for easier understanding and analysis of software complexity.

Granular computing, as explained in chapter 2.3.1, is a branch of study in cognitive informatics which has been recently suggested as a universal problem-solving paradigm analogous to human cognitive process. The discipline addresses the approach to representing the universe of problems with multiple partial-order trees called “granular structures” instead of web in order to make complex problems more easily understandable, leading to efficient and approximate solutions. We therefore apply basic ideas, principles, and strategies of granular computing, described in chapter 2.3.1, into the software complexity evaluation, in order to present the measure that more thoughtfully considers the complexity factors by structuring or grouping them

so that the complexity calculation process can be made analogous to when human understands the software.

The philosophy of granular computing implies two dependent tasks of structured problem solving: constructing a hierarchical view and working with associated hierarchy [15]. To apply granular computing strategies to cognitive complexity measurement, first we decompose software into a hierarchy of granules.

When we comprehend the software, a basic control structure (BCS) can be seen as a comprehension unit of which we need to understand functionalities and inputs/outputs before understanding interaction between BCS units and the whole program. Therefore, in the context of cognitive complexity measurement, we view a granule as a basic control structure (BCS), which may contain nested inner BCS's and information content.

The decomposition methodology of the program can be explained as followed:

1) At the top level of the hierarchy, the whole program or function is partitioned into granules of BCS's in linear structure.

2) Each granule whose corresponding BCS contains nested BCS's inside, is further partitioned generating next level of hierarchy. The partitioning stops when corresponding BCS to the granule is a linear BCS.

In brief, each level of the hierarchy consists of BCS's in linear structure to one another, and because a BCS that contains no nested BCS's inside can be said to contain a single linear BCS, leaf nodes of the decomposed hierarchy are the linear BCS's. An example construction of the hierarchy from a program from [32] can be demonstrated as in Figure 3.

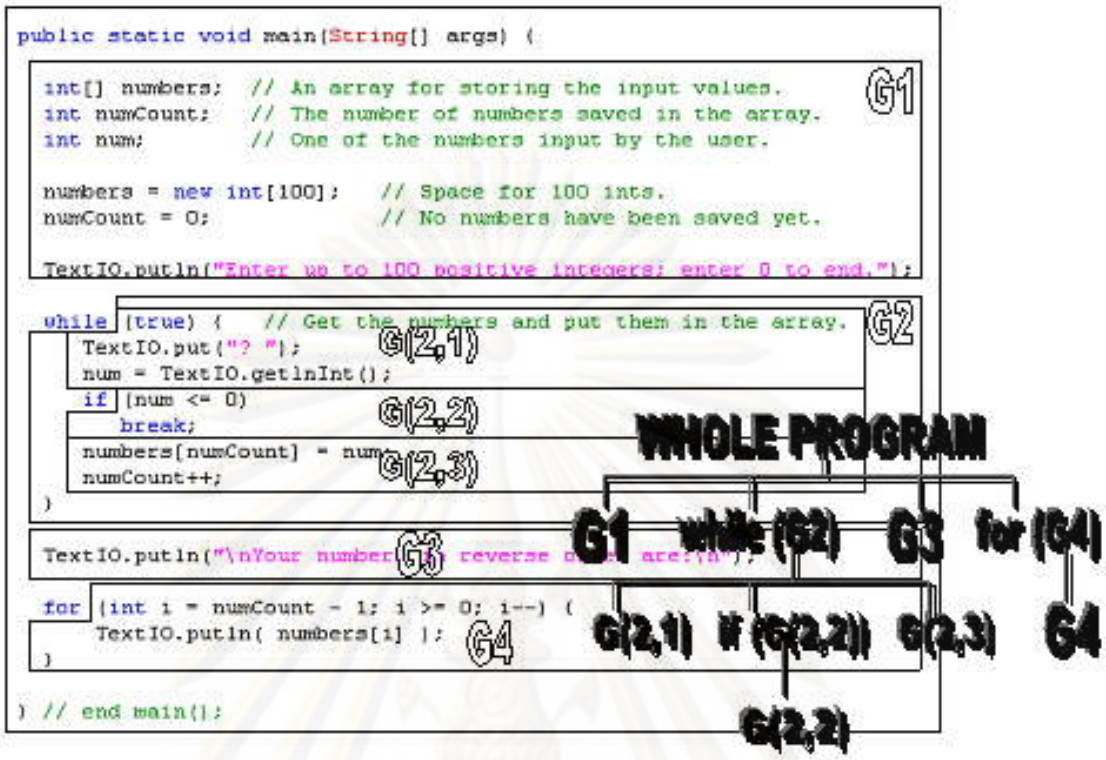


Figure 3. Demonstration of granular hierarchical structure construction

3.2 Derivative of the Total Cognitive Weight and Combinatorial Counting Rules

The computation of the total cognitive weight of basic control structures (W_c) used in previously proposed cognitive complexity measures resembles the counting rules in Combinatorics, which are the foundation of any matters involved counting, as described in chapter 2.3.2.

As the cognitive weight of BCS is defined as “the extent of relative effort spent on comprehending the function and semantics of BCS” [8], the use of ‘multiplication’ with the weights of nested BCS’s implies that to understand the function of the whole nested BCS’s architecture, it is required to fully understand the whole contents in the inner BCS’s first, then understand the outer ones surrounding it, layer by layer from inside out. This seems reasonable compared to the use of ‘addition’ with linear BCS’s, which implies that the cognition of these BCS’s are completely disjoint. In

other words, the functionalities of BCS's in linear structure can be understood separately and their complexities are completely disjoint. Because of this flaw, applying Algebra's Distributive Property, i.e. "a (b + c) = ab + ac" to the formulation of W_c defined in chapter 2.2.1 can result in an alternative method to calculate W_c i.e. W_c can be computed by finding the basic control structures within which do not contain any basic control structures, multiplying their weights by the weights of all their outer BCS's, then summing them altogether. This flaw causes the equivalence of the two structures derived from combinatorial meanings of two calculation methods of W_c as shown in Figure 4. In Figure 4, the right structure is derived from combinatorial meanings of the alternative method to calculate W_c of the left structure.

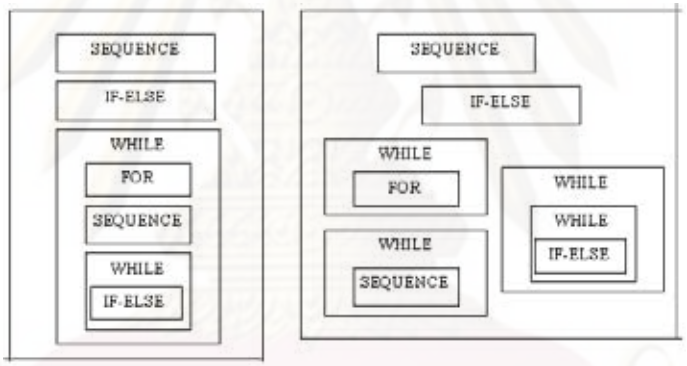


Figure 4. Algebraically Equivalence of W_cs of the two Program Structures.

The algebraically equivalence in Figure 4 only happened by chance because the definition of W_c is based on the assumption that the complexity inside one block cannot be transferred to another block in linear structure, as analyzed from combinatorial reasoning in [17]. This reveals the major weakness of the total cognitive weights of software that it does not consider the possible data flow from one BCS block to another. Moreover, the existing formula to compute W_c assigns the same complexity weight to each BCS of the same kind. For example, the "while" blocks are always

considered as equally difficult to understand no matter how many different numbers of variables contained within as long as they do not contain any nested BCS's inside.

It is obvious that the BCS blocks are not independent from each other, even though they are posed in linear structure. This is because the variables can carry the complexity from one block to another. Therefore, the complexity when trying to understand the linear BCS chunks cannot be evaluated separately as implied by the calculation of the total cognitive weights.

3.3 The Structured Cognitive Information Measure of Software (SCIM)

To work with the hierarchy obtained from the definition in chapter 3.1 to calculate the complexity value, we follow the principle of focused effort in Granular Computing [15], which states that "at a given stage, effort is to be concentrated on a particular granule, relatively independent of other granules". By focusing on a leaf-node granule, we evaluate its complexity from the amount of information it contains.

In order to measure the amount of information contained in the linear BCS, we agree with Kushwaha and A.K. Misra [9] that "the amount of information contained in software is a function of identifiers that hold the information and operators that perform the operations on the information." However, S. Misra's argument [12] was also true that "operators are run time attributes and cannot be regarded as information contained in the software." Therefore in our approach, identifiers in the form of variables are viewed as the major objects that contain the information in the software. However, operators cannot be completely disregarded like in [12], as they perform operations on the information, therefore increase the complexity of the information.

Furthermore, to solve the problem analyzed in chapter 3.2, which prevent other cognitive complexity measures from satisfying Weyuker's property 6, we observe that a variable accumulates the complexity from its preceding occurrences where it was assigned the value, as its value depends on those preceding

appearances. Since we have to focus on particular granule when evaluating its complexity, we include the complexity from the variable's occurrences in preceding granules into the variable itself, so that we can focus on its current occurrence in the granule that are being evaluating. Our strategies can be defined as followed:

Definition 1. At the beginning of the program or function, the Informatics Complexity Number (ICN) of every variable is zero. When a variable is assigned the value in the program or function, its ICN increases by 1, and if that assignment statement contains operators, ICN of the variable that is assigned the value also further increases by the number of operators in that statement.

The counting of ICN can be illustrated as presented in Figure 5. The figure shows that occurrences of the variable in the left-hand-side of the assignment statements increase its ICN, while occurrences in the right-hand-side of the statements do not.

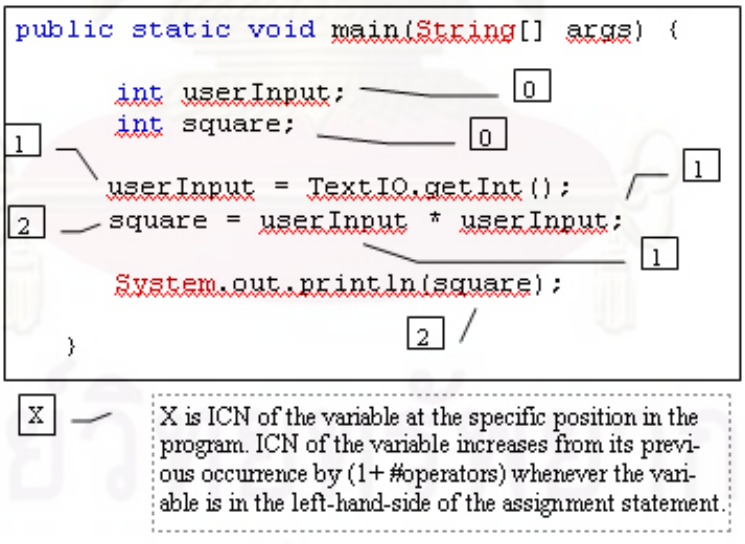


Figure 5. Example of ICN Computation

The purpose of this cumulative variable complexity counting scheme is to enable focusing on particular granule without losing the sense of interrelationships

between granules. It is arguable that the definition is based on the assumption that programs are read linearly, however, most of the times, programs are actually read linearly, therefore the definition is sensible as an approximate solution conforming to the principle that "in forming a granule, subtle differences between its elements and between their individual connections to others can be ignored or approximated. [15]"

Definition 2. For variable 'V' appearing in leaf node granule 'L', $ICN_{max}(V,L)$ is the highest ICN of V's occurrences in L.

Definition 3. Information contained in leaf node granule 'L' (I(L)) is defined as the sum of $ICN_{max}(V,L)$ of every variable V exists in L.

After obtaining the complexity of leaf granules, we evaluate the complexity of other granules up the hierarchy by the product of weight of the corresponding BCS to the granule and the sum of complexity values of their children granules. The use of 'sum' and 'product' resembles the counting rules in Combinatorics, which suggest that 'sum' is used when two sets are completely independent and 'product' is used when two things being counted depend on each other. We can say that complexities of granules in linear structures are independent because we already approximated the complexity transferred between granules by the cumulative variable complexity counting scheme. Therefore we use 'sum' with complexity values of granules in linear structures. On the other hand, to comprehend the functionality of the BCS, the processing characteristic of the BCS needs be understood and related to comprehension of the information inside. Therefore we use 'product' with Wc and the summed complexity of children granules. As a result, SCIM formula can be summarized as in definition 4.

Definition 4. SCIM is defined as the total sum of the products of corresponding cognitive weights and information contained in leaf node granule (I(L)). Since software may consist of q linear blocks composed in individual BCS's, and each block may consist of 'm' layers of nesting BCS's, and each layer with 'n' linear BCS's, then

$$SCIM = \sum_{j=1}^{q} \prod_{k=1}^{m} [\, W_c(j,k) \sum_{i=1}^{n} I(j,k,i) \,]$$

where weights W_c (j,k) of BCS's are cognitive weights of BCS's presented in table 2, and I(j,k,i) are information contained in a leaf BCS granule as defined in Definition 3.

From the definition, we can say that SCIM evaluates the complexity by taking into account the dependencies of variables and their position in the BCS's structure as suggested by Figures 1 and 2. Number of inputs/outputs can now be disregarded as I/Os variables have already been included as the information contained in the program.

3.4 The Unit of SCIM

In SCIM, the simplest software component with only one variable assignment, no operators, and a linear sequential BCS structure, is defined as the Structured Cognitive Information unit (SCIU), computing SCIM can be formulated as:

SCIM = 1 * 1 = 1 [SCIU]

The value in SSCU of a software system indicates its cognitive complexity relative to that of the defined simplest software component, or

SCIU = cognitive complexity of the system / cognitive complexity of the defined simplest software component.

3.5 Illustration of SCIM Computation

In this section, we illustrate the computation of SCIM value of the below example java program from [32], which are also used in our comparative case studies in chapter 4.4. We also demonstrate the calculation of CFS, CPCM, and MCCM.

```
public class ReverseInputNumbers {

 public static void main(String[] args) {

  int[] numbers;  //An array for storing the input values.
  int numCount;   //The number of numbers saved in the array.
  int num;        //One of the numbers input by the user.

      numbers = new int[100];    //Space for 100 ints.
      numCount = 0;              //No numbers have been saved yet.

      TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

  while(true){ //Get the numbers and put them in the array.
         TextIO.put("? ");
         num = TextIO.getlnInt();
   if(num <= 0)
    break;
         numbers[numCount] = num;
         numCount++;
  }

      TextIO.putln("\nYour numbers in reverse order are:\n");

  for(int i = numCount - 1; i >= 0; i--) {
          TextIO.putln( numbers[i] );
  }

 }//end main();

}//end class ReverseInputNumbers
```

3.5.1 SCIM Computation

The SCIM complexity value of the above example java program from [32] can be calculated by the sum of complexity of all functions in the program. As the program has only one function (the main function), first we have to decompose the function into granular hierarchy. From the definition in chapter 3.1, we obtain the hierarchy as in Figure 6.

