EMPIRICAL STUDY OF SOURCE LEVEL DIFFICULTY

Mr. Xiao Liu

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science and Information Technology

Department of Mathematics and Computer Science

Faculty of Science

Chulalongkorn University

Academic Year 2013

Copyright of Chulalongkorn University

บทคัดย่อและแฟ้มข้อมูลฉบับเต็มของวิทยานิพนธ์ตั้งแต่ปีการศึกษา 2554 ที่ให้บริการในคลังปัญญาจุฬาฯ (CUIR)
เป็นแฟ้มข้อมูลของนิสิตเจ้าของวิทยานิพนธ์ ที่ส่งผ่านทางบัณฑิตวิทยาลัย

The abstract and full text of theses from the academic year 2011 in Chulalongkorn University Intellectual Repository (CUIR)

are the thesis authors' files submitted through the University Graduate School.

การศึกษาเชิงประสบการณ์ของความยากระดับต้นฉบับ

นายเซียว หลิว

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2556
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title	EMPIRICAL STUDY OF SOURCE LEVEL DIFFICULTY
By	Mr. Xiao Liu
Field of Study	Computer Science and Information Technology
Thesis Advisor	Associate Professor Peraphon Sophatsathit, Ph.D.

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

....Dean of the Faculty of Science

(Professor Supot Hannongbua, Dr.rer.nat.)

THESIS COMMITTEE

....Chairman

(Professor Chidchanok Lursinsap, Ph.D.)

....Thesis Advisor

(Associate Professor Peraphon Sophatsathit, Ph.D.)

....Examiner

(Assistant Professor Saranya Maneeroj, Ph.D.)

....External Examiner

(Associate Professor Damras Wongsawang, Ph.D.)

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

เชียว หลิว : การศึกษาเชิงประสบการณ์ของความยากระดับต้นฉบับ. (EMPIRICAL STUDY OF SOURCE LEVEL DIFFICULTY) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: รศ. ดร.พีระพนธ์ โสพัศสถิตย์, 55 หน้า.

วัตถุประสงค์ของวิทยานิพนธ์นี้คือวิจัยการวัดความยากระดับต้นฉบับอย่างตรงไปตรงมา โดยใช้กลุ่มตัววัดมาตรฐานที่ใช้กัน ได้แก่ ตัวดำเนินการ ปัจจัยการดำเนินการ พารามิเตอร์ อินพุตและเอาท์พุต การดำเนินการแฟ้มข้อมูล ฟังก์ชันภายนอกหรือไลบารี การประกาศตัวแปร และกราฟการไหล งานวิจัยนี้ใช้ภาษาหกชนิดในการวัดความยากคือ ซี ซีชาร์บ จาวา ไพตอน พีเอชพี และเพิล โดยศึกษาปัญหาสี่ประเภทคือ การเทียบความเหมือน อินพุต/เอาท์พุต การคำนวณ และการเปรียบเทียบ งานวิจัยนี้ใช้โปรแกรมตัวอย่าง 156 โปรแกรมจากอินเทอร์เน็ตเป็นข้อมูล วิธีวิจัยคือแปลงต้นฉบับเป็นกราฟการไหลแล้ววัดด้วยมาตรวัดซอฟต์แวร์มาตรฐาน คือ LOC, CCM, และ HCM ผลการทดลองพบว่า ต้นฉบับที่เขียนด้วยภาษาแปลมีความซับยากมากกว่าภาษาตีความ นอกจากนี้ ไม่มีภาษาหนึ่งภาษาใดที่เหมาะสมกับปัญหาทุกประเภท ผลลัพธ์เป็นการช่วยให้ผู้พัฒนาโปรแกรมสามารถเลือกภาษาที่เหมาะสมกับงานที่จะทำให้เกิดผล

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต	....
		ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก	....
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ		
ปีการศึกษา	2556		

5572602523 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS: SOFTWARE METRICS / SOURCE CODE COMPLEXITY / COMPILED LANGUAGE / INTERPRETED LANGUAGE

XIAO LIU: EMPIRICAL STUDY OF SOURCE LEVEL DIFFICULTY. ADVISOR: ASSOC. PROF. PERAPHON SOPHATSATHIT, Ph.D., 55 pp.

This thesis aims to provide a straightforward measurement for source level difficulty of programs using a set of well-established measurements, namely, operators, operands, parameters, inputs and outputs, file operations, external functions or libraries, variable declarations, and flow graphs. Six popularly used programming languages, namely, C, C#, Java, Python, PHP, and Perl are selected to measure and assess source code difficulty. Four classes of problems are studied, i.e., matching, I/O, computation, and comparison, One hundred and fifty six programs written in the aforementioned programming languages are collected from the Internet to be measured. The approach transforms source code into program flow graph and utilizes traditional software metrics, namely, LOC, CCM, and HCM to measure code complexity. Experiments show that source code written in compiled languages have greater difficulty than those written in interpreted languages. In addition, there is no one language which is suitable for all types of problems. As a consequence, developers can decide on language that is appropriate for the task to be implemented.

Department:	Mathematics and Computer Science	Student's Signature	....
Field of Study:	Computer Science and Information Technology	Advisor's Signature	....
Academic Year:	2013		

ACKNOWLEDGEMENTS

After more than one year of hard work, I am to complete my thesis. As I enjoy my achievement, as I appreciate those teachers, professors and classmates who helped me with it.

During the three years of study in Chulalongkorn University of Thailand, I learnt a lot and recognized my own shortcomings. The teaches here are more friendly and approachable than those in China. At the beginning when I came here as a stranger, I worried that I couldn’t communicate with teachers and classmates and that I couldn’t learn because my English was not very good. It was Professor Peraphon who made me not worry about that and made myself at home. On the first day of school, he kindly introduced me to everyone and asked them to take care of me. As for my study, all my teachers and classmates were very enthusiastic to help me. They often asked where I don't know and explained to me.

In the twinkling of an eye, my study career in Chulalongkorn University comes to an end. In my supervisor, Peraphon's careful guidance and help, I finally finished the expected graduation thesis. Firstly, I appreciate my supervisor Peraphon who has rigorous scholarship and profound professional knowledge. When I set about writing my paper, I didn’t know how to choose my title due to lack of research experience. It was Peraphon who guided me to decide the Innovative research topic and offered me some valuable reference books. In the process of writing, not only did he carefully read my each manuscript and pointed out the problems, but he told me how to revise and supervised me to finish it on time. His enthusiasm and patience strengthened my confidence to complete my thesis. Therefore, I thank him for all that he did for me.

Secondly, I am grateful to professor Childchanok and all the teachers, like Suphakant, Nagul, Saranya, Pattarasinee, and Rajalida who taught me a lot of computer basic professional knowledge and theoretical knowledge which are useful for my paper. I am still grateful to my classmates who helped me learn when I came across the difficulties in my study.

Finally, I would like to thank my parents. They offered me a guarantee for my living. When I encountered difficulties, they comforted and encouraged me to pull through.

CONTENTS

Page

THAI ABSTRACT iv

ENGLISH ABSTRACT v

ACKNOWLEDGEMENTS vi

CONTENTS vii

LIST OF TABLES 1

LIST OF Figures 3

INTRODUCTION 1

1.1 Problem Identification 1

1.2 Objective and Contributions 2

1.3 Scope of Work and Constraints 2

1.4 Outline of the Thesis 2

LITERATURE REVIEW 3

BACKGROUND KNOWLEDGE 6

3.1 Metrics and Software Metrics 6

3.2 Software Complexity Metrics 6

3.2.1 Lines of Code 7

3.2.2 McCabe's Cyclomatic Complexity Metrics 7

3.2.3 Halstead's Complexity Metrics 7

3.2 4 Henry's and Kafura's Metrics 8

METHODOLOGY 9

4.1 Initial Metrics Statistics 9

4.1.1 Operators 9

4.1.2 Operands 10

4.1.3 Parameters 11

4.1.4 Inputs and Outputs 11

4.1.5 File Operations 12

4.1.6 External References 12

Page

4.1.7 Variable Declarations 12
4.2 Flow Graph Transformation 13
4.3 Data Analysis Process 15
4.3.1Range of the Numerical Spread 15
4.3.2 Normalized Data to Standardize the Results 16
4.3.3 Standard Deviation of the Dispersion from the Average 16
4.3.4 Group Average and Standard Deviation 17
4.4 Comparative Evaluation 17
EXPERIMENTAL RESULTS 19
5.1 Experimental Preparation 19
5.2 Experimental Results 21
5.3 Discussion 49
CONCLUSION 51
REFERENCES 52
VITA 55

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

LIST OF TABLES

Table Page

Table 4.1 Operators of different programming languages 10

Table 5.1 The abbreviations and their meanings.. 19

Table 5.2 Numbers of sample programs 21

Table 5.3 Metrics of string matching program in each language 42

Table 5.4 Metrics of array copying program in each language.................................... 42

Table 5.5 Metrics of open and close program in each language 42

Table 5.6 Metrics of read and write program in each language 43

Table 5.7 Metrics of append and update program in each language 43

Table 5.8 Metrics of numerical computation program in each language.................. 44

Table 5.9 Metrics of condition program in each language... 44

Table 5.10 Metrics of branch program in each language 44

Table 5.11 Metrics of loop program in each language 45

Table 5.12 Range of each group... 45

Table 5.13 Normalized average of each group 45

Table 5.14 Normalized average of each language... 46

Table 5.15 Normalized standard deviation of each group 46

Table 5.16 Normalized standard deviation of each language..................................... 46

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Table 5.17 Result summary of language difficulty under specific functional applications...... 49

Table 5.18 Summary of language difficulty under the specific functional application 50

LIST OF Figures

Figures | Page

Figure 4.1 Flow graph of computing N factorial 14

Figure 4.2 Part of flow graphs converted from Java source code 15

Figure 5.1 C sample code 23

Figure 5.2 C# sample code 27

Figure 5.3 Java sample code 30

Figure 5.4 Python sample code 33

Figure 5.5 PHP sample code 37

Figure 5.6 Perl sample code 41

Figure 5.7 Standard deviation of compiled and interpreted groups 47

Figure 5.8 Standard deviation of compiled language: C, C# and Java 47

Figure 5.9 Standard deviation of interpreted language, Python, PHP and Perl 48

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

CHAPTER I

INTRODUCTION

1.1 Problem Identification

As one of the greatest inventions in the 20th century, computers have had a profound impact on human life. With the development of computer science, the application fields of computers have been getting ever wider, from scientific computing to current scientific research and education, aerospace, finance, health care and many other areas. However, along with growing software size, software complexity has also been increasing rapidly. As a result, the thought-provoking software crisis broke out in the 1960s. Some notable reasons for the software crisis is the lack of effective metrics in the development of software engineering and the lack of emphasis on software engineering and software quality. This in turn calls for research expansion to software metrics.

Software metrics attempts to reduce the time and cost spent on testing phase of the software development life cycle (SDLC), which can only be enforced when program coding is done. Effective management of development process requires quantification, measurement, and modeling. Software metrics provides a quantitative basis for the development and validation of software development process model, thus improving productivity and quality. Software quality improvement is a quantitative measure of the quality of source code, which can be achieved through definition of metrics. The values of software metrics can be calculated by analyzing source code, aka program coded.

A number of software metrics widely used in the software industry are still not well understood, although some software complexity measures were proposed over thirty years ago. Nonetheless, software growth is usually considered in terms of the complexity of source code. Various software metrics are used, but failed to compare approaches and results. In addition, it is not possible or easy to evaluate a given source code. As complexity primarily deals with program comprehensibility and software maintainability, the degree to which characteristics that hamper software maintenance are present and determined by software complexity.

Early in the 1970's, software complexity attracted public attentions. The modularization program styles and the object-oriented paradigms were both introduced to lower such complexity. Meanwhile, a number of useful metrics were employed to measure various level of software complexity, yet were inadequate to settle this problem.

As software becomes more and more complex, the cost inevitably increases. Software organizations are trying to find ways to reduce it. Research efforts are spent on finding the relation of software features and the extent of the problem that would lessen the cost burden. One aim of the investigation in software complexity and its measurement is to control the expenditures of software development, operation, and maintenance over its life time. Unfortunately, software complexity is an inherent property that cannot be straightforwardly

identified, described, and measured. Worse yet, it is often disregarded in the development planning process and incorporated as an after-thought artifact. This is particularly apparent during the maintenance phase where considerable amount of efforts are expended to modify the source code. The overwhelming magnitude of complexity poses a challenge for researchers to reckon with.

1.2 Objective and Contributions

This research aims to provide a straightforward measurement of source level complexity of programs using a set of well-established measurements, namely, operators, operands, parameters, inputs and outputs, file operations, external functions or libraries, variable declarations, and flow graphs. Six popularly used programming languages, namely, C, C#, Java, Python, PHP, and Perl are selected and divided into two groups: the complied group and the interpreted group, to effectively measure and assess source code complexity.

1.3 Scope of Work and Constraints

All the source codes were collected from the Internet and of different sizes written in C, C#, Java, Python, PHP, and Perl. The LOC of some programs are between 50 and 100 lines for short programs, and from 200 to 600 lines for longer ones. The Visustin v7 Flow chart generator is adopted to generate the flow graph for each program.

1.4 Outline of the Thesis

The remaining contents are organized into six chapters. Chapter II reviews the literature within the last five years that focuses on software complexity on source level. Chapter III gives an introduction to software metrics and reviews several traditional software complexity metrics, such as LOC, CCM, HCM, and HKM. And LOC, CCM, and HCM are used in part of the research in Chapter IV. A new method for measuring the complexity of source code written in different programming languages is proposed in Chapter IV. Chapter V presents the experimental results of 156 programs of different sizes written in C, C#, Java, Python, PHP, and Perl, and makes a comparison of these results. Finally, Chapter VI draws a conclusion of this research.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

CHAPTER II

LITERATURE REVIEW

Many researches on software complexity have been carried out in recent years. Several metrics have been defined and tested in specific environments. Although remarkable successes have been reported in the initial application and validation of these metrics, subsequent attempts to test or apply the metrics in different situations have yielded different results. One problem could stem from failure to identify a commonly accepted set of software properties. Moreover, there were virtually no theoretical models and metrics to support the measurement. Principally, there are three types of well-practiced software complexity metrics, namely, process metrics, project metrics, and product metrics [1]. Some classical and efficient software complexity metrics introduced in [2] were popularly applied to measure the complexity of software. These metrics were compared in [3] to identify which metric was the most suitable for the state-of-the-practice development. They are McCabe's Cyclomatic complexity (CCM) [4], Halstead's software science [5] complexity metrics (HCM), Kafura's & Henry's fan-in, fan-out (HKM) and Shao and Wangs' cognitive functional size [6].

While the extent of research in this field is still relatively limited, particularly when compared with research on static measurements, the field is growing given the inherent advantages of dynamic metrics. Tahir, et al. [7] systematically investigated the researches on dynamic software metrics to identify issues associated with their selection, design, and implementation. Current measures can be used to compute complexity, but these methods are not sufficient to express complexity variations among programming languages. New methods are being searched for predicting complexity since high degree of complexity in a module is considered inefficient as opposed to low degree of complexity [8]. In addition, the measurement helps estimate other quality attributes such as testability and maintainability [9].

A. Shukla, et al. [10] proposed a metric called control structure based complexity (CSBC). The metric is a slight modification of the McCabe metrics. It computes software complexity by counting the control structures directly from source code, while McCabe metric counts predicate node from the control flow graph. It also solves the problem that predicating node applies only to individual module rather than n modules.

As obfuscation makes the structure of a program harder to analyze and understand, it is believed that obfuscation is the opposite of refactoring. A. Capiluppi, et al. [11] proposed a method for evaluating the complexity of source code by different obfuscation algorithms and different software engineering metrics. It illustrates how the obfuscation algorithms perform with object-oriented attributes that should be low in refactoring by using structural metrics.