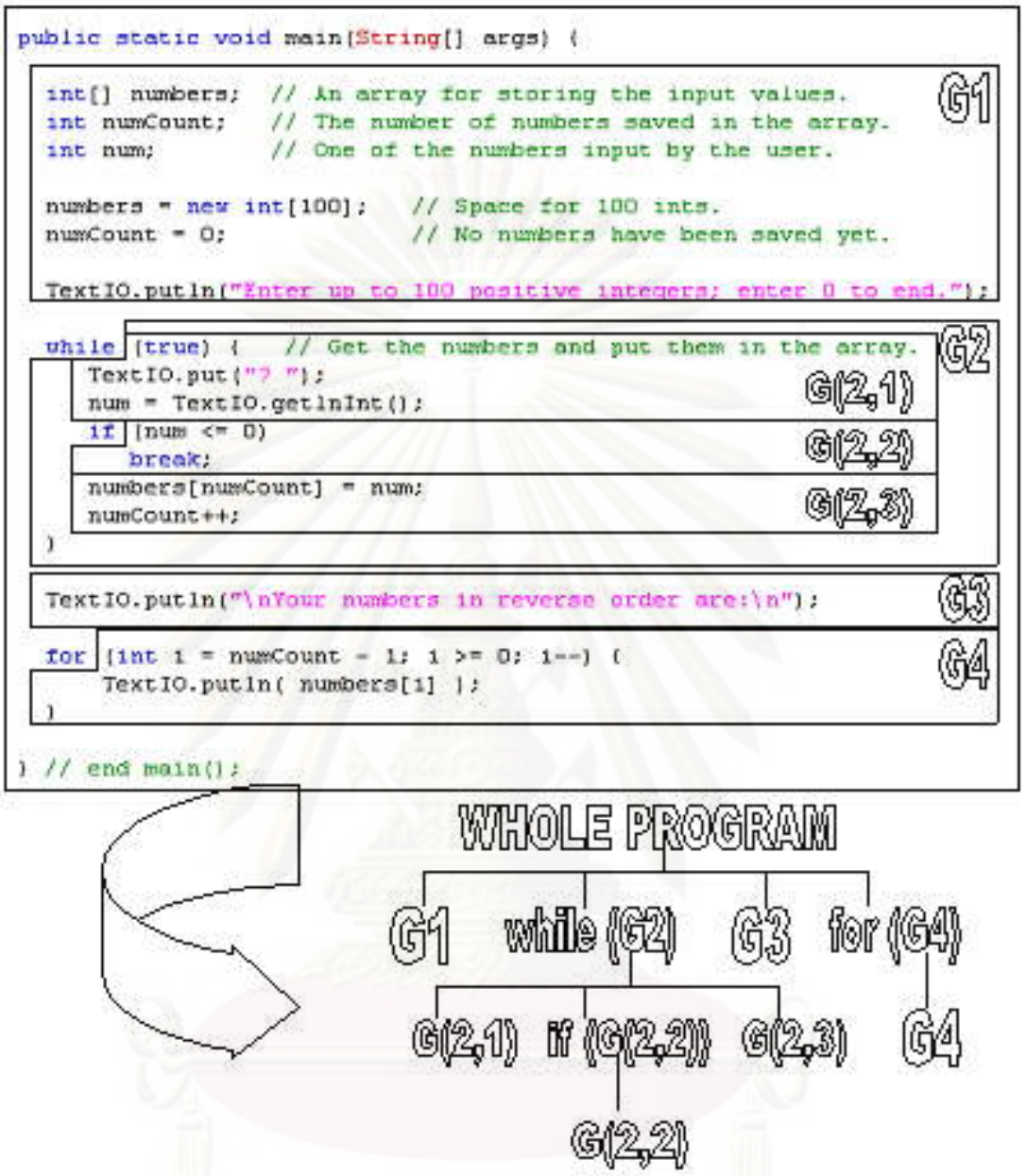


Figure 6. Granular Hierarchy of the Example Program

Referring to definition 1 in chapter 3.3, numbers shown in Figure 7 are ICNs of the variables at each position. Numbers in red are the increased ICNs at the assignment statements. Circled variables are the variables whose ICNs are used to calculate I(L) for the leaf node granules they reside.

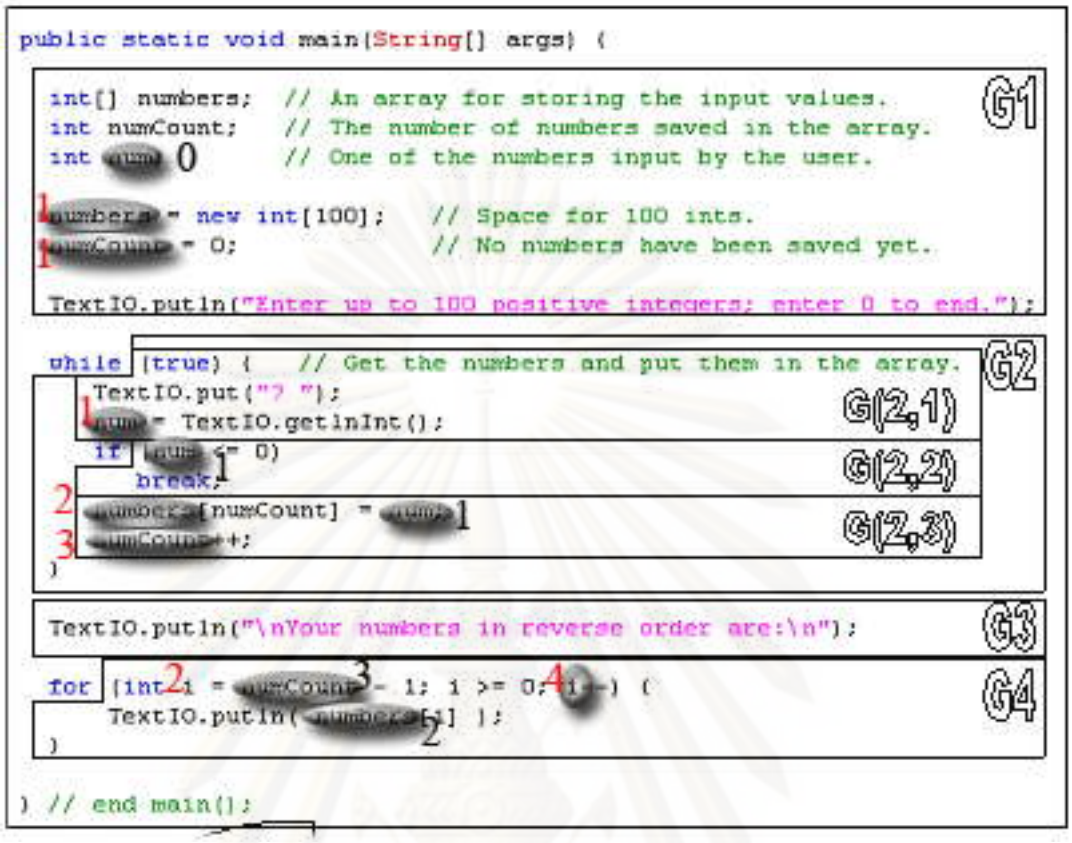


Figure 7. ICNs used to calculate complexity of the example program

Leaf node granule G1 contains 3 variables – num, numbers, and numCount. Their highest ICNs in granule G1 are 0,1, and 1 respectively. Therefore, I(L) of G1 is 0+1+1 = 2.

Leaf node granule G(2,1) contains 1 variables – num. Its highest ICNs in granule G(2,1) is 1. Therefore, I(L) of G(2,1) is 1.

Leaf node granule G(2,2) contains 1 variables – num. Its highest ICNs in granule G(2,2) is 1. Therefore, I(L) of G(2,2) is 1. The granule is surrounded by BCS 'if', whose cognitive weight is 2, therefore the complexity of "if(G(2,2))" is 2x1 =2.

Leaf node granule G(2,3) contains 3 variables – numbers, numCount, and num. Their highest ICNs in granule G(2,3) are 2,3, and 1 respectively. Therefore, I(L) of G(2,3) is 2+3+1 = 6

Leaf node granule G3 does not contain any variables, therefore its I(L) is 0.

Leaf node granule G4 contains 3 variables – i, numCount, and numbers. Their highest ICNs in granule G4 are 4, 3, and 2 respectively. Therefore, I(L) of G4 is 4+3+2 = 9. The granule is surrounded by BCS 'for', whose cognitive weight is 3, therefore the complexity of "for(G(4))" is 3x9 =27

Granule G2 consists of granule G(2,1), if(G2,2), G(2,3) in linear structure. Therefore I(L) of G2 = 1+2+6 = 9. The granule is surrounded by BCS 'while', whose cognitive weight is 3, therefore the complexity of "while(G(2))" is 3x9 =27

The whole function consists of granule G1, while(G2), G3, for(G4) in linear structure. Therefore SCIM = 2+27+0+27 = 56.

3.5.2 CFS, CPCM, and MCCM Computation

From the same example function, we can draw the structure of the BCS's as demonstrated in Figure 8. Thus, the total cognitive weight of the program, as defined in chapter 2.2.1, is 1 + [3 * (1+2+1)] + 1 + 3 = 17

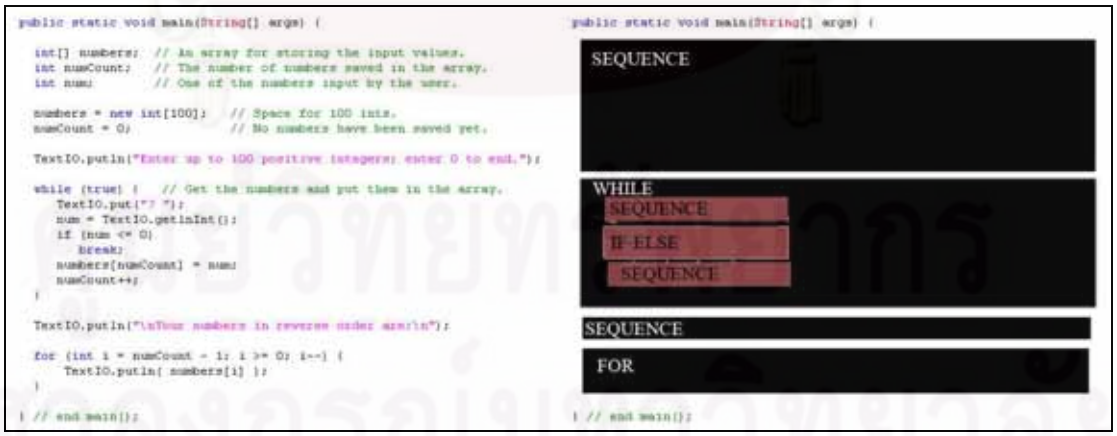


Figure 8. BCS's structure of the example program

The program has 1 input (num) and 1 output (numbers). Thus,

CFS = (1+1) * 17 =34.

The number of occurrences of I/O variables (num and number) in the program is 8, as illustrated in figure 9.

```
public static void main(String[] args) {

   int[] numbers;  // An array for storing the input values.
   int numCount;   // The number of numbers saved in the array.
   int num;        // One of the numbers input by the user.

   numbers = new int[100];   // Space for 100 ints.
   numCount = 0;             // No numbers have been saved yet.

   TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

   while (true) {   // Get the numbers and put them in the array.
      TextIO.put("? ");
      num = TextIO.getlnInt();
      if (num <= 0)
         break;
      numbers[numCount] = num;
      numCount++;
   }

   TextIO.putln("\nYour numbers in reverse order are:\n");

   for (int i = numCount - 1; i >= 0; i--) {
       TextIO.putln( numbers[i] );
   }

} // end main();
```

Figure 9. Inputs/outputs occurrences in the example program

Therefore CPCM = 8 + 17 =25.

The number of identifiers and operators occurrences in the program is 22, as illustrated in figure 10.

Therefore MCCM = 22 * 17 = 374.

```
public static void main(String[] args) {

   int[] numbers;  // An array for storing the input values.
   int numCount;   // The number of numbers saved in the array.
   int num;        // One of the numbers input by the user.

   numbers = new int[100];   // Space for 100 ints.
   numCount = 0;             // No numbers have been saved yet.

   TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

   while (true) {   // Get the numbers and put them in the array.
      TextIO.put("? ");
      num = TextIO.getlnInt();
      if (num <= 0)
         break;
      numbers[numCount] = num;
      numCount++;
   }

   TextIO.putln("\nYour numbers in reverse order are:\n");

   for (int i = numCount - 1; i >= 0; i--) {
       TextIO.putln( numbers[i] );
   }

} // end main();
```

Figure 10. Identifiers/operators in the example program

Chapter 4

Validations of the Proposed Measure

4.1 Theoretical Validation through Weyuker's Properties

The proposed SCIM can be proved to satisfy all nine Weyuker's properties [18] stated in chapter 2.3.3. The properties are often used to evaluate and compare complexity measures [11],[12],[29],[30].

Let P and Q be program body.

Property 1. ($\exists$ P) ($\exists$ Q) (|P| $\neq$ |Q|)

This property states that the measures should not rank all the programs as equally complex. Therefore, SCIM obviously satisfies this property.

Property 2. Let c be a nonnegative number, then there are only finitely many programs of complexity c.

Since all programming languages can have only finite number of BCS's, variable assignments, and operators, it is assumed that some largest numbers can be used as an upper bound on the numbers of BCS's, variable assignments and operators. Therefore, for these numbers, there are finite many programs having that much number of BCS's, variable assignment, and operators. Consequently, for any given value of SCIU, there exists finitely large number of programs, and SCIM satisfies this property.

Property 3. There are distinct program P and Q such that |P| = |Q|

SCIM clearly satisfies this property as at least for any program containing operator '+', replacing '+' with '-' will result in a different program with the same SCIM complexity.

Property 4. ($\exists$ P) ($\exists$ Q) (P $\equiv$ Q & |P| $\neq$ |Q|)

This property states that there exist two programs equivalent to each other (i.e. for all inputs given to the program, they halt on the same values of outputs.) with different complexity. Clearly, the program computing 1+2+3+…+n can be implemented with while loop, or simply sequence structure with formula n(n+1)/2. The values of SCIM from these two implementations are different. Hence, SCIM satisfies this property.

Property 5. ($\forall$ P) ($\forall$ Q) (|P| $\leq$ |P;Q| & |Q| $\leq$ |P;Q|)

SCIM obviously satisfies the property because adding any program body whether to the end or before the beginning of a program body can only increase or hold the SCIM complexity.

Property 6a. ($\exists$ P)($\exists$ Q)($\exists$ R)((|P|=|Q|) & (|P;R| $\neq$ |Q;R|))

Given program P and Q with same value in SCIU, and program R contains some variables that are assigned values in P but no variables that are assigned values in Q, |P;R| is clearly more than |Q;R| because ICNs of those variables in R of P;R are higher than those of the same variables in R of Q;R. Therefore, SCIM satisfies this property.

Property 6b. ($\exists$ P)($\exists$ Q)($\exists$ R)((|P|=|Q|) & (|R;P| $\neq$ |R;Q|))

In the same way as in property 6a, SCIM satisfies this property.

The satisfaction of property 6 indicates one strength of SCIM over other cognitive complexity measures that when different programs with the same complexity value are extended with the same program part, other measures view the extended programs as having the same complexity no matter what. This is because they do not consider possible complexity transferred between BCS in linear structures, or view linear BCS's as completely separately comprehensible, while SCIM estimates the complexity transferred between blocks of BCS by the cumulative variable complexity counting scheme and does not overlook interrelationships among granules.

Property 7. There are some program bodies P and Q such that Q is formed by permuting the order of statements of P, and $|P| \neq |Q|$

SCIM satisfies this property because the permutation of statements can result in different ICNs, hence making the SCIM value different.

Property 8. If P is renaming of Q, then |P| = |Q|

SCIM clearly satisfies this property as it does not take into account the names.

Property 9. ($\exists$ P) ($\exists$ Q) ((|P| + |Q|) < |P;Q|)

SCIM satisfies this property because if some variables assigned values in P occur in Q, the complexity of Q in P;Q will increase from Q alone because the ICNs of those variable will increase, hence making |P;Q| higher than |P| + |Q|.

Table 4. Comparison of Conformance of Complexity Measures to Weyuker's Properties

Property	LOC	McCabe's Cyclomatic	Halstead's Effort	Dataflow Complexity	CFS	MCCM	CPCM	SCIM
1	/	/	/	/	/	/	/	/
2	/	X	/	X	/	/	/	/
3	/	/	/	/	/	/	/	/
4	/	/	/	/	/	/	/	/
5	/	/	X	X	/	/	/	/
6	X	X	/	/	X	X	X	/
7	X	X	X	/	/	X	X	/
8	/	/	/	/	/	/	/	/
9	X	X	/	/	/	/	/	/

Satisfying all nine Weyuker's properties is one of the main obvious leaps forward of SCIM from previous cognitive complexity measures and classical complexity measures, as Weyuker's properties are considered as "necessary but not sufficient

characteristics of ideal complexity measures [22],[30],[33].” Classical measures, e.g. LOC, McCabe’s, Halstead’s, and Dataflow Complexity, each failed to satisfy some different properties. Existing cognitive complexity measures made some progress but still failed to satisfy property 6. SCIM satisfies all the properties, as compared in table 4 [11],[12],[18],[20],[30].