S. Sarala and P. Abdul [12] believed that present metrics of source code was unable to predict the actual information flow complexity in the modules. So they proposed the IF-C metrics to evaluate the complexity of source code, focusing on the improved information flow

complexity estimation. The IF-C, which is the abbreviation of intra-modular information flow complexity, measures the information flow complexity using metrics such as sum of fan-in and fan-out(F-(I+O)), code length(C-L) and procedure call(P-C).

J.J. Vinju, et al. [13] introduced the ideas of Control Flow Patterns (CFPs) and Compressed Control Flow Patterns (CCFPs) based on McCabe's Cyclomatic complexity metrics. The method aimed at eliminating the repetitive structure of control flow graphs. They examined the proposed ideas with eight open source Java systems and disproved the belief that there was a clear-cut relationship between McCabe metrics and understandability. In fact, McCabe metrics should be reconsidered a metric for code understandability.

As most present software metrics measure software complexity based on a per file or a per function, Oliver Hummel and Stefan Burger [14] proposed the hm-Index metrics to decrease such dependencies based on the h-Index in bibliometrics, which already had a great impact on its own field.

One of the most important features of object-oriented system is the inheritance. Based on this feature, S. Misra, et al. [15] proposed a cognitive metric to evaluate the complexity of object-oriented code. It considered internal structure of methods to calculate the complexity of source code at method level. They validated the metrics both theoretically and empirically, and finally proved the robustness of the measure.

L. Prasad, et al. [16] made an experimental analysis of different software metrics. They defined a new set of operational measures for the conceptual coupling of classes to investigate the relationships between existing object-oriented metrics (such as coupling and cohesion) and procedure-oriented metrics (such as Lines of Code, McCabe's Cyclomatic complexity, and Knot metrics). They proved that these metrics captured new dimensions in coupling measurement compared to existing structural metrics.

As it is known that over 75% of project costs come from maintenance phase where most activities involve source code modifications. Such the high cost of maintenance instigates the necessity to produce high quality software. R.G. Kula, et al. [17] proposed a set of metrics based on programs to investigate a quantitative approach on evaluation of the maintenance effort. They compared the metrics with the basic metrics based on source code and found that program slicing metrics had the strongest correlation with maintenance effort, while the basic metrics had a weak correlation. P. Singh, et al. [18] proposed a software quality assurance tool to measure different metrics for C#, as most published papers in object-oriented languages concentrated on C++ and Java. The tool generates abstract syntax trees of source code at class and method levels. A. Abusasd and I.M. Alsmadi [19] evaluated the relation and correlation between two software quality tools: Source Code Analysis (SCA) and Software Metrics. They used tools to fix warnings detected by SCA tools. The tools measured before and after recommended modifications of SCA. It showed that some specific metrics relating to structure, complexity, and maintainability could be significantly impacted by SCA modifications.

Y. Sasaki, et al. [20] believed that metrics did not always represent characteristics of software systems. For example, if a structure was a mere repetition of simple operations,

McCabe's Cyclomatic complexity became large but the source code would be easy to understand. Based on this opinion, they proposed preprocessing for metrics measurement to simplify repeated structures in source code. It was proved that the approach was more efficient to find low-understandability modules by metrics measurement with preprocessing than without it.

CHAPTER III

BACKGROUND KNOWLEDGE

3.1 Metrics and Software Metrics

Metrics is the process of describing the numbers or symbols which are distributed to the attributes of entities according to the clearly defined rules, or the process of mapping the empirical relationship system to the digital relationship system according to the mapping rules [21].In this case, human can understand the features of the entities and the relationship between them easily.

Software metrics is a measure of software process and its product, or its applications. The measurement, usually using numerical ratings, quantifies some characteristics or attributes of the software [22]. Typical measurements include quality of source code, the development process, and the accomplished applications [23].

As software development progresses, software metrics has been widely studied and has become a very important index of software measurement. Many software developers measure software quality in many aspects such as design quality, requirements availability and consistency, and testable source code. Many project managers measure attributes of processes and products to decide if the software is ready for delivery and the budget is exceeded. The informed customers measure aspects of the final products to determine if it meets the requirements having sufficient quality.

Fenton and Pfleeger [21] believed that metrics should be established on the basis of measurement theory, and be tested whether a given measurement criterion suited the particular environment. They proposed four activities needed in the process of software measurement:

1) confirm software attributes concerned
2) establish empirical relationship system for software attributes
3) define formalized metrics and map empirical relationship system to digital relationship system
4) evaluate the measurement criteria

3.2 Software Complexity Metrics

Complexity is generally defined as the degree to which a system or component is understood. Complexity metrics are used to predict critical information about reliability and maintainability of software systems [24]. Software complexity metrics is an essential part of software metrics, which focuses on the quality of source code.

Software complexity is a term that includes properties of software which have a great impact on internal interactions. The complexity indicates the interactions among the entities of software. The number of interactions would increase exponentially with the number of entities, and the value would increase to a point where it is impossible to understand the program.

Software complexity metrics also takes an important part in predicting the failure rate of software. As complexity is the leading cause of software errors, the nature of software reliability is actually a complexity issue. When the complexity exceeds a certain limit, software defects and errors will rise sharply. Thus, software complexity metrics and control is an important issue to solve in software reliability engineering.

Traditional software complexity metrics have been designed and applied for measuring the software complexity of structured systems since 1976. Of these metrics, it is found that LOC, CCM, HCM and HKM are the most commonly used ones.

3.2.1 Lines of Code

Lines of code (LOC) are widely used software metrics for measuring the size of a program by counting the number of lines in the program source code. It is used to estimate the effort that will be spent on the development, productivity, and maintainability of a program.

LOC is based on the size of methods and gives measure of physical lines, statements, and comments. High value of this metric shows more complexity [2,10].

3.2.2 McCabe's Cyclomatic Complexity Metrics

McCabe's Cyclomatic Complexity Metrics (CCM) was proposed by McCabe in 1976. It uses the control flow graph (CFG) to compute the complexity of a program. The nodes in CFG represent command statements of the program and the edges connecting nodes represent the data flow of command statement in the form of order.

It is defined as

$$V(G) = e - n + 2p \tag{3.1}$$

where e is the number of edges, n is the number of nodes, and p is the number of connected node or region. This metric gives the measure of independent algorithmic test paths. More independent paths mean more testing effort.

3.2.3 Halstead's Complexity Metrics

Halstead's Complexity Metrics (HCM) attempts to estimate the programming effort [2,5,11] which is evaluated statically from the source code. HCM is used to evaluate the measurable properties of software and the relations between them. It measures the complexity by summarizing the number of operators and operands contained in a program. The measurable and countable properties are:

n1 = number of unique or distinct operators appearing in that implementation,

n2 = number of unique or distinct operands appearing in that implementation,

N1 = total usage of all of the operators appearing in that implementation, and

N2 = total usage of all of the operands appearing in that implementation.

The difficulty, D, of the program is defined as

$$D = (n1 * N2)/(2 * n2) \tag{3.2}$$

3.2 4 Henry's and Kafura's Metrics

Henry and Kafura Metrics (HKM) also proposed the complexity metrics based on the number of local information flows entering (fan-in) and exiting (fan-out) in each procedure. This metrics is given by

$$Complexity = (Proc.\ Length) * (fan\text{-}in * fan\text{-}out)^2 \tag{3.3}$$

where *length* is any measure of length such as LOC or alternatively CCM is sometimes substituted.

All these four metrics can be applied for measuring the component complexity at method level based on the availability of source code which would be available in the case of white box components.

CHAPTER IV

METHODOLOGY

This study aims to measure the complexity of source code written in different programming languages. A method using metrics that focuses on operators, function parameters, file operations, and flowcharts is proposed, which is divided into four stages, namely, initial metrics statistics, flow graph transformation, data analysis process, and comparative evaluation.

The source programs employed are collected in modern popular software application languages and grouped based on their operational characteristics. There are two kinds of programming languages: compiled programming languages, such as C, C#, Java, and interpreted programming languages, such as Python, PHP, and Perl. A compiled programming language is a language whose implementation utilizes compilers instead of interpreters. The compilers translate source code into machine code, while interpreters execute instructions directly without pre-runtime translation.

4.1 Initial Metrics Statistics

For source level complexity evaluation of software, seven kinds of essential programming tokens are selected to be evaluated, namely, operators, operands, parameters, inputs and outputs, file operations, external function references, and variable declarations.

This step is to count the number of the following tokens:

1) operators such as +, -, *, /, %, ->, (), +=, -=, ++, --;
2) operands in executable statements;
3) parameters in and out of each module or function;
4) inputs and outputs of the program;
5) file operations including open, close, read, and write;
6) external references including library functions, external source files, and external user-defined functions; and
7) variable declarations in the program.

4.1.1 Operators

An operator is a program element that takes one or more expressions and produces another value. There are different sets of operators in programming languages, such as simple arithmetic operations +, -, *, /, logical operations **AND** or &&, **OR** or||,**NOT** or ~, file access to an object or a record, and scope resolution operation **::**.

The operators of different programming languages are not exactly the same. For example, () in C++ is a non-alphanumeric operator, but is not an operator in Perl. Programs of different sizes involve different numbers of operators. Generally speaking, the number of operators reflects the scale of the source code to a certain degree.

In this research, the complexity of source code in six programming languages are studied, namely, C, C#, Java, PHP, Perl, and Python. Their operators have certain differences. Table 4.1 shows the operators of these programming languages.

Table 4.1 Operators of different programming languages

<table>
<tr><th>Programming Languages</th><th>Non-alphanumeric operator symbols</th><th>Alphanumeric operator symbols</th></tr>
<tr><td>C</td><td>() [] -> . ! ~ ++ -- + - * & / % <<>><= == != ^ | && || ?: = += -= *= /= %= &= ^= |= << = >>=</td><td>sizeof</td></tr>
<tr><td>C#</td><td>() [] . :: + - * / % & | ^ ! ~ = <> ?: ++ -- && || <<>> == != <= >= += -= *= /= %= &= |= ^= <<= >>= -> ?? =></td><td>new, typeof, checked, unchecked, delegate, sizeof, await</td></tr>
<tr><td>Java</td><td>() [] -> . ! ~ ++ -- + - * & / % <<>><= == != ^ | && || ?: = += -= *= /= %= &= ^= |= << = >>=</td><td>new
instanceof</td></tr>
<tr><td>Python</td><td>+ - * / % ** // == != <>><>= <= = += -= *= /= %= **= //= & | ^ ~ <<>></td><td>and, or, not, in, not in, is, is not</td></tr>
<tr><td>PHP</td><td>+ - * / % = += -= *= /= %= . .= ++ -- == === != <> !== ><>= <= && || !</td><td>and, or, xor</td></tr>
<tr><td>Perl</td><td>-> ++ -- ** ! ~ \ + - . =~ !~ * / % <><= >= == != <=> ~~ & | ^ && || ‘</td><td>print, sort, chmod, chdir, rand, and, or, not, xor, lt, gt, le, ge, eq, ne, cmp, x</td></tr>
</table>

It can be seen that the simple arithmetic operators such as +, **-**, *, /exist in all the six programming languages, while the operators such as **()**, **[]** exist only in C, C#, and Java, but does not exist in PHP, Python, and Perl. Note that almost every language has several special operators that other languages do not have. For example, the checked and unchecked operators exist only in C#, the operators **in**, **not in**, **is**, and **is not** exist only in Python, and the operator === exists only in PHP.

4.1.2 Operands

As operators have great impact on the complexity of source code, the operands are also an important factor of source code complexity. An operand is the object of a mathematical operation in mathematics and the part of a computer instruction in computer science.

In all programming languages, expressions are composed of two components, namely, operators and operands. For example, in the expression $x+y$, x and y are operands and +is an operator. In other words, the number of operands is directly related to the number of operators.

Operators fall into three types: unary operators, binary operators, and ternary operators. A unary operator takes only one operand or a single input. The operators, such as ++, **--**, **!**,**&**, and ~ are all unary operators. If there is a unary operator in a program, there is one operand. Similarly, if there is a binary operator, such as +, **-**, *, /, or **%**, there are two operands. There is only one ternary operator **?:**, which takes three values and adds 3 to the number of operands each time it appears in the source code of a program.

4.1.3 Parameters

A parameter is a variable used in a subprogram to refer to one of the arguments and provided as an input to the subprogram. Every time the subprogram is called, its arguments will point at the corresponding parameters. With that said, the time that a parameter is called can reflect the time that the subprogram is called in a program. A parameter refers to the variable which is found in the definition of a function, but an argument is usually used to refer to the actual value passed.

In general, a parameter is used to customize a program. Each module or function can have its own parameters. The number of parameters differs in different functions. The amount of parameters will reflect the number of functions or modules in a program.

Most modern programming languages allow multiple parameters in different functions. Although the syntax of the declaration of a function may be different among the programming languages, a typical function with two parameters may be presented by

```
function add_XY (x, y)
  {

    ...

  }
```

If this function is called, there will be two variables passed into the function as input parameters. Typically, a single function can be called multiple times in a program. If the above function appears in a program, the number of parameters increases by 2.

4.1.4 Inputs and Outputs

Inputs and outputs are the communication between source code and the outside world. Inputs mean the data needed by the program to complete its execution, while outputs are the data that a program conveys to the user.

There are many forms that may be taken by the program to implement input and output operations. For example, some programs use graphical components to receiver and return the information typed by user, some programs are controlled by clicking the mouse or keyboard, and some programs get the information from the internet or devices like scanners. These programs all contain input operations. There are many devices used for output operations such as monitors and printers.

In any programming language, input means to put the data into a program, which can be achieved by reading data from files or command line, while output means to display the result on a screen, printer, or in a file. The file operations will be discussed in the next step.

Every programming language provides a set of built-in functions to read data as inputs, or to output required data. But different programming languages have different input and output functions. For example, in Java programming language, input uses a **ScannerObject** connected to **System.in** and output uses a **PrintStream** object connected to **System.out**. In C

programming language, function **scanf()** is used to obtain data from the console and **printf()** to output data to output devices.

4.1.5 File Operations

File operations are simply the action that one can do to a file. There are six basic types of file operations: create, delete, open, close, read, and write. In most cases, the programs that are executed on computers handle these operations. It also means that a program with a certain scale will involve file operations inevitably, and the number of file operations in a program can also represent the scale of the program to a certain degree.

File operations are similar to the input and output operations, besides the fact that the object is replaced by a file. It should be noticed that the functions or the classes about file operations differ in different programming languages. Java programming language uses a package **java.io.File** to store information about a file on a computer drive. C programming language uses a standard library **stdio.h** for file operations which provides many functions, such as **fopen(), fread()** and **fwrite()** to open, read, and write a file. Similar to C programming language, PHP programming language also uses the function **fopen()** to open a file, **fread()** to read a file, and **fwrite()** to write data to a file.

In a word, all the operations on files should be counted as the number of file operations.

4.1.6 External References

The external references mean the reference of an external function defined outside the program, including library functions, external source files, and external user-defined functions. For a large program, the elements such as inputs and outputs, file operations, and external function references will be used inevitably. As a result, the times they are used in a program can also indicate the complexity of source code.

Library functions are the collection of the implementations of behavior, which are invoked by multiple independent programs. When a program invokes a library function, it will obtain the result evaluated by the functions without the need to implement that operation by itself. So the library functions can simplify the internal structure of a program and improve reusability and portability.

If a program needs a function or a class that is defined in the external source file or the external user-defined functions, the programmer will reference this file or function in the program. For example, Java programming language usually uses the keyword **import** to reference the needed package at the head of the source code, while PHP programming language uses the **include()** function to reference the external source file.