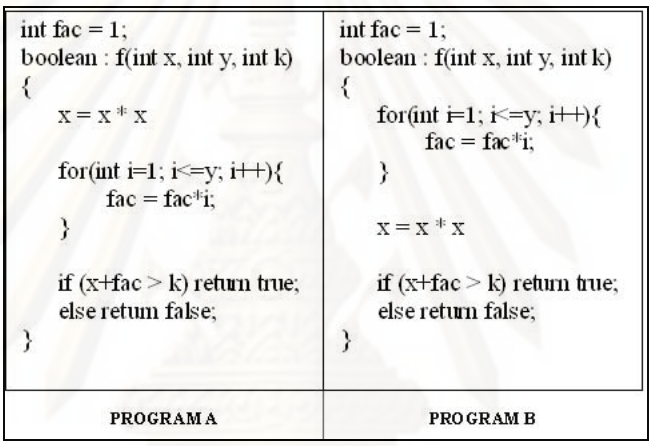
4.2 The New Inductive Framework for Evaluating Software Cognitive Complexity Measures

Over the past three decades, numbers of software complexity measures have been proposed to better predict quality dimensions such as testability, reliability, and maintainability of software systems. However, the evaluation basis for validating the measures is still not well-established. Weyuker’s properties were proposed and generally-used as basic necessary criteria for comprehensive software measure. However, the properties are still far from theoretical perfection [33], leading to many attempts to modify [33],[34],[35].

The intent behind Weyuker’s Properties is to check whether complexity value of a program is suitable with complexity values of its parts. However, the definitions of the properties leave some room for some measures to slip through. For example, the meaning of Property 6, $(\exists P)(\exists Q)(\exists R)((|P|=|Q|) \ \& \ (|P;R| \neq |Q;R|))$, is that when two different programs with the same complexity are concatenated with the same program, the two programs resulting from the concatenation may not have the same complexity. Logical explanation to this property is that if part P or Q have some effect on the execution of part R, the complexity in understanding of part R when concatenated to part P or Q, should increase from the complexity of part R alone, causing the complexity of P;R and Q;R to be different. On the other hand, if the calculation of part R has nothing to do with part P or Q, or in other words, part R is completely comprehensible separately from parts P and Q ,the complexity of P;R and Q;R should be the same. This is because P and Q are as complex and R in both P;R and Q;R are as complex. However, from the way the property is defined using logical predicate, CICM happens to satisfy Property 6 because its weighing of information

content is so random that there exist programs P, Q, R that |P|=|Q| but |P;R| ≠ |Q;R|. Even though sometimes, if R is completely independent of P and Q, |P;R| should actually be the same as |Q;R|.

Another example that exposes the improper definition of Weyuker's properties is the two programs in Figure 11. The two programs answer whether or not "x^2 + (y!) > k". The permuting order of x^2 and (y!) should not affect the cognitive complexity of program as the calculation of x^2 and (y!) can be perceived as completely independent of each other.



```
int fac = 1;
boolean : f(int x, int y, int k)
{
    x = x * x

    for(int i=1; i<=y; i++){
        fac = fac*i;
    }

    if (x+fac > k) return true;
    else return false;
}
```

PROGRAM A

```
int fac = 1;
boolean : f(int x, int y, int k)
{
    for(int i=1; i<=y; i++){
        fac = fac*i;
    }

    x = x * x

    if (x+fac > k) return true;
    else return false;
}
```

PROGRAM B

Figure 11. Two programs with same function

However, CICM, weighing the complexity of identifiers/operators in the n^{th} line by 1 / (LOC-n), does not give the same complexity for both programs, resulting in the satisfaction of Property 7, which says "There are some program bodies P and Q such that Q is formed by permuting the order of statements of P, and |P| ≠ |Q|."

$CICM_A = (4/6 + 6/5 + 4/4 + 5/2)*\{4+1+1\} = 32.2$

$CICM_B = (6/6 + 4/5 + 4/4 + 5/2)*\{4+1+1\} = 31.8$

For these reasons, we can say that some definitions of Weyuker's Properties are sometimes too loose to determine if the measures satisfy its true intent. The measure that truly satisfies the intent of Weyuker's properties should be able to

answer, “What would happen to |P;R| when P and R are in some condition to each other.” Therefore, we propose an inductive framework for evaluating software cognitive complexity measures. The purpose of the framework is to assess how well the measure reflects the cognitive complexity based on granular computing principles that “programs is understood by programmers by understanding their parts and relationships to the whole [20],[31].” In our framework, the measures have to answer following questions:

1) How is the unit of the measure defined? This is the basis step of the framework to make sure that the simplest unit of the measure is related to the measure’s purpose.

2) Inductive Step: given program parts P and R, what is the relationships between |P|, |R|, and |P;R| in case:

- 2.1) P and R are completely independent (The computation of P and R are not up to each other.)?
- 2.2) P and R are dependent (The computation of R can give different results if the results from the computation of P is different)?
- 2.3) P contains similar patterns with R, and P and R are completely independent?
- 2.4) P contains similar patterns with R, and P and R are dependent?

The measures shall be assessed through the framework by giving the descriptive answers to the questions. This gives flexibility to the assessment of how well the measure satisfies its purposes, definitions, meanings, and how well it is related to cognitive complexity. The descriptive answers can range from very generic, e.g. using inequalities or ‘for some’ logic, to very specific, e.g. using equalities, up to the measure’s

definition and how fine or coarse the measure wants to go. The answers shall show the strengths and weaknesses, in order to guide where to improve the measure.

4.2.1 Evaluation and Comparison of Complexity Measures through the Inductive Framework

We have evaluated LOC, DF, CFS, CICM, MCCM, CPCM, and SCIM though the new inductive framework to compare and analyze how well the measures relate to cognitive complexity.

Table 5. Evaluation of Complexity Measures through the Inductive Framework.

Measure	Questions				
	1	2.1	2.2	2.3	2.4
LOC	A line of code	$\lvert P;R\rvert = \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert = \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert = \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert = \lvert P\rvert + \lvert R\rvert$
CC	A linearly-independent path through a software component	$\lvert P;R\rvert \geq \max(\lvert P\rvert, \lvert R\rvert)$ and $\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$. Most likely, $\lvert P;R\rvert \gg \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert \geq \max(\lvert P\rvert, \lvert R\rvert)$ and $\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$. Most likely, $\lvert P;R\rvert \gg \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert \geq \max(\lvert P\rvert, \lvert R\rvert)$ and $\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$. Most likely, $\lvert P;R\rvert \gg \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert \geq \max(\lvert P\rvert, \lvert R\rvert)$ and $\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$. Most likely, $\lvert P;R\rvert \gg \lvert P\rvert + \lvert R\rvert$
HE	The complexity of the simplest software component with one operator occurrence and one operand occurrence.	$\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$.	$\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$.	$\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$.	$\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$.
DF	A variable definition defined outside a block but is used inside the block.	$\lvert P;R\rvert = \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$.	$\lvert P;R\rvert = \lvert P\rvert + \lvert R\rvert$	$\lvert P;R\rvert$ can be either >, <, or = $\lvert P\rvert + \lvert R\rvert$.

CFS	The complexity of the simplest software component with only one input or output, and a linear sequential BCS structure	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|
CICM	Cannot be defined.	|P;R| can be either >, <, or = |P| + |R| . Most likely, |P;R| ≠ |P| + |R|	|P;R| can be either >, <, or = |P| + |R| . Most likely, |P;R| ≠ |P| + |R|	|P;R| can be either >, <, or = |P| + |R| . Most likely, |P;R| ≠ |P| + |R|	|P;R| can be either >, <, or = |P| + |R| . Most likely, |P;R| ≠ |P| + |R|
MCCM	The complexity of the simplest software component with only single number of operator and operand and a linear structured BCS.	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|
CPCM	The cognitive weight of the simplest software component, a linear structured.	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|
SCIM	The complexity of the simplest software component with only one variable assignment, no operators, and a linear sequential BCS structure	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R| and |P;R| can be either >, <, or = |P| + |R| .	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R|	max (|P|, |R|) ≤ |P;R| ≤ |P| + |R| and |P;R| can be either >, <, or = |P| + |R| .

From the results in Table 5, LOC, CFS, MCCM, and CPCM all evaluate |P;R| as having no greater complexity than |P| + |R| no matter what. This shows the weakness of the measures that they ignore the fact that understanding a program part, when concatenated to another program, can be more difficult than understanding that part alone if two parts are dependent.

For CC, $|P;R| \geq \max(|P|, |R|)$, and can be either >,<, or = |P| + |R|, but most likely, |P;R| >> |P| + |R| no matter P and R are dependent or not. This indicates that CC value grows too fast and does not handle the dependency between P and R well.

For HE and CICM, which satisfy Weyuker's Property 6, the results from this inductive framework show that they just happen to satisfy the property because the relationship between |P;R|, |P|, and |R| derived from these measures are so random that there easily exists the set of programs that make the measures satisfy the 'for some'' logic of Weyuker's, but the satisfaction is just by chance and not associated with relative cognition difficulty. Moreover, the unit of CICM cannot be defined, thus making it a bogus measure.

For DF and SCIM, question 2.1 shows that "$|P;R| \leq |P|+|R|$" when P and R are completely independent, which makes sense for the difficulty for human brain to understand the software, as we need to understand P and R separately and need not comprehend the additional effect of P on R.

For SCIM, question 2.2 shows that |P;R| can be higher than |P|+|R| when P has some effects on R. This makes sense as to understand P;R, we need to under stand P, R, and also the relationships between P and R. Moreover, for SCIM, |P;R| cannot be less than max (|P|,|R|), but the problem of DF is that |P;R| can go lower than max (|P|,|R|), which is not reasonable.

However, if we get into more details, questions 2.3 and 2.4 show the weakness of all the measures, including SCIM, that none of them handle the situation when two program parts have the same pattern or similar logic, it would be much easier for programmers to understand the second part as they have already understood and memorized the first one with similar pattern. To improve the measures to better reflect the difficulty for human to comprehend the software, the measures need to take into account this important fact.

4.3 Evaluation through Kaner and Bond's Framework

We apply Kaner and Bond's metric evaluation framework [19] to assess how meaningful and practical our measure can "capture the essence of what they are supposed to measure." The framework is based on the following points:

- The purpose of the measure:

The main purposes of SCIM are to facilitate assessment of product quality, self-assessment and improvement for developer, as well as estimation of effort needed for support and maintenance of the software.

- Scope of usage:

SCIM is categorized as a technical metric applicable after coding. Its scope of usage is for software development and maintenance groups.

- Attributes to measure:

SCIM measures the difficulty in comprehending the software from the structures of basic control structures, variables, and operators. Nested control structures and frequent occurrences of variables and operators make the program more difficult to understand, hence, harder for maintenance and less desirable as a product.

- Natural scale of the attributes:

The existence of natural scale of 'difficulty in comprehending the software' requires the development of its common and non-subjective view. We have no knowledge about natural scale of 'difficulty'.

- Natural variability of the attribute:

Since the attribute is subjective and involves human cognition, its variation depends on so many complicated factors. The challenge is to develop a sound approach to handle such attribute with no knowledge about its variability.

- Definition of the metric:

The metric has been defined formally in chapter 3.3

- Measuring instruments:

SCIM uses the instrument of counting by either human or automation. The items to be counted are variable assignments and operators occurrences, matching to cognitive weights of BCS's. For automation purpose, a token generator can be developed to facilitate counting process in the future.

- Natural scale of the metric:

SCIM is on ratio scale according the measurement theory [36].

- Natural variability of readings from the instrument:

Assuming there are no bugs in the automated algorithm, we can expect no variability on readings from our counting instrument when the measure is strictly defined, as readings from our counting instrument are not subjective.

- Relationship of the attribute to the metric value:

There is a direct relation between the difficulty in comprehending the software and SCIM. This is because the as increase in SCIM value means that software is more difficult to understand and maintain, as it implies that the software contains more information content and more nested complex control structures.

- Natural and foreseeable side effects of using the instruments:

Once SCIM calculation is automated, it will not require additional human effort. Automation will be the only cost of the measure.

4.4 Comparative Case Studies with Real-World Programs

We have taken different java programs from [32] for analysis of complexity of the program. We calculated SCIM for each program and compared to LOC, CFS, MCCM, and CPCM. The results are presented in Table 6-7 and Figure 12-14.

Table 6. Experimental Results

Program	Program Name	Reference Section in the Book	LOC	CPCM	CFS	SCIM	MCCM
1	Interest	2.2	15	11	2	6	15
2	PrintSquare	2.4	11	8	2	3	8
3	Interest2	2.4	16	10	3	6	14
4	CreateProfile	2.4	28	13	8	4	15
5	Interest3	3.1	26	19	16	30	88
6	ComputeAverage	3.3	27	19	18	43	150
7	CountDivisors	3.4	29	42	99	105	1023
8	ListLetters	3.4	28	48	96	204	928
9	GuessingGame	4.2	41	66	116	205	1013
10	HighLow	5.4	83	166	494	456	5691
11	ReverseInputNumbers	7.3	22	25	34	56	374
12	CheckersData	7.5	165	1854	3472	12225	310317
13	Board	7.5	147	358	376	1211	17783
14	TowersOfHanoi	9.2	29	39	99	65	425
15	SimpleParser3	9.5	149	842	2731	788	15738
16	WordCount	10.4	104	171	366	473	4118
17	SimpleInterpreter	10.4	174	486	604	567	8858

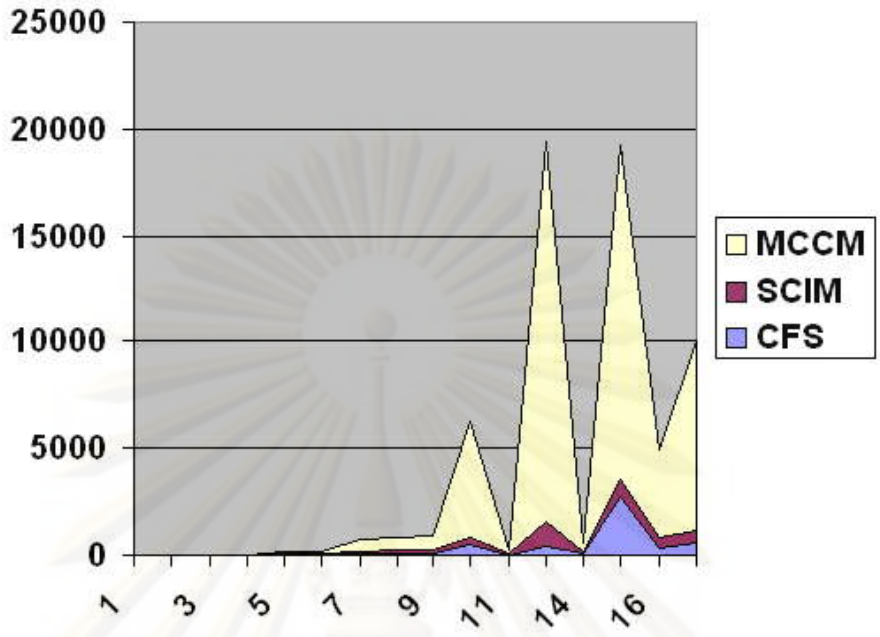


Figure 12. Comparison of SCIM-MCCM-CFS

Figure 12 shows that SCIM value usually falls between CFS and MCCM because CFS does not consider information in the forms of operators and operands, while MCCM over weighs them, thus making MCCM values far too high from others. On the other hand, SCIM suitably structures and weighs the operands, operators and basic control structures, therefore better reflects the complexity of information contained in the software. This works well on mathematical programs that have high numbers of internal variables and mathematical processing.