4.1.7 Variable Declarations

In programming languages, a declaration of a variable specifies the identifier, type, and other aspects of language elements. In many strongly typed languages, such as C, the declaration is of great importance to announce the existence of the element to the compiler, while it is not very important to the weakly–typed languages, such as PHP. The scope of a variable should also be noticed because there may be several variables in a program with the same name.

By this step, the number of seven essential tokens of various programs in different programming languages is obtained, which is to be used in the following steps to evaluate the complexity of source code.

4.2 Flow Graph Transformation

This step transforms each program into a flow graph and counts the number of nodes and edges of the flow graph. A flow graph is a diagram that represents an algorithm or a process, turning source code into boxes of various kinds and arrows.

In this study, flow graph transformation is carried out using Visustin v7 Flow Chart Generator. It is an automated program flowcharting utility used to create flowcharts from source code. Visustin supports 43 languages, including C, C#, Java, PHP, Python, and Perl.

Figure 4.1 shows the flow graph of the program for computing the factorial of *N*. A flow graph may have the symbols as follows:

1) Start and end symbols: represented as circles, ovals, or rounded fillet rectangles with the words “Start” and “End”.
2) Arrows: denoting the data passing from one symbol to another symbol that the arrow points to.
3) Generic processing steps: represented as rectangles, such as “M=1” or “F=F*M” in Figure 4.1.
4) Subroutines: represented as rectangles with double vertical edges which are used to indicate complex processes that may be shown in a separate flow graph.
5) Input/Output: represented as a parallelogram, such as “Print F” in Figure 4.1.
6) Conditional or decision: represented as a diamond to show where a decision is necessary, usually a Yes/No question or True/False test. The “Is M=N?” in the following figure is a conditional.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

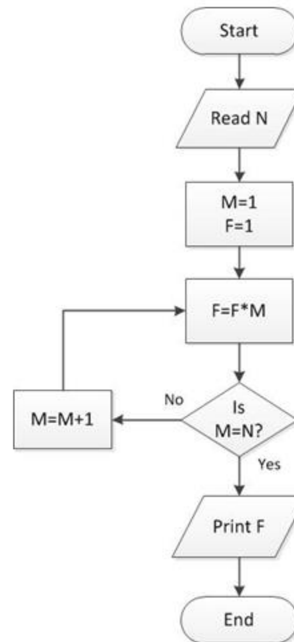

Figure 4.1Flow graph of computing N factorial

Almost all flow graphs contain the above symbols, but some flow graphs may include other symbols, such as junction symbols, labeled connectors, and concurrency symbols.

Figure 4.2 shows parts of the flow graph converted from Java source code. It can be seen from the graph that the number of nodes and edges are 31 and 33, respectively. Under common conditions, the number of nodes and edges of a large program is generally large. In this case, the source code written in different programming languages is transformed into the corresponding flow graphs with the number of nodes and edges of the flow graphs as a relevant factor to measure the complexity of source code in the corresponding programming language.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

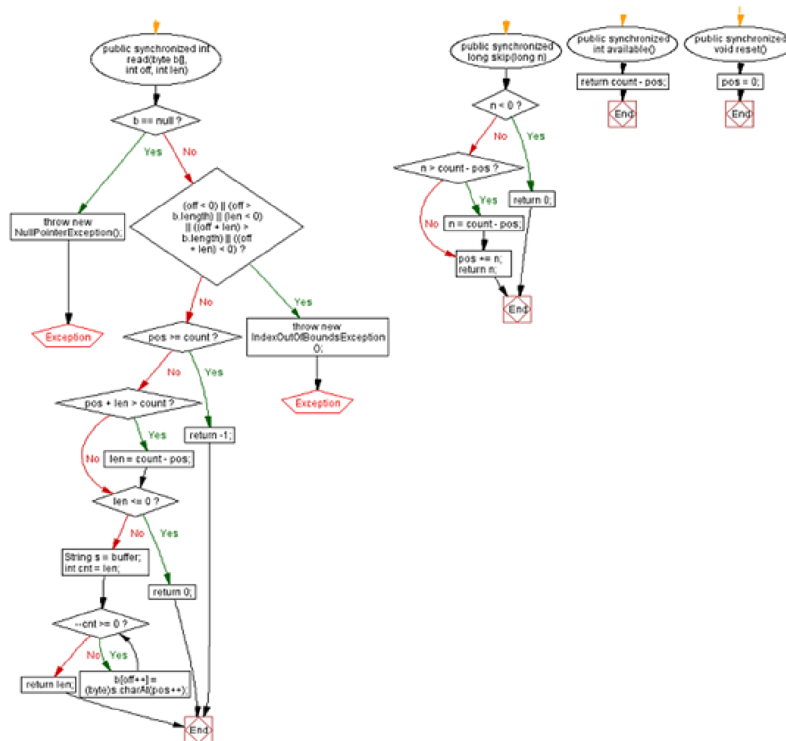

Figure 4.2 Part of flow graphs converted from Java source code

4.3 Data Analysis Process

Once the data from the above two steps are obtained, a quantitative data analysis is needed to make sense of what the data means. A quantitative data analysis is the use of statistical technologies to describe and analyze the variation of quantitative data.

Summary statistics describe particular features of a distribution and facilitate comparison among distributions. Based on the results of the first two steps, the following statistics are computed: range of the numerical spread, normalized data to standardize the results, standard deviation of the dispersion from the average, and group average and standard deviation.

4.3.1Range of the Numerical Spread

The first step of data analysis process is to discover the range of the numerical spread. The range is the simplest method for measuring the variation. It is the total spread in the population, which can be obtained from the difference between the maximum and minimum values.

Firstly, the smallest and largest numbers in the data set are identified and represented by variables Min and Max respectively. Then, the range of the numerical spread can be evaluated by

$$Range = Max - Min + 1 \qquad (4.1)$$

It is necessary to measure the range of the numerical spread by using the first two steps in order to identify the whole range of the values.

CHULALONGKORN UNIVERSITY

4.3.2 Normalized Data to Standardize the Results

The range is greatly affected by exceptionally small or large values in a population. In order to exclude the effect of the exceptional values, one should normalize the data and standardize the results.

Data normalization is the process of mapping the original data range into another scale. It is often used in the process of comparing and evaluating some metrics, removing the limit of data units, converting them to dimensionless values, and making it possible to compare and weight the metrics with different units or magnitudes. In order to get the normalized data, one needs a data set minimum and maximum, a normalized scale minimum and maximum, a number in the data set, and a normalized value.

Firstly, identify the largest number in the data set obtained in the above step, and represent it with the variable Max. Secondly, identify the smallest and largest numbers in the normalized scale and represent them by the variables lowercase ***a*** and lowercase ***b*** respectively, with a=0 and b=1inthis experiment. Thirdly, calculate the normalized value of any number X in the data set using

$$X^* = a + \frac{X*(b-a)}{Max} \tag{4.2}$$

The equation can be simplified as

$$X^* = \frac{X}{Max} \tag{4.3}$$

All numbers except CCM and HCM should be normalized by the above equation. Several examples of the calculation of normalized value in different programming languages are illustrated below.

1) LOC of C
   LOC of C = 46/59 = 0.78

Here 46 represents the value of LOC in C programming language, and 59 represents the largest value of LOC in the six programming languages. In this case, 59 is obtained from the LOC of Java programming language.

2) OR of Java (number of operators)
   OR of Java = 16/16 = 1.00

The maximum value of OR in the programming languages is 16, so the normalized result of OR in Java equals 1.00.

3) EL of C# (number of external functions, libraries, and files)
   EL of C# = 1/5 = 0.20

Similar to the evaluation of LOC of C programming language, the maximum of EL exists in Perl programming language, which is 5, and the value of EL in C# programming language is 1. After calculation, the normalized value equals 0.20.

4.3.3 Standard Deviation of the Dispersion from the Average

The standard deviation is a measure of the dispersion of data set from its average. The more apart the data spread, the greater the standard deviation is. A small standard deviation

indicates that the points of data are more inclined to be close to the average, while a large deviation indicates that the points of data spread over a large range of values.