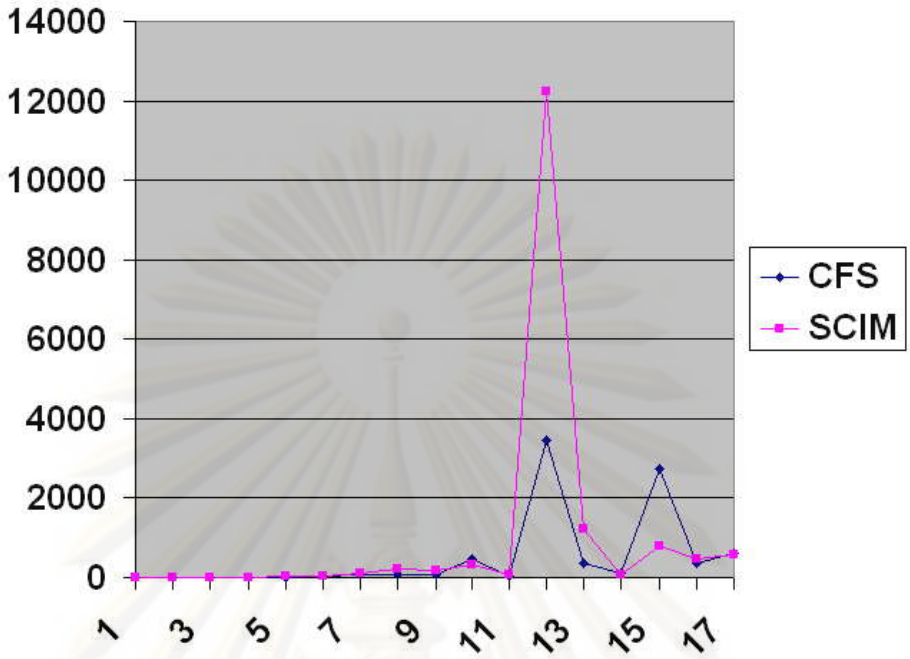


Figure 13. Comparison of SCIM and CFS

Analysis of the graph between CFS and SCIM can indicate some special nature of the programs. Normally the graphs of CFS and SCIM are of quite the same shape. In figure 13, there are five points where CFS values are higher than SCIM, and one point where SCIM value rises sharply. The five points where CFS values are higher than SCIM indicate that the programs are likely to contain the following characteristics:

- The programs have high number of inputs and outputs but most I/Os are not processed through the whole control structure. In other words, the programs whose CFS values are higher than SCIM can be seen as having many subroutines that receive inputs from users and produce their own outputs which are also outputs of the program, while these inputs/outputs are rarely used in other subroutines.

- Some input variables are the same as output variables, thus they are counted twice as inputs and outputs according to the definition of CFS, while SCIM sees them as the same variables.

- The programs contain many function calls with no parameter passing, or contain many nested BCS's that do not contain any information objects, e.g. containing following BCS inside the nested structure.

```
if(TextIO.peek() == '\n'){
   break;  //A blank input line ends the while loop and the program.

            }
```

CFS may take the weight of BCS 'if' to multiply with the weights of nested BCS's, causing the complexity value to be too high. While SCIM eliminates this BCS (calculates the complexity of this part as 0) since the granule contains no information object. The part thus becomes like the same granule as linear structures surrounding it. This fits well with the fact that experienced programmers do not see a block like this as more complex than just regular command executed in sequence with no additional complexity. The programs with GUI or File-Reading are likely to have this characteristic.

- The programs contain very few non-I/O variables, and the function contains many nested BCS's but the I/O variables are processed in the outer layer.

In contrast, the point where SCIM value is extra-ordinarily high from CFS indicates that the program may have following characteristics:

- The program contains static (fixed) functions that do not have any inputs or outputs, but have many internal variables and nested structures, e.g. the function to set up the checker board:

```
 void setUpGame() {
  for(int row = 0; row < 8; row++) {
   for(int col = 0; col < 8; col++) {
    if(row % 2==col % 2){
     if(row < 3)
                    board[row][col] = BLACK;
    else if(row > 4)
                    board[row][col] = RED;
     else
                    board[row][col] = EMPTY;
    }
    else{
                board[row][col] = EMPTY;
```

```
      }
    }
  }

                } //end setUpGame()
```

- The program contains many internal variables, and they are processed through many parts of the program or re-assigned to re-evaluate the values often. Or the programs contain many arrays and indexing. Mathematical functions tend to fit into this category.

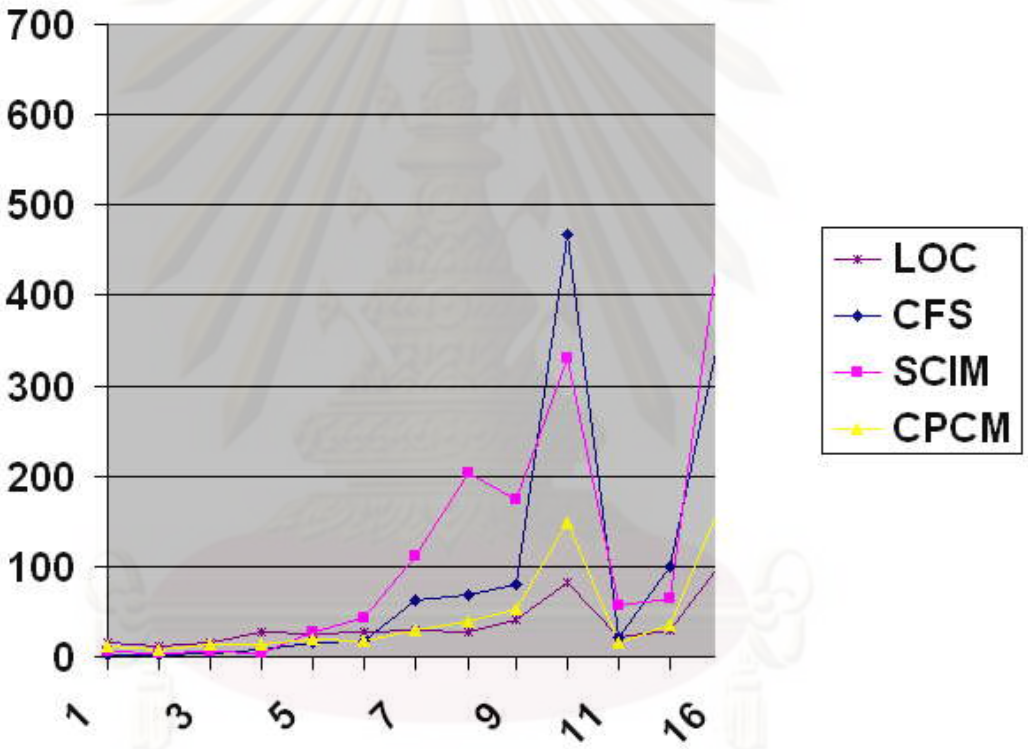


Figure 14. LOC-CFS-SCIM-CPCM comparison

Furthermore, the results in Figure 14 show that LOC and CPCM values are quite similar, while CFS and SCIM can indicate the coding efficiency (E), which can be defined as: *E = SCIM / LOC*

The higher coding efficiency indicates the higher-complexity information packed in the shorter program code, therefore the program is likely to contain more defects than the program with lower coding efficiency.

Table 7. Simplicity Rank Results

Program Name	SCIM Rank	CFS Rank	MCCM Rank	CPCM Rank	LOC Rank
PrintSquare	1	1.5	1	1	1
CreateProfile	2	4	3.5	4	7.5
Interest	3.5	1.5	3.5	3	2
Interest2	3.5	3	2	2	3
Interest3	5	5	5	5.5	5
ComputeAverage	6	6	6	5.5	6
ReverseInputNumbers	7	7	7	7	4
TowersOfHanoi	8	9.5	8	8	9.5
CountDivisors	9	9.5	11	9	9.5
ListLetters	10	8	9	10	7.5
GuessingGame	11	11	10	11	11
HighLow	12	14	13	12	12
WordCount	13	12	12	13	13
SimpleInterpreter	14	15	14	15	17
SimpleParser3	15	16	15	16	15
Board	16	13	16	14	14
CheckersData	17	17	17	17	16

[-0.5 , +0.5] [-2 , +2] [-3.5 , 3.5]

Additionally, Table 7 shows that SCIM can rank the complexity/simplicity of programs quite well. We ranked the simplicity of programs rating from 1 (relatively most simple) to 17 (highest complexity) by LOC, CFS, MCCM, CPCM, and CPCM. The results show that SCIM ranks are quite the same as CFS, MCCM, and CPCM, i.e. the variance is within [-2 , +2] ranks, while LOC gives ranks between [-3.5 , +3.5] ranks away from SCIM. Therefore SCIM can be used to rank the complexity in quite the same way as other cognitive complexity measures.

Chapter 5

Conclusion and Future Direction

5.1 Conclusion

In this thesis, the drawbacks of existing cognitive complexity measures were analyzed showing that the measures ignored the relationships among factors. We therefore applied granular computing strategies to present an approach to structuring the factors before the complexity calculation. We then proposed the cognitive complexity measure called "Structured Cognitive Information Measure (SCIM)", making cognitive complexity metrics more related to human cognitive process and supporting the suggested universal applicability of granular computing.

Our proposed measure solves three major problems of existing cognitive complexity measures we and other researchers analyzed. One is the lack of consideration of information content in CFS. Another is the ignorance of the detailed relationships between some factors e.g. the data objects and the basic control structures. The other is the irrational use of '+' and '*' in the formulation of the total cognitive weights of BCS's questioned by the Combinatorial Counting Rules. SCIM advances classical software measures like LOC by measuring the software more rigorously, as we take into account more complexity attributes. Halstead's and McCabe's only consider the complexity of software in terms of control flows of the program, and amount of data objects, respectively, while we take into account both aspects. Moreover, our Cumulative Variable Complexity Counting Scheme is superior to Data Flow Complexity (DF) because DF only measures the complexity transferred between blocks in the form of variables, while we cumulate the complexity of variables transferred between blocks and complexity of their own occurrences within the blocks as well. Function Point is not cost effective and needs subjective evaluation, while our measure is more objective and can be automated.

Work in this thesis tends to study in terms of conceptual or schema-based rather than empirical or practical studies. Proposed SCIM was proven by satisfying Weyuker's properties. The inductive framework for evaluating software

complexity measure was also proposed to patch some holes in Weyuker's properties and guide the direction for improving the cognitive complexity measures. Practical evaluation through comparative case studies also showed that SCIM better considers details of comparative complexity, therefore can help establish cognitive complexity metrics as a more mature and sound discipline, as well as contribute to product quality assessment and improvement for the software development and support groups.

5.2 Future Direction

The work in this thesis shall be further developed in the following directions:

- Cognitive weights of BCS's shall be further experimented and re-adjusted to make them truly reflect the cognitive complexity of the structure.

- SCIM and other cognitive complexity measures shall be fully automated, in order to implement the experimental evaluation in the software industry. The implementation for each programming language may require some degrees of interpretation.

- Cognitive complexity measures shall be made to consider human ability to recognize the repeated pattern in the software, which highly influences the difficulty for human to comprehend the software, yet no measures have taken into account this aspect, as suggested by the proposed inductive framework.

- The proposed inductive framework shall be extended with more conditions to the inductive questions, in order to make it more rigorously defines the characteristics of desired cognitive complexity measures.

As complex as human brain is, there are still many miles to go, many challenges to tackle, many mysteries to explore in the study of software cognitive complexity. We hope this thesis brings some kinds of light to the starting of the journey into the wonder of software and human cognition, and is able to guide some degrees of

directions towards the search of the ideal software cognitive complexity measure that can efficiently reflect human's effort to comprehend the software.

References

[1] Thomas J. McCabe. A Complexity Measure. Proceedings of the 2nd international conference on Software engineering June 1976 : pp. 407.

[2] Maurice H. Halstead. Elements of Software Science (Operating and programming systems series). New York : Elsevier Science Inc., 1977.

[3] Mark Burgin. From Craft to Engineering:Software Development and Schema Theory. Proceedings of the 1st WRI World Congress on Computer Science and Information Engineering (CSIE 2009) March 2009 : pp. 787.

[4] Yinxu Wang. On Cognitive Informatics, Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy vol. 4, no.2, 2003 : pp.151.

[5] Yinxu Wang. On the Cognitive Informatics Foundations of Software Engineering. Proceedings of the 3rd IEEE International Conference on Cognitive Informatics 2004 : pp. 22.

[6] Yinxu Wang. On the informatics laws of software. Proceedings of the 1st IEEE International Conference on Cognitive Informatics 2002 : pp 132.

[7] Yinxu Wang and Jingqiu Shao,. Measurement of the Cognitive Functional Complexity of Software. Proceedings of the 2nd IEEE International Conference on Cognitive Informatics August 2003 : pp. 67.

[8] Yingxu Wang. Cognitive Complexity of Software and its Measurement. Proceedings of the 5th IEEE International Conference on Cognitive Informatics July 2006 : pp. 226.

[9] D.S Kushwaha and A. K. Misra. A modified cognitive information complexity measure of software. ACM SIGSOFT Software Engineering Notes v.31 n.1, January 2006 : pp. 1.

[10] D.S Kushwaha and A. K. Misra. Improved cognitive information complexity measure: a metric that establishes program comprehension effort. ACM SIGSOFT Software Engineering Notes v.31 n.5, September 2006 : pp. 1.

[11] S. Misra. Modified Cognitive Complexity Measure. Computer and Information Sciences – ISCIS 2006, pp. 1050-1059. Springer Berlin / Heidelberg, October 2006.

[12] S. Misra. Cognitive Program Complexity Measure. Proceedings of the 6th IEEE

International Conference on Cognitive Informatics (ICCI'07) August 2007 : pp. 120.

[13] Yiyu Yao. The Art of Granular Computing. Proceedings of the International Conference on Rough Sets and Emerging Intelligent Systems Paradigms, LNAI 4585 2007 : pp. 101.

[14] Yiyu Yao. Granular Computing: Past, Present& Future. Proceedings of IEEE International Conference on Granular Computing 2008 : pp. 80.

[15] Yiyu Yao. A Unified Framework of Granular Computing. Handbook of Granular Computing, pp. 401-410. Wiley, 2008.

[16] Yiyu Yao. Structured writing with granular computing strategies. Proceedings of IEEE International Conference on Granular Computing 2007 : pp. 72.

[17] Benjapol Auprasert and Yachai Limpiyakorn. Underlying Cognitive Complexity Measure with Combinatorial Rules. Proceedings of World Academy of Science, Engineering and Technology Volume 45, November 2008, ISSN: 2070-3740 : pp 432-437.

[18] E. J. Weyuker. Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering v.14 n.9, September 1988 : pp.1357.

[19] C. Kaner and W. Bond. Software Engineering Metrics: What do they Measure and how do we know?. Proceedings of the 10th International Software Metrics Symposium, Metrics 2004 2004.

[20] Benjapol Auprasert and Yachai Limpiyakorn. Towards Structured Software Cognitive Complexity Measurement with Granular Computing Strategies. Proceedings of the 8th IEEE International Conference on Cognitive Informatics (ICCI 2009) June 2009 : pp 365-370.

[21] Benjapol Auprasert and Yachai Limpiyakorn. Representing source code with granular hierarchical structures. Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC 2009) May 2009 : pp 319-320.

[22] Benjapol Auprasert and Yachai Limpiyakorn. Structuring Cognitive Information for Software Complexity Measurement. Proceedings of the 1st WRI World Congress on Computer Science and Information Engineering (CSIE 2009) March 2009 : pp. 830-834.