For a finite set of data, the standard deviation is evaluated by taking the square root of the average of the squared differences of the values from their average value. Firstly, calculate the average of the data set, which is represented by the variable x_i. Secondly, calculate the difference of each data point from the average and square the result of each. Then, calculate the average of these values and take the square root. And the above steps can be concluded by

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{x})^2}{n}}$$

(4.4)

Where $\bar{x}$ represents the average, n is the number of the cases, $\Sigma$is the sum of all the cases, and $x_i$is the value of the variable x with a subscript i

The mathematical properties of standard deviation make it the preferred measure of variability in many cases, especially when a variable is normally distributed. So the standard deviation helps find how great the variation is and when the range will fall in any cases.

4.3.4 Group Average and Standard Deviation

After the above steps, the normalized average and normalized standard deviation for each program are obtained. In this section, the data of each group or each language should be grouped together to be analyzed in Chapter V.

The goal of this research on the complexity of source code in different programming languages is to find the effect of different programming languages on the complexity of source code. The six programming languages are divided first into two groups: the compiled language group and the interpreted language group. Then the average and standard deviation of each group are calculated, which are used to analyze the different effects of the complied languages and the interpreted languages on the complexity of source code. The normalized standard deviation of each group and each programming language should also be calculated, in order to analyze the concrete impacts of programming languages in each group.

For example, the average of OR is 0.42 in C, 0.35 in C #, and 0.33 in Java. Then the average of OR in the complied group equals 0.37, which is evaluated by

OR of Complied language = (0.42+0.35+0.33)/3 = 0.37

4.4 Comparative Evaluation

Until now, all the data needed have been obtained to evaluate the complexity of source code written in different programming languages. In this step, the data are analyzed in graphic form to compare the effect of different programming languages on the complexity of source code.

There are many graphing options to present data, such as statistical maps, column graphs, pie charts, frequency polygons, and histograms. Column graphs are chosen for comparative evaluation. The column graph is an effective demonstration of the differences in percentages

or frequencies of ordinal variables visually. Compared with other methods, the column graph is very useful in comparing the differences among variables in different groups.

In this step, two kinds of column graphs are employed: one is about the standard deviation of compiled groups and interpreted groups; the other is about the standard deviation of programming languages, such as C, C# and Java in compiled group, and Python, PHP, and Perl in interpreted group.

The graphs will give a visual comparison of the measurement results. Any discernible proportion of the programming languages or the groups will reflect the level of variations in the complexity of source code.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

CHAPTER V

EXPERIMENTAL RESULTS

5.1 Experimental Preparation

The experiment on the evaluation of source code complexity by the proposed method is elaborated. The following table shows all the abbreviations used to denote all metrics being collected, namely, LOC, OR, OD, PR, IO, FE, EL, VE, FG, CCM, and HCM.

Table 5.1 The abbreviations and their meanings

Abbreviations	Meanings
LOC	lines of code
OR	the number of operators
OD	the number of operands
PR	the number of formal arguments
IO	the number of inputs and outputs invoked by each function
FE	the number of file operations
EL	the number of external functions, libraries, and files linked
VE	the number of variable declarations
FG	the sum of nodes and edges derived from program flowchart
CCM	McCabe's Cyclomatic Complexity Metrics
HCM	Halstead Complexity Metrics

The value of CCM and HCM can be obtained from their definitions in Chapter III. McCabe's CCM is based on the program flow graph and is defined as **V(G)=e-n+2**, where *e* and *n* represents the number of edges and number of nodes respectively. The complexity of a program measured by HCM is defined as **D=(n1*N2)/(2*n2)**, where *n1*is the number of unique or distinct operators appearing in the code, *n2* is the number of unique or distinct operands appearing in the same code, and *N2* is the total usage of all the operands appearing in that code.

This thesis collects sample programs of different sizes written in C, C#, Java, Python, PHP, and Perl from the Internet. The programs are source code of the above programming languages having various lengths, ranging from 50 to 100 for short ones, and from 200 to 500 for long ones. The dataset includes 156 programs were selected based on predetermined functionalities in order to compare the effectiveness of each language under the same conditions. There are four basic types of functions to be exercised, namely, matching, I/O, computations, and comparison. Each type contains different sub-problems. For example, matching includes string matching and array copying; I/O includes file open and close, read and write, and append and create; computations include numerical computations; and comparison includes condition, branch, and loop operations. The numbers of sample programs for each sub-problem collected is summarized in Table 5.2.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Table 5.2 Numbers of sample programs

		C	C#	Java	Python	PHP	Perl
Matching	String matching	3	3	3	3	3	3
	array copying	3	3	3	3	3	3
	swapping	3	3	3	3	3	3
Input or Output	open,close	2	2	2	2	2	2
	read,write	2	2	2	2	2	2
	appeal,creat, rewind,update	2	2	2	2	2	2
Compute	numerical computation	2	2	2	2	2	2
Comparison	condition comparision	3	3	3	3	3	3
	branches comparision	3	3	3	3	3	3
	loop	3	3	3	3	3	3

Visustin v7 Flow Chart Generator is used to generate flow graph of each program. The first three languages C, C#, and Java are grouped as compiled programming languages, whereas the remaining three Python, PHP, and Perl as interpreted programming languages.

CHULALONGKORN UNIVERSITY

5.2 Experimental Results

The programs are collected specifically according to the designated functionalities to be studied, namely, matching, I/O, computations, and comparison. Each program owns at least one specific function, such as array copying, numerical computations, or file operations.

The following sample source codes are written in C, C#, Java, Python, PHP, and Perl. Each program is collected according to **string matching** sub-problem.

Figure 5.1 shows a C program for string matching whose OR is 27, OD is 48, PR is 8, I/O and EL are 1, FE is 0, and VE is 12.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


Int get_nextval(char *pattern, int next[])

{

 //get the next value of the pattern

            Int i = 0, j = -1;

   next[0] = -1;

int patlen = strlen(pattern);

   while ( i<patlen - 1){

      if ( j == -1 || pattern[i] == pattern[j]){

                                   ++i;

         ++j;

         if (pattern[i] != pattern[j])

            next[i] = j;

         else

            next[i] = next[j];

      }

      else

         j = next[j];

      }

   return(0);

}

Int kmpindex(char *target, char *pattern, int pos)

{

Int tari = pos, pati = 0;

Int tarlen = strlen(target), patlen = strlen(pattern);

int *next = (int *)malloc(patlen * sizeof(int));

get_nextval(pattern, next);
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
    while ( tari<tarlen&&pati<patlen ){
        if (pati == -1 ||target[tari] == pattern[pati]){
            ++tari;
            ++pati;
            }else{
pati = next[pati];
            }
    }
             if(next != NULL) free(next);
             next = NULL;
             if (pati == patlen)
                         return tari - pati;
             else
                         return -1;
}
int main()
{
    char target[50], pattern[50];
printf("imput the target:\n" );
scanf("%s",target);
printf("imput the pattern:\n" );
scanf("%s",pattern);
int ans = kmpindex(target,pattern,0);
    if (ans == -1)
printf("error\n");
    else
printf("index:%d\n",ans);
    return 0;
}
```

Figure 5.1 C sample code

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Figure 5.2 shows the same functionality program written in C# whose OR is 82, OD is 137, PR is 3, I/O is 17, FE is 0, EL is 22, and VE is 14.

```
static void Main(string[] args)
{
    string str = "abababababab";
    string str1 = "abc3241abc8979";
    string str2 = "abababababac";

Console.WriteLine("The whole string is as following:");
Console.WriteLine(str);
Console.WriteLine("After running, the result of the max substring is:");
Int len = str.Length;
int[] sd = new int[8];
ConWrite(str1);
getNext(str1);
ConWrite(str);
getNext(str);

ConWrite(str2);
getNext(str2);
Console.ReadLine();
}

/// <summary>
/// Get the next value of pattern string
/// </summary>
private static void getNext(string s)
{
Int len = s.Length;
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
int[] next = new int[len];
     string temp = "", temp2 = "";
     //when i=0 , next=-1 ;
     next[0] = -1;
int index = 0;
     for (int j = 1; j <len; j++)
     {
        for (int k = 1; k > 0 && k < j; k++)
        {                  /
            temp = s.Substring(0, k);
            temp2 = s.Substring(j - k, k);
            if (temp2 != "" && temp != "")
            {
                if (temp2 == temp)
                {
                     next[j] = k;
                }
            }
       }
       if (next[j] <= 0)
       {
            next[j] = 0;
        }
   }
Int tempLength = 0,tempIndex=0;
Console.Write("the next value is  : ");
    for (int i = 0; i<len; i++)
    {
Console.Write(" "+next[i] + " ,");
        if (next[i] > 0)
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
        {
            if (next[i]>tempLength)
            {
tempIndex = i;
tempLength = next[i];
             }
        }
  }
Console.WriteLine();

  if (s.Substring(0, len - 1).Contains(s.Substring(len - tempLength)))
  {
tempLength += 1;
      index = tempIndex>tempLength ? tempIndex - tempLength + 1 : tempLength - tempIndex
+ 1;
   }else{
       index = tempIndex>tempLength ? tempIndex - tempLength : tempLength - tempIndex;
    }
Console.WriteLine();
Console.WriteLine("index of the longest string : "+index);
Console.WriteLine("length of the longest string : " + tempLength);
}

private static void ConWrite(string str)
{
    Console.WriteLine("/****************************************************/");
foreach (char ch in str)
    {
Console.Write(" "+ch + " ,");
    }
```

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

```
}
```

Figure 5.2 C# sample code

Figure 5.3 shows the same functionality program written in Java whose OR is 79, OD is 97, PR is 3, I/O is 3, FE is 0, EL is 9, and VE is 16.

```
package com.lmning.string;


/**
 * KMP Algorithm
 * @author lmning
 *Mar 22, 2009 12:01:42 PM
 */
public class KMP {
	public static int index_KMP(String s,Stringt,int pos){
		int lens = s.length();
		int lent = t.length();
		char[] chs = s.toCharArray();
		char[] cht = t.toCharArray();
		int i=pos,j=0,index=-1;
		int[] next=getNext(t);
		while(i<lens){
			if(j==0||chs[i]==cht[j]){
				i++;
				j++;
			}
			else j=next[j];

			if(j==lent){
				index = i-j;
				System.out.println("index:"+index);
				j=0;
			}
		}
		return index;
	}
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
/**
 * get the next value of s and save it into array
 * @param s
 * @return
 */
public static int[] getNext(String s){
        int len = s.length();
        int[] next = new int[len];
        char[] ch = s.toCharArray();
        int i=1;
        int k=0;
        while(i<len){
                if(i==0||i==1){
                        next[i]=0;i++;continue;
                }
                k=next[i-1];
                 while(true){
                        if(ch[i-1]==ch[k]){
                                k++;
                                        next[i]=k;
                                        break;
                 } else if(k==0){
                                next[i]=0;break;
                        }
                k=next[k];
                }
                i++;
        }
        return next;
}
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
        public static void main(String[] args) {
                String s = "abcacabcabcacccccabcacvzvcz";
                String t = "abcac";
                int[] next = KMP.getNext(t);
                System.out.print("next[]=");
                for(int i:next)
                System.out.print(i+"  ");
                KMP.index_KMP(s, t, 0);
        }
}
```

Figure 5.3 Java sample code

Figure 5.4 shows the same functionality program written in Python whose OR is 30, OD is 55, PR is 3, I/O is 22, FE is 0, EL is 25, and VE is 12.

```
def print_board():
    print
    print
    print '*'*50
    for i in range(0,3):
        for j in range(0,3):
            if map[i][j] != " ":
                print map[i][j],
            else:
                print "%d" %(i*3+j+1),
            if j != 2:
                print "|",
        print ""
```

```
    print '*'*50


def check_done():
    for i in range(0,3):
        if map[i][0] == map[i][1] == map[i][2] != " " \
        or map[0][i] == map[1][i] == map[2][i] != " ":
            print turn, "won!!!"
            return True


    if map[0][0] == map[1][1] == map[2][2] != " " \
    or map[0][2] == map[1][1] == map[2][0] != " ":
        print turn, "won!!!"
        return True


    if " " not in map[0] and " " not in map[1] and " " not in map[2]:
        print "Draw"
        return True
    return False


def help():
    print "*" *76
    print "*" *76
    print "**  Welcome to the tic-tac world,it's modified by me(hunter xue).        **"
    print "**  the origin source is here:                                        **"
    print "**  \"http://jiabin.tk/2013/07/22/tic-tac-toe-in-python/\"                **"
    print "**  Hope you enjoy this simple game                                     **"
    print "*" *76
    print "*" *76


turn = "X"
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
map = [[" "," "," "],
      [" "," "," "],
      [" "," "," "]]
done = False

help()
while done != True:
print_board()
   print
print turn, "'s turn=>",

   moved = False
   while moved != True:
      try:
pos = input("Select: ")
         if pos<=9 and pos>=1:
            Y = pos/3
            X = pos%3
            if X != 0:
               X -=1
            else:
                X = 2
                Y -=1

            if map[Y][X] == " ":
               map[Y][X] = turn
               moved = True
               done = check_done()

               if done == False:
```

```
                if turn == "X":
                    turn = "O"
                else:
                    turn = "X"
            else:
                print "The position is occupied by ",turn


        else:
            print "You need to add a numeric value between 1-9"


    except:
        print "You need to add a numeric value between 1-9"
```

Figure 5.4 Python sample code

Figure 5.5 shows the same functionality program written in PHP whose OR is 56, OD is 107, PR is 12, I/O and FE are both 0, EL is 23, and VE is 21.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
<?php
class ZUser
{
	const SECRET_KEY = '@4!@#$%@';

	static public function GenPassword($p) {
		return md5($p . self::SECRET_KEY);
	}

	static public function Create($user_row, $uc=true) {
		if (function_exists('zuitu_uc_register') && $uc) {
			$pp = $user_row['password'];
			$em = $user_row['email'];
			$un = $user_row['username'];
			$ret = zuitu_uc_register($em, $un, $pp);
			if (!$ret) return false;
		}

		$user_row['password'] = self::GenPassword($user_row['password']);
		$user_row['create_time'] = $user_row['login_time'] = time();
		$user_row['ip'] = Utility::GetRemoteIp();
		$user_row['secret'] = md5(rand(1000000,9999999).time().$user_row['email']);
		$user_row['id'] = DB::Insert('user', $user_row);
		$_rid = abs(intval(cookieget('_rid')));
		if ($_rid) {
			$r_user = Table::Fetch('user', $_rid);
			if ( $r_user ) ZInvite::Create($r_user, $user_row);
		}
		if ( $user_row['id'] == 1 ) {
			Table::UpdateCache('user', $user_row['id'], array(
```

```
                                        'manager'=>'Y',
                                        'secret' => '',
                                        ));
        }
        return $user_row['id'];
    }

    static public function GetUser($user_id) {
        if (!$user_id) return array();
        return DB::GetTableRow('user', array('id' => $user_id));
    }

    static public function GetLoginCookie($cname='ru') {
        $cv = cookieget($cname);
        if ($cv) {
            $zone = base64_decode($cv);
            $p = explode('@', $zone, 2);
            return DB::GetTableRow('user', array(
                'id' => $p[0],
                'password' => $p[1],
            ));
        }
        return Array();
    }

    static public function Modify($user_id, $newuser=array()) {
        if (!$user_id) return;
        /* uc */
        $curuser = Table::Fetch('user', $user_id);
        if ($newuser['password'] &&function_exists('zuitu_uc_updatepw') ) {
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
                $em = $curuser['email'];
                $un = $newuser['username'];
                $pp = $newuser['password'];
                if ( ! zuitu_uc_updatepw($em, $un, $pp)) {
                        return false;
                }
        }

        /* tuandb */
        $table = new Table('user', $newuser);
        $table->SetPk('id', $user_id);
        if ($table->password) {
                $plainpass = $table->password;
                $table->password = self::GenPassword($table->password);
        }
        return $table->Update( array_keys($newuser) );
}

static public function GetLogin($email, $unpass, $en=true) {
        if($en) $password = self::GenPassword($unpass);
        $field = strpos($email, '@') ? 'email' : 'username';
        $zuituuser = DB::GetTableRow('user', array(
                                $field => $email,
                                'password' => $password,
        ));
        if ($zuituuser)  return $zuituuser;
        if (function_exists('zuitu_uc_login')) {
                return zuitu_uc_login($email, $unpass);
        }
        return array();
```

```
	}

	static public function SynLogin($email, $unpass) {
		if (function_exists('zuitu_uc_synlogin')) {
			return zuitu_uc_synlogin($email, $unpass);
		}
		return true;
	}

	static public function SynLogout() {
		if (function_exists('zuitu_uc_synlogout')) {
			return zuitu_uc_synlogout();
		}
		return true;
	}
}
?>
```