[23] E. I. Oviedo. Control flow, data flow, and program complexity. Proceedings of COMPSAC 1980 : pp.146.

[24] Albrecht, A.J Gaffney, and J.E., Jr. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. IEEE Transactions on Software Engineering Volume SE-9 Issue 6, Nov. 1983.

[25] Joe Sacada, "The Basic of Counting", Lecture Notes: CIS 2910: Discrete Structures in Computer Science II, University of Guelph.

[26] A.T. Benjamin, J.J. Quinn. Proofs that Really Count: The Art of Combinatorial Proof, Washington, DC: Mathematical Association of America, 2003.

[27] Aigner, Martin, Ziegler, and Günter. Proofs from THE BOOK. Berlin; New York: Springer, 2003.

[28] M. H. A. Newman. On Theories with a Combinatorial Definition of "Equivalence". The Annals of Mathematics Second Series, Vol. 43, No. 2, Apr. 1942 : pp. 223-243.

[29] S. Misra and A. K. Misra. Evaluation and comparison of cognitive complexity measure. ACM SIGSOFT Software Engineering Notes v.32 n.2, Mar. 2007.

[30] S. Misra and A.K. Misra. Evaluating cognitive complexity measure with Weyuker properties. Proceedings of the 3rd IEEE International Conference on Cognitive Informatics August 2004.

[31] Ralf Laue. Experiments for Measuring Cognitive Weights for Software Control Structures. Proceedings of the 6th.IEEE International Conference on Cognitive Informatics 2007 : pp 116.

[32] David J. Eck. Introduction to Programming Using Java. Fifth Edition Version 5.02, November 2007.

[33] John C. Cherniavsky, Carl H. Smith. On Weyuker's Axioms for Software Complexity Measures. IEEE Transactions on Software Engineering Vol. 17, Issue 6, June 1991.

[34] Sanjay Misra. Modified Set of Weyuker's Properties. Proceedings of the 5th.IEEE International Conference on Cognitive Informatics 2006 : pp 242.

[35] Sanjay Misra and A.K. Misra. A proposed additional property to the Weyuker's existing properties. International Journal of Information Technology and Management Vol. 5, No.1, 2006 : pp. 66.

[36] Yingxu Wang. The Measurement Theory for Software Engineering. Proceedings of Canadian Conference on Electrical and Computer Engineering IEEE CCECE 2003 May 2003 : pp.4.

Appendix

Appendix A

Source Codes of Programs Used in Case Studies

Interest

```
/**
*This class implements a simple program that
*will compute the amount of interest that is
*earned on $17,000 invested at an interest
*rate of 0.07 for one year.  The interest and
*the value of the investment after one year are
*printed to standard output.
*/

public class Interest {

 public static void main(String[] args) {

  /*Declare the variables. */

  double principal;      //The value of the investment.
  double rate;           //The annual interest rate.
  double interest;       //Interest earned in one year.

  /*Do the computations. */

       principal = 17000;
       rate = 0.07;
       interest = principal * rate;    //Compute the interest.

       principal = principal + interest;
  //Compute value of investment after one year, with interest.
  //(Note: The new value replaces the old value of principal.)

  /*Output the results. */

  System.out.print("The interest earned is $");
  System.out.println(interest);
  System.out.print("The value of the investment after one year is $");
  System.out.println(principal);

 }//end of main()

}//end of class Interest
```

PrintSquare

```
/**
*A program that reads an integer that is typed in by the
*user and computes and prints the square of that integer.
*/

public class PrintSquare {

 public static void main(String[] args) {
```

```
  int userInput;  //The number input by the user.
  int square;     //The userInput, multiplied by itself.

  System.out.print("Please type a number: ");
       userInput = TextIO.getInt();
       square = userInput * userInput;
  System.out.print("The square of that number is ");
  System.out.println(square);

 }//end of main()

}//end of class PrintSquare
```

Interest2

```
/**
*This class implements a simple program that will compute
*the amount of interest that is earned on an investment over
*a period of one year.  The initial amount of the investment
*and the interest rate are input by the user.  The value of
*the investment at the end of the year is output.  The
*rate must be input as a decimal, not a percentage (for
*example, 0.05 rather than 5).
*/

public class Interest2 {

 public static void main(String[] args) {

  double principal;  //The value of the investment.
  double rate;       //The annual interest rate.
  double interest;   //The interest earned during the year.

       TextIO.put("Enter the initial investment: ");
       principal = TextIO.getlnDouble();

       TextIO.put("Enter the annual interest rate (decimal, not
percentage!): ");
       rate = TextIO.getlnDouble();

       interest = principal * rate;       //Compute this year's
interest.
       principal = principal + interest;  //Add it to principal.

       TextIO.put("The value of the investment after one year is $");
       TextIO.putln(principal);

 }//end of main()

}//end of class Interest2
```

CreateProfile

```
public class CreateProfile {

 public static void main(String[] args) {

  String name;     //The user's name.
  String email;    //The user's email address.
  double salary;   //the user's yearly salary.
  String favColor; //The user's favorite color.

      TextIO.putln("Good Afternoon!  This program will create");
      TextIO.putln("your profile file, if you will just answer");
      TextIO.putln("a few simple questions.");
      TextIO.putln();

  /*Gather responses from the user. */

      TextIO.put("What is your name?           ");
      name = TextIO.getln();
      TextIO.put("What is your email address?  ");
      email = TextIO.getln();
      TextIO.put("What is your yearly income?  ");
      salary = TextIO.getlnDouble();
      TextIO.put("What is your favorite color? ");
      favColor = TextIO.getln();

  /*Write the user's information to the file named profile.txt. */

        TextIO.writeFile("profile.txt");  //subsequent output goes to
the file
        TextIO.putln("Name:           "+name);
        TextIO.putln("Email:          "+email);
        TextIO.putln("Favorite Color: "+favColor);
        TextIO.putf( "Yearly Income:  %12f\n", salary);
     //The "/n" in the previous line is a carriage return.

  /*Print a final message to standard output. */

      TextIO.writeStandardOutput();
      TextIO.putln("Thank you.  Your profile has been written to
profile.txt.");

 }

}
```

Interest3

```
public class Interest3 {

 /*
      This class implements a simple program that will compute the
amount of
      interest that is earned on an investment over a period of 5
years.  The
```

```
      initial amount of the investment and the interest rate are
input by the
      user.  The value of the investment at the end of each year is
output.
 */

 public static void main(String[] args) {

  double principal;  // The value of the investment.
  double rate;       // The annual interest rate.

  /* Get the initial investment and interest rate from the user. */

      TextIO.put("Enter the initial investment: ");
      principal = TextIO.getlnDouble();

      TextIO.putln();
      TextIO.putln("Enter the annual interest rate.");
      TextIO.put("Enter a decimal, not a percentage: ");
      rate = TextIO.getlnDouble();
      TextIO.putln();

  /* Simulate the investment for 5 years. */

  int years;  // Counts the number of years that have passed.

      years = 0;
  while (years < 5) {
   double interest;  // Interest for this year.
         interest = principal * rate;
         principal = principal + interest;      // Add it to principal.
         years = years + 1;     // Count the current year.
   System.out.print("The value of the investment after ");
   System.out.print(years);
   System.out.print(" years is $");
   System.out.printf("%1.2f", principal);
   System.out.println();
  } // end of while loop

 } // end of main()

} // end of class Interest3
```

ComputeAverage

```
/*
 * This program reads a sequence of positive integers input
 * by the user, and it will print out the average of those
 * integers.  The user is prompted to enter one integer at a
 * time.  The user must enter a 0 to mark the end of the
 * data.  (The zero is not counted as part of the data to
 * be averaged.)  The program does not check whether the
 * user's input is positive, so it will actually work for
 * both positive and negative input values.
```

```
*/

public class ComputeAverage {

 public static void main(String[] args) {

  int inputNumber;    // One of the integers input by the user.
  int sum;            // The sum of the positive integers.
  int count;          // The number of positive integers.
  double average;     // The average of the positive integers.

  /* Initialize the summation and counting variables. */

       sum = 0;
       count = 0;

  /* Read and process the user's input. */

       TextIO.put("Enter your first positive integer: ");
       inputNumber = TextIO.getlnInt();

  while (inputNumber != 0) {
          sum += inputNumber;    // Add inputNumber to running sum.
          count++;               // Count the input by adding 1 to count.
          TextIO.put("Enter your next positive integer, or 0 to end: ");
          inputNumber = TextIO.getlnInt();
  }

  /* Display the result. */

  if (count == 0) {
          TextIO.putln("You didn't enter any data!");
  }
  else {
          average = ((double)sum) / count;
          TextIO.putln();
          TextIO.putln("You entered " + count + " positive integers.");
          TextIO.putf("Their average is %1.3f.\n", average);
  }

 } // end main()

} // end class ComputeAverage
```

CountDivisors

```
/**
 * This program reads a positive integer from the user.
 * It counts how many divisors that number has, and
 * then it prints the result.
 */

public class CountDivisors {
```

```
 public static void main(String[] args) {

  int N;  //A positive integer entered by the user.
    //Divisors of this number will be counted.

  int testDivisor;  //A number between 1 and N that is a
       //possible divisor of N.

  int divisorCount;  //Number of divisors of N that have been found.

  int numberTested;  //Used to count how many possible divisors
       //of N have been tested.  When the number
       //reaches 1000000, a period is output and
       //the value of numberTested is reset to zero.

  /*Get a positive integer from the user. */

  while(true){
          TextIO.put("Enter a positive integer: ");
          N = TextIO.getlnInt();
   if(N > 0)
    break;
          TextIO.putln("That number is not positive.  Please try
again.");
  }

  /*Count the divisors, printing a "." after every 1000000 tests. */

       divisorCount = 0;
       numberTested = 0;

  for(testDivisor = 1; testDivisor <= N; testDivisor++) {
   if(N % testDivisor == 0)
             divisorCount++;
          numberTested++;
   if(numberTested == 1000000) {
             TextIO.put('.');
             numberTested = 0;
   }
  }

  /*Display the result. */

       TextIO.putln();
       TextIO.putln("The number of divisors of "+N
        +"is "+divisorCount);

 }//end main()

}//end class CountDivisors
```

ListLetters

```
/**
```

```
*This program reads a line of text entered by the user.
*It prints a list of the letters that occur in the text,
*and it reports how many different letters were found.
*/

public class ListLetters {

 public static void main(String[]args) {

  String str;  //Line of text entered by the user.
  int count;   //Number of different letters found in str.
  char letter; //A letter of the alphabet.

       TextIO.putln("Please type in a line of text.");
       str = TextIO.getln();

       str = str.toUpperCase();

       count = 0;
       TextIO.putln("Your input contains the following letters:");
       TextIO.putln();
       TextIO.put(" ");
  for(letter = 'A'; letter <= 'Z'; letter++ ) {
   int i;  //Position of a character in str.
   for(i = 0; i < str.length(); i++ ) {
    if(letter == str.charAt(i) ) {
                  TextIO.put(letter);
                  TextIO.put(' ');
                  count++;
     break;
    }
   }
  }

       TextIO.putln();
       TextIO.putln();
       TextIO.putln("There were "+count + "different letters.");

 }//end main()

}//end class ListLetters
```

GuessingGame

```
/**
*This program lets the user play one or more guessing games.  In each
*game, the computer selects a number in the range 1 to 100. The use
*tries to guess the numbers.  If the user's guess is right, the user
*wins the game.  If the user makes six incorrect guesses, the user
*loses the game.   The computer tells the user whether his guess is
*high or low.  After each game, the computer askes the user whether
*the user wants to play again.
*/
```

```
public class GuessingGame {

 public static void main(String[] args) {
      TextIO.putln("Let's play a game.  I'll pick a number between");
      TextIO.putln("1 and 100, and you try to guess it.");
  boolean playAgain;
  do {
         playGame();  //call subroutine to play one game
         TextIO.put("Would you like to play again? ");
         playAgain = TextIO.getlnBoolean();
  } while (playAgain);
      TextIO.putln("Thanks for playing.  Goodbye.");
 } //end of main()


 /**
  * This subroutine lets the user play one guessing game and tells
  * the user whether he won or lost.
  */
 static void playGame() {
  int computersNumber; //A random number picked by the computer.
  int usersGuess;      //A number entered by user as a guess.
  int guessCount;      //Number of guesses the user has made.
       computersNumber = (int)(100 * Math.random()) + 1;
    //The value assigned to computersNumber is a randomly
    // chosen integer between 1 and 100, inclusive.
       guessCount = 0;
       TextIO.putln();
       TextIO.put("What is your first guess? ");
  while (true) {
         usersGuess = TextIO.getInt();  //Get the user's guess.
         guessCount++;
   if (usersGuess == computersNumber) {
            TextIO.putln("You got it in " + guessCount
      + "guesses!  My number was " + computersNumber);
    break;  //The game is over; the user has won.
   }
   if (guessCount == 6) {
            TextIO.putln("You didn't get the number in 6 guesses.");
            TextIO.putln("You lose.  My number was " +
computersNumber);
    break;  //The game is over; the user has lost.
   }
   //If we get to this point, the game continues.
   //Tell the user if the guess was too high or too low.
   if (usersGuess < computersNumber)
            TextIO.put("That's too low.  Try again: ");
   else if (usersGuess > computersNumber)
            TextIO.put("That's too high.  Try again: ");
  }
       TextIO.putln();
 } //end of playGame()
```

```
}//end of class GuessingGame
```

HighLow

```
/**
*This program lets the user play HighLow, a simple card game
*that is described in the output statements at the beginning of
*the main() routine.  After the user plays several games,
*the user's average score is reported.
*/

public class HighLow {


 public static void main(String[] args) {

  System.out.println("This program lets you play the simple card
game,");
  System.out.println("HighLow.  A card is dealt from a deck of
cards.");
  System.out.println("You have to predict whether the next card will
be");
  System.out.println("higher or lower.  Your score in the game is
the");
  System.out.println("number of correct predictions you make before");
  System.out.println("you guess wrong.");
  System.out.println();

  int gamesPlayed = 0;     //Number of games user has played.
  int sumOfScores = 0;     //The sum of all the scores from
         //  all the games played.
  double averageScore;     //Average score, computed by dividing
         //  sumOfScores by gamesPlayed.
  boolean playAgain;       //Record user's response when user is
         // asked whether he wants to play
         // another game.

  do {
   int scoreThisGame;        //Score for one game.
         scoreThisGame = play();   //Play the game and get the score.
         sumOfScores += scoreThisGame;
         gamesPlayed++;
         TextIO.put("Play again? ");
         playAgain = TextIO.getlnBoolean();
  } while (playAgain);

      averageScore = ((double)sumOfScores) / gamesPlayed;

  System.out.println();
  System.out.println("You played " + gamesPlayed + "games.");
  System.out.printf("Your average score was %1.3f.\n", averageScore);

 } //end main()
```

```
/**
 *Lets the user play one game of HighLow, and returns the
 *user's score on that game.  The score is the number of
 *correct guesses that the user makes.
 */
private static int play() {

      Deck deck = new Deck();  //Get a new deck of cards, and
        // store a reference to it in
        // the variable, deck.

      Card currentCard;  //The current card, which the user sees.

      Card nextCard;   //The next card in the deck.  The user tries
       // to predict whether this is higher or lower
       // than the current card.

 int correctGuesses ;  //The number of correct predictions the
       // user has made.  At the end of the game,
       // this will be the user's score.

 char guess;   //The user's guess.  'H' if the user predicts that
     // the next card will be higher, 'L' if the user
     // predicts that it will be lower.

      deck.shuffle();  //Shuffle the deck into a random order before
       // starting the game.

      correctGuesses = 0;
      currentCard = deck.dealCard();
      TextIO.putln("The first card is the "+currentCard);

 while(true){ //Loop ends when user's prediction is wrong.

  /*Get the user's prediction, 'H' or 'L' (or 'h' or 'l'). */

         TextIO.put("Will the next card be higher (H) or lower (L)?
");
   do{
            guess = TextIO.getlnChar();
            guess = Character.toUpperCase(guess);
    if(guess != 'H' && guess != 'L')
               TextIO.put("Please respond with H or L:  ");
   }while(guess != 'H' && guess != 'L');

  /*Get the next card and show it to the user. */

         nextCard = deck.dealCard();
         TextIO.putln("The next card is "+nextCard);

  /*Check the user's prediction. */
```

```
   if(nextCard.getValue() == currentCard.getValue()) {
            TextIO.putln("The value is the same as the previous
card.");
            TextIO.putln("You lose on ties.  Sorry!");
    break;  //End the game.
   }
   else if(nextCard.getValue() > currentCard.getValue()) {
    if(guess == 'H'){
                TextIO.putln("Your prediction was correct.");
                correctGuesses++;
    }
    else {
                TextIO.putln("Your prediction was incorrect.");
     break;  //End the game.
    }
   }
   else { //nextCard is lower
    if(guess == 'L'){
                TextIO.putln("Your prediction was correct.");
                correctGuesses++;
    }
    else {
                TextIO.putln("Your prediction was incorrect.");
     break;  //End the game.
    }
   }

   /*To set up for the next iteration of the loop, the nextCard
          becomes the currentCard, since the currentCard has to be
          the card that the user sees, and the nextCard will be
          set to the next card in the deck after the user makes
          his prediction.  */

         currentCard = nextCard;
         TextIO.putln();
         TextIO.putln("The card is "+currentCard);

  }//end of while loop

      TextIO.putln();
      TextIO.putln("The game is over.");
      TextIO.putln("You made "+correctGuesses
    +"correct predictions.");
      TextIO.putln();

  return correctGuesses;

 }//end play()


}//end class HighLow
```

ReverseInputNumbers

```
/**
```

```
*This program reads some positive integers from the user and
*then prints them in reverse order.  The numbers are stored
*in an array.
*/
public class ReverseInputNumbers {

 public static void main(String[] args) {

  int[] numbers;  //An array for storing the input values.
  int numCount;   //The number of numbers saved in the array.
  int num;        //One of the numbers input by the user.

      numbers = new int[100];   //Space for 100 ints.
      numCount = 0;             //No numbers have been saved yet.

      TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

  while(true){ //Get the numbers and put them in the array.
         TextIO.put("? ");
         num = TextIO.getlnInt();
   if(num <= 0)
    break;
         numbers[numCount] = num;
         numCount++;
  }

      TextIO.putln("\nYour numbers in reverse order are:\n");

  for(int i = numCount - 1; i >= 0; i--) {
          TextIO.putln( numbers[i] );
  }

 }//end main();

}//end class ReverseInputNumbers
```

CheckersData

```
 /**
 *An object of this class holds data about a game of checkers.
 *It knows what kind of piece is on each square of the checkerboard.
 *Note that RED moves "up" the board (i.e. row number decreases)
 *while BLACK moves "down" the board (i.e. row number increases).
 *Methods are provided to return lists of available legal moves.
 */
 private static class CheckersData {

  /* The following constants represent the possible contents of a
square
          on the board.  The constants RED and BLACK also represent
players
          in the game. */

  static final int
                EMPTY = 0,
```

```
                RED = 1,
                RED_KING = 2,
                BLACK = 3,
                BLACK_KING = 4;


int[][] board;  //board[r][c] is the contents of row r, column c.


/**
 * Constructor.  Create the board and set it up for a new game.
 */
     CheckersData() {
        board = new int[8][8];
        setUpGame();
}


/**
 * Set up the board with checkers in position for the beginning
 * of a game.  Note that checkers can only be found in squares
 * that satisfy  row % 2 == col % 2.  At the start of the game,
 * all such squares in the first three rows contain black squares
 * and all such squares in the last three rows contain red squares.
 */
void setUpGame() {
 for(int row = 0; row < 8; row++) {
  for(int col = 0; col < 8; col++) {
   if(row % 2 == col % 2){
    if(row < 3)
                    board[row][col] = BLACK;
    else if(row > 4)
                    board[row][col] = RED;
    else
                    board[row][col] = EMPTY;
   }
   else {
                 board[row][col] = EMPTY;
   }
  }
 }
} //end setUpGame()


/**
 * Return the contents of the square in the specified row and column.
 */
int pieceAt(int row, int col) {
 return board[row][col];
}


/**
 * Set the contents of the square in the specified row and column.
 * piece must be one of the constants EMPTY, RED, BLACK, RED_KING,
 * BLACK_KING.
```

```
   */
   void setPieceAt(int row, int col, int piece) {
          board[row][col] = piece;
   }


   /**
   *Make the specified move.  It is assumed that move
   *is non-null and that the move it represents is legal.
   */
   void makeMove(CheckersMove move) {
          makeMove(move.fromRow, move.fromCol, move.toRow,
move.toCol);
   }


   /**
   *Make the move from (fromRow,fromCol) to (toRow,toCol).  It is
   *assumed that this move is legal.  If the move is a jump, the
   *jumped piece is removed from the board.  If a piece moves
   *the last row on the opponent's side of the board, the
   *piece becomes a king.
   */
   void makeMove(int fromRow, int fromCol, int toRow, int toCol) {
          board[toRow][toCol] = board[fromRow][fromCol];
          board[fromRow][fromCol] = EMPTY;
    if(fromRow - toRow == 2 || fromRow - toRow == -2){
     //The move is a jump.  Remove the jumped piece from the board.
     int jumpRow = (fromRow + toRow) / 2;  //Row of the jumped piece.
     int jumpCol = (fromCol + toCol) / 2;  //Column of the jumped piece.
            board[jumpRow][jumpCol] = EMPTY;
    }
    if(toRow == 0 && board[toRow][toCol] == RED)
            board[toRow][toCol] = RED_KING;
    if(toRow == 7 && board[toRow][toCol] == BLACK)
            board[toRow][toCol] = BLACK_KING;
   }

   /**
   *Return an array containing all the legal CheckersMoves
   *for the specfied player on the current board.  If the player
   *has no legal moves, null is returned.  The value of player
   *should be one of the constants RED or BLACK; if not, null
   *is returned.  If the returned value is non-null, it consists
   *entirely of jump moves or entirely of regular moves, since
   *if the player can jump, only jumps are legal moves.
   */
       CheckersMove[] getLegalMoves(int player) {

    if(player != RED && player != BLACK)
     return null;

    int playerKing;  //The constant representing a King belonging to
player.
    if(player == RED)
```

```
            playerKing = RED_KING;
   else
            playerKing = BLACK_KING;

   ArrayList<CheckersMove> moves = new ArrayList<CheckersMove>();  //
Moves will be stored in this list.

   /* First, check for any possible jumps.  Look at each square on the
board.
          If that square contains one of the player's pieces, look at
a possible
          jump in each of the four directions from that square.  If
there is
          a legal jump in that direction, put it in the moves
ArrayList.
   */

   for(int row = 0; row < 8; row++) {
    for(int col = 0; col < 8; col++) {
     if(board[row][col] == player || board[row][col] == playerKing) {
      if(canJump(player, row, col, row+1, col+1, row+2, col+2))
                    moves.add(new CheckersMove(row, col, row+2,
col+2));
      if(canJump(player, row, col, row-1, col+1, row-2, col+2))
                    moves.add(new CheckersMove(row, col, row-2,
col+2));
      if(canJump(player, row, col, row+1, col-1, row+2, col-2))
                    moves.add(new CheckersMove(row, col, row+2, col-
2));
      if(canJump(player, row, col, row-1, col-1, row-2, col-2))
                    moves.add(new CheckersMove(row, col, row-2, col-
2));
     }
    }
   }

   /* If any jump moves were found, then the user must jump, so we
don't
          add any regular moves.  However, if no jumps were found,
check for
          any legal regualar moves.  Look at each square on the
board.
          If that square contains one of the player's pieces, look at
a possible
          move in each of the four directions from that square.  If
there is
          a legal move in that direction, put it in the moves
ArrayList.
   */

   if(moves.size() == 0){
    for(int row = 0; row < 8; row++) {
     for(int col = 0; col < 8; col++) {
      if(board[row][col] == player || board[row][col] == playerKing) {
       if(canMove(player,row,col,row+1,col+1))
```

```
                        moves.add(new
CheckersMove(row,col,row+1,col+1));
      if(canMove(player,row,col,row-1,col+1))
                        moves.add(new CheckersMove(row,col,row-
1,col+1));
      if(canMove(player,row,col,row+1,col-1))
                        moves.add(new CheckersMove(row,col,row+1,col-
1));
      if(canMove(player,row,col,row-1,col-1))
                        moves.add(new CheckersMove(row,col,row-1,col-
1));
      }
     }
    }
   }

   /*If no legal moves have been found, return null.  Otherwise,
create
          an array just big enough to hold all the legal moves, copy
the
          legal moves from the ArrayList into the array, and return
the array. */

   if(moves.size() == 0)
    return null;
   else {
            CheckersMove[] moveArray = new
CheckersMove[moves.size()];
    for(int i = 0; i < moves.size(); i++)
               moveArray[i] = moves.get(i);
    return moveArray;
   }

  } //end getLegalMoves


  /**
  *Return a list of the legal jumps that the specified player can
  *make starting from the specified row and column.  If no such
  *jumps are possible, null is returned.  The logic is similar
  *to the logic of the getLegalMoves() method.
  */
      CheckersMove[] getLegalJumpsFrom(int player, int row, int col)
{
   if(player != RED && player != BLACK)
    return null;
   int playerKing;  //The constant representing a King belonging to
player.
   if(player == RED)
            playerKing = RED_KING;
   else
            playerKing = BLACK_KING;
   ArrayList<CheckersMove> moves = new ArrayList<CheckersMove>();  //
The legal jumps will be stored in this list.
   if(board[row][col] == player || board[row][col] == playerKing) {
    if(canJump(player, row, col, row+1, col+1, row+2, col+2))
```

```
                moves.add(new CheckersMove(row, col, row+2, col+2));
    if(canJump(player, row, col, row-1, col+1, row-2, col+2))
                moves.add(new CheckersMove(row, col, row-2, col+2));
    if(canJump(player, row, col, row+1, col-1, row+2, col-2))
                moves.add(new CheckersMove(row, col, row+2, col-2));
    if(canJump(player, row, col, row-1, col-1, row-2, col-2))
                moves.add(new CheckersMove(row, col, row-2, col-2));
   }
   if(moves.size() == 0)
    return null;
   else {
            CheckersMove[] moveArray = new
CheckersMove[moves.size()];
    for(int i = 0; i < moves.size(); i++)
                moveArray[i] = moves.get(i);
    return moveArray;
   }
  } //end getLegalMovesFrom()


  /**
  *This is called by the two previous methods to check whether the
  *player can legally jump from (r1,c1) to (r3,c3). It is assumed
  *that the player has a piece at (r1,c1), that (r3,c3) is a position
  *that is 2 rows and 2 columns distant from (r1,c1) and that
  *(r2,c2) is the square between (r1,c1) and (r3,c3).
  */
  private boolean canJump(int player, int r1, int c1, int r2, int c2,
int r3, int c3) {

   if(r3 < 0 || r3 >= 8 || c3 < 0 || c3 >= 8)
    return false;  //(r3,c3) is off the board.

   if(board[r3][c3] != EMPTY)
    return false;  //(r3,c3) already contains a piece.

   if(player == RED) {
    if(board[r1][c1] == RED && r3 > r1)
     return false;  //Regular red piece can only move  up.
    if(board[r2][c2] != BLACK && board[r2][c2] != BLACK_KING)
     return false;  //There is no black piece to jump.
    return true;  //The jump is legal.
   }
   else {
    if(board[r1][c1] == BLACK && r3 < r1)
     return false;  //Regular black piece can only move downn.
    if(board[r2][c2] != RED && board[r2][c2] != RED_KING)
     return false;  //There is no red piece to jump.
    return true;  //The jump is legal.
   }

  } //end canJump()
```

```
  /**
   * This is called by the getLegalMoves() method to determine whether
   * the player can legally move from (r1,c1) to (r2,c2). It is
   * assumed that (r1,r2) contains one of the player's pieces and
   * that (r2,c2) is a neighboring square.
   */
  private boolean canMove(int player, int r1, int c1, int r2, int c2) {

    if (r2 < 0 || r2 >= 8 || c2 < 0 || c2 >= 8)
      return false;  // (r2,c2) is off the board.

    if (board[r2][c2] != EMPTY)
      return false;  // (r2,c2) already contains a piece.

    if (player == RED) {
      if (board[r1][c1] == RED && r2 > r1)
        return false;  // Regualr red piece can only move down.
      return true;  // The move is legal.
    }
    else {
      if (board[r1][c1] == BLACK && r2 < r1)
        return false;  // Regular black piece can only move up.
      return true;  // The move is legal.
    }

  } // end canMove()


} // end class CheckersData
```

Board

```
/**
 * This panel displays a 160-by-160 checkerboard pattern with
 * a 2-pixel black border.  It is assumed that the size of the
 * canvas is set to exactly 164-by-164 pixels.  This class does
 * the work of letting the users play checkers, and it displays
 * the checkerboard.
 */
private class Board extends JPanel implements ActionListener,
MouseListener {


    CheckersData board;  // The data for the checkers board is kept
here.
     // This board is also responsible for generating
     // lists of legal moves.

  boolean gameInProgress; // Is a game currently in progress?

  /* The next three variables are valid only when the game is in
progress. */
```

```
  int currentPlayer;      //Whose turn is it now?  The possible
values
        // are CheckersData.RED and CheckersData.BLACK.

  int selectedRow, selectedCol;  //If the current player has selected
a piece to
         // move, these give the row and column
         // containing that piece.  If no piece is
         // yet selected, then selectedRow is -1.

      CheckersMove[] legalMoves;  //An array containing the legal
moves for the
         // current player.


  /**
  *Constructor.  Create the buttons and lable.  Listens for mouse
  *clicks and for clicks on the buttons.  Create the board and
  *start the first game.
  */
      Board() {
         setBackground(Color.BLACK);
         addMouseListener(this);
         resignButton = new JButton("Resign");
         resignButton.addActionListener(this);
         newGameButton = new JButton("New Game");
         newGameButton.addActionListener(this);
         message = new JLabel("",JLabel.CENTER);
         message.setFont(new Font("Serif", Font.BOLD, 14));
         message.setForeground(Color.green);
         board = new CheckersData();
         doNewGame();
  }


  /**
  *Respond to user's click on one of the two buttons.
  */
  public void actionPerformed(ActionEvent evt) {
   Object src = evt.getSource();
   if(src == newGameButton)
            doNewGame();
   else if(src == resignButton)
            doResign();
  }


  /**
  *Start a new game
  */
  void doNewGame() {
   if(gameInProgress == true){
     //This should not be possible, but it doens't hurt to check.
            message.setText("Finish the current game first!");
```

```
    return;
   }
          board.setUpGame();   //Set up the pieces.
          currentPlayer = CheckersData.RED;   //RED moves first.
          legalMoves = board.getLegalMoves(CheckersData.RED);  //Get
RED's legal moves.
          selectedRow = -1;   //RED has not yet selected a piece to
move.
          message.setText("Red:  Make your move.");
          gameInProgress = true;
          newGameButton.setEnabled(false);
          resignButton.setEnabled(true);
          repaint();
  }


  /**
  *Current player resigns.  Game ends.  Opponent wins.
  */
  void doResign() {
   if(gameInProgress == false){
             message.setText("There is no game in progress!");
    return;
   }
   if(currentPlayer == CheckersData.RED)
             gameOver("RED resigns.  BLACK wins.");
   else
             gameOver("BLACK resigns.  RED wins.");
  }


  /**
  *The game ends.  The parameter, str, is displayed as a message
  *to the user.  The states of the buttons are adjusted so playes
  *can start a new game.  This method is called when the game
  *ends at any point in this class.
  */
  void gameOver(String str) {
          message.setText(str);
          newGameButton.setEnabled(true);
          resignButton.setEnabled(false);
          gameInProgress = false;
  }


  /**
  *This is called by mousePressed() when a player clicks on the
  *square in the specified row and col.  It has already been checked
  *that a game is, in fact, in progress.
  */
  void doClickSquare(int row, int col) {

   /*If the player clicked on one of the pieces that the player
          can move, mark this row and col as selected and return.
(This
```

```
            might change a previous selection.)  Reset the message,
in
            case it was previously displaying an error message. */

   for(int i = 0; i < legalMoves.length; i++)
    if(legalMoves[i].fromRow == row && legalMoves[i].fromCol == col) {
                selectedRow = row;
                selectedCol = col;
     if(currentPlayer == CheckersData.RED)
                   message.setText("RED:  Make your move.");
     else
                   message.setText("BLACK:  Make your move.");
                repaint();
     return;
    }

   /*If no piece has been selected to be moved, the user must first
            select a piece.  Show an error message and return. */

   if(selectedRow < 0){
             message.setText("Click the piece you want to move.");
    return;
   }

   /*If the user clicked on a squre where the selected piece can be
            legally moved, then make the move and return. */

   for(int i = 0; i < legalMoves.length; i++)
    if(legalMoves[i].fromRow == selectedRow && legalMoves[i].fromCol
== selectedCol
                   && legalMoves[i].toRow == row &&
legalMoves[i].toCol == col) {
                doMakeMove(legalMoves[i]);
     return;
    }

   /*If we get to this point, there is a piece selected, and the
square where
            the user just clicked is not one where that piece can be
legally moved.
            Show an error message. */

          message.setText("Click the square you want to move to.");

  } //end doClickSquare()


  /**
  *This is called when the current player has chosen the specified
  *move.  Make the move, and then either end or continue the game
  *appropriately.
  */
  void doMakeMove(CheckersMove move) {

          board.makeMove(move);

   /*If the move was a jump, it's possible that the player has another
```

```
            jump.  Check for legal jumps starting from the square
that the player
            just moved to.  If there are any, the player must jump.
The same
            player continues moving.
      */

      if(move.isJump()) {
               legalMoves =
board.getLegalJumpsFrom(currentPlayer,move.toRow,move.toCol);
       if(legalMoves != null){
        if(currentPlayer == CheckersData.RED)
                     message.setText("RED:  You must continue jumping.");
        else
                     message.setText("BLACK:  You must continue
jumping.");
                  selectedRow = move.toRow;  //Since only one piece can
be moved, select it.
                  selectedCol = move.toCol;
                  repaint();
        return;
       }
      }

     /*The current player's turn is ended, so change to the other
player.
            Get that player's legal moves.  If the player has no legal
moves,
            then the game ends. */

      if(currentPlayer == CheckersData.RED) {
               currentPlayer = CheckersData.BLACK;
               legalMoves = board.getLegalMoves(currentPlayer);
       if(legalMoves == null)
                  gameOver("BLACK has no moves.  RED wins.");
       else if(legalMoves[0].isJump())
                  message.setText("BLACK:  Make your move.  You must
jump.");
       else
                  message.setText("BLACK:  Make your move.");
      }
      else {
               currentPlayer = CheckersData.RED;
               legalMoves = board.getLegalMoves(currentPlayer);
       if(legalMoves == null)
                  gameOver("RED has no moves.  BLACK wins.");
       else if(legalMoves[0].isJump())
                  message.setText("RED:  Make your move.  You must
jump.");
       else
                  message.setText("RED:  Make your move.");
      }

     /*Set selectedRow = -1 to record that the player has not yet
selected
            a piece to move. */
```

```
            selectedRow = -1;

      /*As a courtesy to the user, if all legal moves use the same piece,
then
             select that piece automatically so the use won't have to
click on it
             to select it. */

      if(legalMoves != null){
       boolean sameStartSquare = true;
       for(int i = 1; i < legalMoves.length; i++)
        if(legalMoves[i].fromRow != legalMoves[0].fromRow
                        || legalMoves[i].fromCol != legalMoves[0].fromCol)
{
                    sameStartSquare = false;
         break;
        }
       if(sameStartSquare) {
                  selectedRow = legalMoves[0].fromRow;
                  selectedCol = legalMoves[0].fromCol;
       }
      }

      /*Make sure the board is redrawn in its new state. */

            repaint();

    } //end doMakeMove();

    /**
    *Respond to a user click on the board.  If no game is in progress,
show
    *an error message.  Otherwise, find the row and column that the
user
    *clicked and call doClickSquare() to handle it.
    */
    public void mousePressed(MouseEvent evt) {
     if(gameInProgress == false)
               message.setText("Click \"New Game\"to start a new game.");
     else{
      int col = (evt.getX() - 2)/20;
      int row = (evt.getY() - 2)/20;
      if(col >= 0 && col < 8 && row >= 0 && row < 8)
                  doClickSquare(row,col);
     }
    }

    public void mouseReleased(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }


   } //end class Board
```

TowersOfHanoi

```
/**
*This program lists the steps in the solution of the TowersOfHanoi
*problem.  The number of disks to be moved is specified by the user.
*Warning:  The number of moves grows very quickly with the number of
*disks!
*/
public class TowersOfHanoi {

 public static void main(String[] args) {

  int N;  // The number of disks in the original stack,
    // as specified by the user.

      TextIO.putln("This applet will list the steps in the solution
of");
      TextIO.putln("the Towers of Hanoi problem.  You can specify
the");
      TextIO.putln("number of disks to be moved.  Try it for small
numbers");
      TextIO.putln("of disks, like 1, 2, 3, and 4.");
      TextIO.putln();
      TextIO.putln("How many disks are to be moved from Stack 0 to
Stack 1?");
      TextIO.putln();
      TextIO.put("? ");

      N = TextIO.getInt();

      TextIO.putln();
      TextIO.putln();

      TowersOfHanoi(N, 0, 1, 2);  // Print the solution.

 }

 /**
 *Solve the problem of moving the number of disks specified
 *by the first parameter from the stack specified by the
 *second parameter to the stack specified by the third
 *parameter.  The stack specified by the fourth parameter
 *is available for use as a spare.  Stacks are specified by
 *number: 0, 1, or 2.
 */
 static void TowersOfHanoi(int disks, int from, int to, int spare) {
  if (disks == 1) {
    // There is only one disk to be moved.  Just move it.
   System.out.println("Move a disk from stack number "
     + from + "to stack number " + to);
  }
  else {
```

```
    //Move all but one disk to the spare stack, then
    //move the bottom disk, then put all the other
    //disks on top of it.
          TtowersOfHanoi(disks-1, from, spare, to);
   System.out.println("Move a disk from stack number "
      +from + "to stack number "+to);
          TtowersOfHanoi(disks-1, spare, to, from);
  }
 }

}
```

SimpleParser3

```
/*
    This program reads standard expressions typed in by the user.
    The program constructs an expression tree to represent the
    exprssion.  It then prints the value of the tree.  It also uses
    the tree to print out a list of commands that could be used
    on a stack machine to evaluate the expresion.
    The expressions can use positive real numbers and
    the binary operators +, -, *, and /.  The unary minus operation
    is supported.  The expressions are defined by the BNF rules:

           <expression>  ::=  [ "-" ] <term> [ [ "+" | "-" ]
<term> ]...

           <term>  ::=  <factor> [ [ "*" | "/" ] <factor> ]...

           <factor>  ::=  <number>  |  "(" <expression> ")"

    A number must begin with a digit (i.e., not a decimal point).
    A line of input must contain exactly one such expression.  If
extra
    data is found on a line after an expression has been read, it is
    considered an error.

    In addtion to the main program class, SimpleParser3, this program
    defines a set of four nested classes for implementing expression
trees.

*/

public class SimpleParser3 {

// ----------------Nested classes for Expression Trees -----------------------
------


 /**
  * An abstract class representing any node in an expression tree.
  * The three concrete node classes are concrete subclasses.
  * Two instance methods are specified, so that they can be used with
  * any ExpNode.  The value() method returns the value of the
  * expression.  The printStackCommands() method prints a list
  * of commands that could be used to evaluate the expression on
```

```
 * a stack machine (assuming that the value of the expression is
 * to be left on the stack).
 */
abstract private static class ExpNode {
 abstract double value();
 abstract void printStackCommands();
}

/**
 *Represents en expression node that holds a number.
 */
private static class ConstNode extends ExpNode {
 double number;  //The number.
      ConstNode(double val) {
   //Construct a ConstNode containing the specified number.
         number = val;
 }
 double value() {
   //The value of the node is the number that it contains.
  return number;
 }
 void printStackCommands() {
   //On a stack machine, just push the number onto the stack.
         TextIO.putln(" Push "+number);
 }
}


/**
 *An expression node representing a binary operator,
 */
private static class BinOpNode extends ExpNode {
 char op;        //The operator.
      ExpNode left;   //The expression for its left operand.
      ExpNode right;  //The expression for its right operand.
      BinOpNode(char op, ExpNode left, ExpNode right) {
   //Construct a BinOpNode containing the specified data.
  assert op == '+' || op == '-' || op == '*' || op == '/';
  assert left != null && right != null;
  this.op = op;
  this.left = left;
  this.right = right;
 }
 double value() {
   //The value is obtained by evaluating the left and right
   //operands and combining the values with the operator.
  double x = left.value();
  double y = right.value();
  switch(op) {
  case '+': return x + y;
  case '-': return x - y;
  case '*': return x * y;
  case '/': return x / y;
  default: return Double.NaN;  //Bad operator!
  }
```

```
    }
    void  printStackCommands() {
      //To evalute the expression on a stack machine, first do
      //whatever is necessary to evaluate the left operand, leaving
      //the answer on the stack.  Then do the same thing for the
      //second operand.  Then apply the operator (which means popping
      //the operands, applying the operator, and pushing the result).
            left.printStackCommands();
            right.printStackCommands();
            TextIO.putln("Operator "+op);
    }
  }


  /**
   *An expression node to represent a unary minus operator.
   */
  private static class UnaryMinusNode extends ExpNode {
        ExpNode operand;  //The operand to which the unary minus
applies.
        UnaryMinusNode(ExpNode operand) {
      //Construct a UnaryMinusNode with the specified operand.
     assert operand != null;
     this.operand = operand;
    }
    double value() {
      //The value is the negative of the value of the operand.
     double neg = operand.value();
     return -neg;
    }
    void printStackCommands() {
      //To evaluate this expression on a stack machine, first do
      //whatever is necessary to evaluate the operand, leaving the
      //operand on the stack.  Then apply the unary minus (which means
      //popping the operand, negating it, and pushing the result).
            operand.printStackCommands();
            TextIO.putln("Unary minus");
    }
  }


// -----------------------------------------------------------------


  /**
   *An object of type ParseError represents a syntax error found in
   *the user's input.
   */
  private static class ParseError extends Exception {
        ParseError(String message) {
     super(message);
    }
  }//end nested class ParseError


  public static void main(String[] args) {
```

```
  while(true){
          TextIO.putln("\n\nEnter an expression, or press return to
end.");
          TextIO.put("\n?  ");
          TextIO.skipBlanks();
    if(TextIO.peek() == '\n')
     break;
    try{
              ExpNode exp = expressionTree();
              TextIO.skipBlanks();
     if(TextIO.peek() != '\n')
      throw new ParseError("Extra data after end of expression.");
              TextIO.getln();
              TextIO.putln("\nValue is "+exp.value());
              TextIO.putln("\nOrder of postfix evaluation is:\n");
              exp.printStackCommands();
    }
    catch(ParseError e) {
              TextIO.putln("\n*** Error in input:    "+e.getMessage());
              TextIO.putln("***Discarding input:  "+TextIO.getln());
    }
   }

       TextIO.putln("\n\nDone.");

  }//end main()


 /**
  *Reads an expression from the current line of input and builds
  *an expression tree that represents the expression.
  *@return an ExpNode which is a pointer to the root node of the
  * expression tree
  *@throws ParseError if a syntax error is found in the input
  */
 private static ExpNode expressionTree() throws ParseError {
       TextIO.skipBlanks();
  boolean negative;  //True if there is a leading minus sign.
       negative = false;
  if(TextIO.peek() == '-'){
          TextIO.getAnyChar();
          negative = true;
  }
       ExpNode exp;       //The expression tree for the expression.
       exp = termTree();  //Start with the first term.
  if(negative)
          exp = new UnaryMinusNode(exp);
       TextIO.skipBlanks();
  while(TextIO.peek() == '+' || TextIO.peek() == '-'){
    //Read the next term and combine it with the
    //previous terms into a bigger expression tree.
   char op = TextIO.getAnyChar();
          ExpNode nextTerm = termTree();
          exp = new BinOpNode(op, exp, nextTerm);
```

```
        TextIO.skipBlanks();
 }
 return exp;
}//end expressionTree()


/**
*Reads a term from the current line of input and builds
*an expression tree that represents the expression.
*@return an ExpNode which is a pointer to the root node of the
* expression tree
*@throws ParseError if a syntax error is found in the input
*/
private static ExpNode termTree() throws ParseError {
     TextIO.skipBlanks();
     ExpNode term;  //The expression tree representing the term.
     term = factorTree();
     TextIO.skipBlanks();
 while(TextIO.peek() == '*' || TextIO.peek() == '/'){
   //Read the next factor, and combine it with the
   //previous factors into a bigger expression tree.
  char op = TextIO.getAnyChar();
        ExpNode nextFactor = factorTree();
        term = new BinOpNode(op,term,nextFactor);
        TextIO.skipBlanks();
 }
 return term;
}//end termValue()


/**
*Reads a factor from the current line of input and builds
*an expression tree that represents the expression.
*@return an ExpNode which is a pointer to the root node of the
* expression tree
*@throws ParseError if a syntax error is found in the input
*/

private static ExpNode factorTree() throws ParseError {
     TextIO.skipBlanks();
 char ch = TextIO.peek();
 if(Character.isDigit(ch) ) {
   //The factor is a number.  Return a ConstNode.
  double num = TextIO.getDouble();
  return new ConstNode(num);
 }
 else if(ch == '('){
   //The factor is an expression in parentheses.
   //Return a tree representing that expression.
        TextIO.getAnyChar();  //Read the "("
        ExpNode exp = expressionTree();
        TextIO.skipBlanks();
  if(TextIO.peek() != ')')
   throw new ParseError("Missing right parenthesis.");
        TextIO.getAnyChar();  //Read the ")"
```

```
    return exp;
   }
   else if(ch == '\n')
    throw new ParseError("End-of-line encountered in the middle of an
expression.");
   else if(ch == ')')
    throw new ParseError("Extra right parenthesis.");
   else if(ch == '+' || ch == '-' || ch == '*' || ch == '/')
    throw new ParseError("Misplaced operator.");
   else
    throw new ParseError("Unexpected character \""+ch + "\"encountered.");
  } //end factorTree()


}//end class SimpleParser3
```

WordCount

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.TreeMap;
import java.util.Comparator;

/**
 * This program will read words from an input file, and count the
 * number of occurrences of each word.  The word data is written to
 * an output file twice, once with the words in alphabetical order
 * and once with the words ordered by number of occurrences.  The
 * user specifies the input file and the ouput file.
 *
 * The program demonstrates several parts of Java's framework for
 * generic programming:  TreeMap, List sorting, Comparators, etc.
 */
public class WordCount {


  /**
   *Represents the data we need about a word:  the word and
   *the number of times it has been encountered.
   */
  private static class WordData {
   String word;
   int count;
       WordData(String w) {
    //Constructor for creating a WordData object when
    //we encounter a new word.
          word = w;
          count = 1;  //The initial value of count is 1.
   }
  }//end class WordData


  /**
   *A comparator for comparing objects of type WordData according to
```

```
 *their counts.  This is used for sorting the list of words by
frequency.
 */
 private static class CountCompare implements Comparator<WordData> {
  public int compare(WordData data1, WordData data2){
   return data2.count - data1.count;
    //The return value is positive if data2.count > data1.count.
    //I.E., data1 comes after data2 in the ordering if there
    //were more occurrences of data2.word than of data1.word.
    //The words are sorted according to decreasing counts.
  }
 }//end class CountCompare


 public static void main(String[] args) {

  System.out.println("\n\n   This program will ask you to select an
input file");
  System.out.println("It make a list of all the words that occur in
the file");
  System.out.println("along with the number of time that each word
occurs.");
  System.out.println("This list will be output twice, first in
alphabetical,");
  System.out.println("then in order of frequency of occurrence with
the most");
  System.out.println("common word at the top and the least common at
the end.");
  System.out.println(" After reading the input file, the program asks
you to");
  System.out.println("select an output file.  If you select a file,
the list of");
  System.out.println("words will be written to that file; if you
cancel, the list");
  System.out.println("be written to standard output.  All words are
converted to");
  System.out.println("lower case.\n\n");
  System.out.print("Press return to begin.");
       TextIO.getln();  //Wait for user to press return.

  try {

   if(TextIO.readUserSelectedFile() == false){
    System.out.println("No input file selected.  Exiting.");
    System.exit(1);
   }

   //Create a TreeMap to hold the data.  Read the file and record
   //data in the map about the words that are found in the file.

   TreeMap<String,WordData> words = new TreeMap<String,WordData>();
   String word = readNextWord();
   while(word != null){
```

```
            word = word.toLowerCase();  //convert word to lower case
            WordData data = words.get(word);
    if(data == null)
               words.put( word, new WordData(word) );
    else
               data.count++;
            word = readNextWord();
   }

   System.out.println("Number of different words found in file:  "
     +words.size());
   System.out.println();
   if(words.size() == 0){
    System.out.println("No words found in file.");
    System.out.println("Exiting without saving data.");
    System.exit(0);
   }

   //Copy the word data into an array list, and sort the list
   //into order of decreasing frequency.

   ArrayList<WordData> wordsByFrequency = new
ArrayList<WordData>( words.values() );
   Collections.sort( wordsByFrequency, new CountCompare() );

   //Output the data from the map and from the list.

         TextIO.writeUserSelectedFile(); //If user cancels, output
automatically
           //goes to standard output.
         TextIO.putln(words.size() + "words found in file:\n");
         TextIO.putln("List of words in alphabetical order"
    +"(with counts in parentheses):\n");
   for(WordData data : words.values() )
            TextIO.putln(" "+data.word + "("+data.count + ")");
         TextIO.putln("\n\nList of words by frequency of
occurence:\n");
   for(WordData data : wordsByFrequency )
            TextIO.putln(" "+data.word + "("+data.count + ")");
   System.out.println("\n\nDone.\n\n");

  }
  catch(Exception e) {
   System.out.println("Sorry, an error has occurred.");
   System.out.println("Error Message:  "+e.getMessage());
         e.printStackTrace();
  }
  System.exit(0);  //Might be necessary, because of use of file dialogs.
 }

 /**
  *Read the next word from TextIO, if there is one.  First, skip past
  *any non-letters in the input.  If an end-of-file is encountered
before
```

```
 *a word is found, return null.  Otherwise, read and return the
word.
 *A word is defined as a sequence of letters.  Also, a word can
include
 *an apostrophe if the apostrophe is surrounded by letters on each
side.
 *@return the next word from TextIO, or null if an end-of-file is
encountered
 */
 private static String readNextWord() {
  char ch = TextIO.peek(); //Look at next character in input.
  while(ch != TextIO.EOF && ! Character.isLetter(ch)) {
          TextIO.getAnyChar();  //Read the character.
          ch = TextIO.peek();   //Look at the next character.
  }
  if(ch == TextIO.EOF) //Encountered end-of-file
   return null;
  //At this point, we know that the next character, so read a word.
  String word = "";  //This will be the word that is read.
  while(true){
          word += TextIO.getAnyChar();  //Append the letter onto word.
          ch = TextIO.peek();  //Look at next character.
   if(ch == '\''){
     //The next character is an apostrophe.  Read it, and
     //if the following character is a letter, add both the
     //apostrophe and the letter onto the word and continue
     //reading the word.  If the character after the apostrophe
     //is not a letter, the word is done, so break out of the loop.
             TextIO.getAnyChar();    //Read the apostrophe.
             ch = TextIO.peek();     //Look at char that follows
apostrophe.
    if(Character.isLetter(ch)) {
                word += "\'"+TextIO.getAnyChar();
                ch = TextIO.peek();  //Look at next char.
    }
    else
     break;
   }
   if(!Character.isLetter(ch) ) {
     //If the next character is not a letter, the word is
     //finished, so bread out of the loop.
    break;
   }
   //If we haven't broken out of the loop, next char is a letter.
  }
  return word;  //Return the word that has been read.
 }

}//end class WordCount
```

SimpleInterpreter

```
/*
```

```
    This program can evaluate expressions that can include
    numbers, variables, parentheses, and the operators +,
-, *, /, and ^ (where ^ indicates raising to a power).
    A variable name must consist of letters and digits,
    beginning with a letter.  Names are case-sensitive.
    This program accepts commands of two types from the user.
    For a command of the form  print <expression> , the expression
    is evaluated and the value is output.  For a command of
    the form  let <variable> = <expression> , the expression is
    evaluated and the value is assigned to the variable.
    If a variable is used in an expression before it has been
    assigned a value, an error occurs.  A number must begin with
    a digit (i.e., not a decimal point).

    Commands are formally defined by the BNF rules:

            <command>  ::=  "print" <expression>
                              |  "let" <variable> "=" <expression>

            <expression>  ::=  [ "-" ] <term> [ [ "+" | "-" ]
<term> ]...

            <term>  ::=  <factor> [ [ "*" | "/" ] <factor> ]...

            <factor>  ::=  <primary> [ "^" <primary> ]...

            <primary>  ::=  <number> | <variable> | "(" <expression>
")"

    A line of input must contain exactly one such command.  If extra
    data is found on a line after an expression has been read, it is
    considered an error.  The variables "pi" and "e" are defined
    when the program starts to represent the usual mathematical
    constants.

    This program demonstrates the use of a HashMap as a symbol
    table.

    SimpleInterpreter.java is based on the program SimpleParser2.java.
    It uses the non-standard class, TextIO.
*/

import java.util.HashMap;

public class SimpleInterpreter {

 /**
  *Represents a syntax error found in the user's input.
  */
 private static class ParseError extends Exception {
       ParseError(String message) {
   super(message);
  }
 }//end nested class ParseError


 /**
  *The symbolTable contains information about the
  *values of variables.  When a variable is assigned
```

```
 *a value, it is recorded in the symbol table.
 *The key is the name of the variable, and the
 *value is an object of type Double that contains
 *the value of the variable.  (The wrapper class
 *Double is used, since a HashMap cannot contain
 *objects belonging to the primitive type double.)
 */
private static HashMap<String,Double> symbolTable;


public static void main(String[] args) {

 //Create the map that represents symbol table.

      symbolTable = new HashMap<String,Double>();

 //To start, add variables named "pi" and "e" to the symbol
 //table.  Their values are the usual mathematical constants.

      symbolTable.put("pi", Math.PI);
      symbolTable.put("e", Math.E);

      TextIO.putln("\n\nEnter commands; press return to end.");
      TextIO.putln("Commands must have the form:\n");
      TextIO.putln("  print <expression>");
      TextIO.putln(" or");
      TextIO.putln("  let <variable> = <expression>");

 while(true){
        TextIO.put("\n?  ");
        TextIO.skipBlanks();
  if(TextIO.peek() == '\n'){
   break;  //A blank input line ends the while loop and the program.
  }
  try{
   String command = TextIO.getWord();
   if(command.equalsIgnoreCase("print"))
              doPrintCommand();
   else if(command.equalsIgnoreCase("let"))
              doLetCommand();
   else
    throw new ParseError("Command must begin with 'print' or 'let'.");
          TextIO.getln();
  }
  catch(ParseError e) {
          TextIO.putln("\n*** Error in input:    "+e.getMessage());
          TextIO.putln("*** Discarding input:  "+TextIO.getln());
  }
 }

      TextIO.putln("\n\nDone.");

}//end main()
```

```
/**
 *Process a command of the form  let <variable> = <expression>.
 *When this method is called, the word "let" has already
 *been read.  Read the variable name and the expression, and
 *store the value of the variable in the symbol table.
 */
private static void doLetCommand() throws ParseError {
     TextIO.skipBlanks();
 if(!Character.isLetter(TextIO.peek()) )
  throw new ParseError("Expected variable name after 'let'.");
 String varName = readWord();  //The name of the variable.
     TextIO.skipBlanks();
 if(TextIO.peek() != '=')
  throw new ParseError("Expected '=' operator for 'let' command.");
     TextIO.getChar();
 double val = expressionValue();  //The value of the variable.
     TextIO.skipBlanks();
 if(TextIO.peek() != '\n')
  throw new ParseError("Extra data after end of expression.");
     symbolTable.put( varName, val );  //Add to symbol table.
     TextIO.putln("ok");
}


/**
 *Process a command of the form  print <expression>.
 *When this method is called, the word "print" has already
 *been read.  Evaluate the expression and print the value.
 */
private static void doPrintCommand() throws ParseError {
 double val = expressionValue();
     TextIO.skipBlanks();
 if(TextIO.peek() != '\n')
  throw new ParseError("Extra data after end of expression.");
     TextIO.putln("Value is "+val);
}


/**
 *Read an expression from the current line of input and return its
value.
 */
private static double expressionValue() throws ParseError {
     TextIO.skipBlanks();
 boolean negative;  //True if there is a leading minus sign.
     negative = false;
 if(TextIO.peek() == '-'){
        TextIO.getAnyChar();
        negative = true;
 }
 double val;  //Value of the expression.
     val = termValue();  //An expression must start with a term.
 if(negative)
        val = -val; //Apply the leading minus sign
```

```
        TextIO.skipBlanks();
    while(TextIO.peek() == '+' || TextIO.peek() == '-'){
      //Read the next term and add it to or subtract it from
      //the value of previous terms in the expression.
     char op = TextIO.getAnyChar();
     double nextVal = termValue();
     if(op == '+')
              val += nextVal;
     else
              val -= nextVal;
           TextIO.skipBlanks();
    }
    return val;
}//end expressionValue()


/**
*Read a term from the current line of input and return its value.
*/
private static double termValue() throws ParseError {
        TextIO.skipBlanks();
    double val;  //The value of the term.
        val = factorValue();  //A term must start with a factor.
        TextIO.skipBlanks();
    while(TextIO.peek() == '*' || TextIO.peek() == '/'){
      //Read the next factor, and multiply or divide
      //the value-so-far by the value of this factor.
     char op = TextIO.getAnyChar();
     double nextVal = factorValue();
     if(op == '*')
              val *= nextVal;
     else
              val /= nextVal;
           TextIO.skipBlanks();
    }
    return val;
}//end termValue()


/**
*Read a factor from the current line of input and return its value.
*/
private static double factorValue() throws ParseError {
        TextIO.skipBlanks();
    double val;  //Value of the factor.
        val = primaryValue();  //A factor must start with a primary.
        TextIO.skipBlanks();
    while(TextIO.peek() == '^'){
      //Read the next primary, and exponentiate
      //the value-so-far by the value of this primary.
           TextIO.getChar();
     double nextVal = primaryValue();
           val = Math.pow(val,nextVal);
     if(Double.isNaN(val))
      throw new ParseError("Illegal values for ^ operator.");
           TextIO.skipBlanks();
```

```
 }
 return val;
}//end termValue()


/**
 * Read a primary from the current line of input and
 * return its value.  A primary must be a number,
 * a variable, or an expression enclosed in parentheses.
 */
private static double primaryValue() throws ParseError {
      TextIO.skipBlanks();
 char ch = TextIO.peek();
 if(Character.isDigit(ch) ) {
   //The factor is a number.  Read it and
   //return its value.
  return TextIO.getDouble();
 }
 else if(Character.isLetter(ch) ) {
   //The factor is a variable.  Read its name and
   //look up its value in the symbol table.  If the
   //variable is not in the symbol table, an error
   //occurs.  (Note that the values in the symbol
   //table are objects of type Double.)
  String name = readWord();
  Double val = symbolTable.get(name);
  if(val == null)
   throw new ParseError("Unknown variable \""+name + "\"");
  return val.doubleValue();
 }
 else if(ch == '('){
   //The factor is an expression in parentheses.
   //Return the value of the expression.
         TextIO.getAnyChar();  //Read the "("
  double val = expressionValue();
         TextIO.skipBlanks();
  if(TextIO.peek() != ')')
   throw new ParseError("Missing right parenthesis.");
         TextIO.getAnyChar();  //Read the ")"
  return val;
 }
 else if(ch == '\n')
  throw new ParseError("End-of-line encountered in the middle of an
expression.");
 else if(ch == ')')
  throw new ParseError("Extra right parenthesis.");
 else if(ch == '+' || ch == '-' || ch == '*' || ch == '/')
  throw new ParseError("Misplaced operator.");
 else
  throw new ParseError("Unexpected character \""+ch + "\"encountered.");
}


/**
```

```
    * Reads a word from input.  A word is any sequence of
    * letters and digits, starting with a letter.  When
    * this subroutine is called, it should already be
    * known that the next character in the input is
    * a letter.
    */
    private static String readWord() {
     String word = "";  //The word.
     char ch = TextIO.peek();
     while(Character.isLetter(ch) || Character.isDigit(ch)) {
            word += TextIO.getChar(); //Add the character to the word.
            ch = TextIO.peek();
     }
     return word;
    }

}//end class SimpleInterpreter
```

Biography

Benjapol Auprasert is the lone son of Suparat Auprasert and Assoc.Prof Kanya Auprasert. He is based in Bangkok and currently pursuing a Master Degree in Computer Engineering at Chulalongkorn University after receiving the Bachelor Degree from the same faculty. He previously graduated from Saint Gabriel's College and Triamudomsuksa School respectively. The various awards and prizes he has achieved throughout his life include: Finalist for National Software Competition (NSC 2008) for Project - 'Complex Document Study Complementer', Bronze Medal in Thailand National Maths Olympiad Representative Contest 2003, High Distinction Certificate for Australian National Chemistry Quiz 2003, Gold Medal in Thailand National Secondary School Maths Olympiad 2000, 1st Runner-up for Mac Scholarships National Academic Competition 1997, 5 Annual Gold Medals for Top Academic Score of Saint Gabriel's College, and 19 other wins and runner-ups from inter-school academic quiz competitions. He also completed specialist trainings and certification, including SAP HR400 and HR305, CMMI Training, and Sun Certified Java Programmer. His miscellaneous achievement includes Trinity College of London's Grade 5 in Solo Piano, The World Tae Kwan Do Federation's 9th Grade (Yellow II Belt), Marine Life-Saving Training, winner of Orange Photo World Cool Guys Contest, and semifinalist in Mahidol Youth Music Contest 2000.

Benjapol's current areas of interests in research topics are Software Metrics, Complexity Measurement, Cognitive Informatics, and Technological Singularity, which are driven by the belief that understandings of the mechanisms of natural intelligence and cognitive processes of the brain could lead to a big leap forward in information revolution, which will ultimately lift up the advancement of mankind to another level beyond imagination. His researches and scholarships are funded by Software Industry Promotion Agency (SIPA), Ministry of Information and Communication Technology, in collaborate with Chulalongkorn University in the Software Quality Research and Development Project. He has so far published four papers associated with software cognitive complexity measures in four international conference proceedings during the past eight-month period. The success, he believes, is the result of working enthusiastically, working his heart out, having fun, and always keeping the balance between work and life to enable working at the full potential with great attitudes.