Figure 5.5 PHP sample code

Figure 5.6 shows the same functionality program written in Perl whose OR is 17, OD is 39, PR is 3, I/O is 17, FEis5, EL is 21, and VE is 13.

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

```
$|=1;
local (*F1,*F2); my %farray = (); my $statF1;


# ------------------------------
# traverse directories
sub scan ($) {
   my ($dir) = $_[0];
opendir (DIR, $dir) or die "($dir) $!:$@";
   map {
         (-d) ? scan ($_) : push @{$farray{-s $_}},$_
            unless (-l or -S  or -p or -c or -b);
   } map "$dir/$_", grep !/^\.\.?$/, readdir (DIR); closedir (DIR);
}


# ------------------------------
# get chunk of bytes from a file
sub getchunk ($$) {
  my ($fsize,$pfname) = @_;
  my $chunksize = 32;
  my ($nread,$buff);

  return undef unless open(F1,$$pfname);

  $statF1 = [(stat  F1)[3,1]];
binmode F1;
  $nread = read (F1,$buff,$chunksize);
  ($nread == $chunksize || $nread == $fsize) ? "$buff" : undef;
}


# ------------------------------
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
# compare two files
sub mycmp ($) {
  my ($fptr) = $_[0];
  my ($buffa, $buffb);
  my ($nread1,$nread2);
  my $statF2;
  my ($buffsize) = 16*1024;

  return -1 unless (open(F2,"<$$fptr"));

  $statF2 = [(stat  F2)[3,1]];

  return 0
            if ($statF2->[0] > 1 && $statF1->[1] == $statF2->[1]);

binmode F2;
  seek (F1,0,0);

  do {  $nread1 = read (F1,$buffa,$buffsize);
            $nread2 = read (F2,$buffb,$buffsize);

            if (($nread1 != $nread2) || ($buffacmp $buffb)) {
              return -1;
       }
  } while ($nread1);

  return 0;
}
# ------------------------------
print "collecting files and sizes ...\n";
```

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

```
if (-t STDIN) {
            $ARGV[0] = '.' unless $ARGV[0]; # use wd if no arguments given
            map scan $_, @ARGV;
} else {
            while (<STDIN>)  {
                        s°[\r\n]$°°g;
                        push @{$farray{-s $_}},$_
                         unless (-l or -S  or -p or -c or -b);
            }
}

print "now comparing ...\n";
for my $fsize (reverse sort {$a <=> $b}
.

 keys %farray) {

  my ($i,$fptr,$fref,$pnum,%dupes,%index,$chunk);

  # skip files with unique file size
  next if $#{$farray{$fsize}} == 0;

  $pnum  = 0;
  %dupes = %index = ();

nx:
  for (my $nx=0;$nx<=$#{$farray{$fsize}};$nx++) # $nx now 1..count of files
  {                                       # with the same size
            $fptr = \$farray{$fsize}[$nx];          # ref to the first file
```

```
    $chunk = getchunk $fsize,$fptr;
    if ($pnum) {
            for $i (@{$index{$chunk}}) {
        $fref = ${$dupes{$i}}[0];
                unless (mycmp $fref) {
          # found duplicate, collecting
                  push @{$dupes{$i}},$fptr;
                                next nx;
                }
            }
    }

    # nothing found, collecting
    push @{$dupes{$pnum}},$fptr;
    push @{$index{$chunk}}, $pnum++;
  }
  # show found dupes for actual size
  for $i (keys %dupes) {
    $#{$dupes{$i}} || next;
    print "\n size: $fsize\n\n";
    for (@{$dupes{$i}}) {
       print $$_,"\n";
    }
  }
}
close F1;
close F2;
```

Figure 5.6 Perl sample code

Table 5.3 to 5.11 show the results of seven metrics measured on all programming languages within the designated functionalities, namely, OR, OD, PR, IO, FE, EL, and VE.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Table 5.3 Metrics of string matching program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	39	52	11	1	0	1	11
C#	70	114	4	6	0	13	8
Java	121	151	4	2	0	8	16
Python	14	24	4	15	1	34	23
PHP	101	186	9	1	0	21	19
Perl	49	108	5	9	2	13	17

Table 5.4 Metrics of array copying program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	59	36	28	4	0	4	17
C#	25	43	3	5	0	10	5
Java	131	151	7	3	0	13	18
Python	84	87	20	6	2	15	18
PHP	97	189	7	0	0	15	25
Perl	30	40	3	6	2	10	14

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Table 5.5 Metrics of open and close program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	23	10	14	6	4	9	6
C#	23	33	5	1	11	11	9
Java	69	89	5	4	3	8	8
Python	15	26	9	4	7	29	32
PHP	62	115	17	1	7	17	16
Perl	46	68	2	9	6	18	25

Table 5.6 Metrics of read and write program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	26	13	17	8	5	10	3
C#	28	48	2	2	7	12	5
Java	151	208	6	3	15	21	14
Python	4	8	5	3	3	27	18
PHP	133	259	16	0	5	27	23
Perl	34	55	1	5	6	13	10

Table 5.7 Metrics of append and update program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	41	19	27	9	0	9	9
C#	59	96	4	3	4	13	8
Java	180	214	8	12	7	31	18
Python	6	9	6	6	1	27	14
PHP	83	165	8	1	0	20	21
Perl	59	76	3	10	5	18	10

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Table 5.8 Metrics of numerical computation program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	34	35	6	5	5	10	7
C#	32	48	5	4	0	11	13
Java	151	191	7	3	0	20	20
Python	36	57	8	3	0	45	21
PHP	115	226	12	0	1	16	29
Perl	59	73	4	10	4	15	21

Table 5.9 Metrics of condition program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	77	48	25	8	0	10	15
C#	16	28	0	8	0	10	1
Java	106	136	4	7	0	15	11
Python	13	22	8	3	0	22	23
PHP	197	250	1	5	0	17	19
Perl	63	95	1	5	3	16	20

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

Table 5.10 Metrics of branch program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	24	25	8	1	0	3	7
C#	10	19	0	5	0	6	2
Java	91	110	8	3	0	15	15
Python	11	22	11	6	2	33	20
PHP	167	332	7	1	3	26	28
Perl	60	130	4	9	4	16	19

Table 5.11 Metrics of loop program in each language

PL	OR	OD	PR	IO	FE	EL	VE
C	40	38	15	8	0	20	17
C#	5	9	0	1	0	1	1
Java	90	119	4	6	0	10	11
Python	19	35	10	2	1	49	24
PHP	113	230	21	0	0	21	30
Perl	75	123	9	10	9	33	17

Further analyses based on the proposed method yield the results listed in the tables that follow.

Table 5.12 Range of each group

Group	LOC	OR	OD	PR	IO	FE	EL	VE	FG
Compiled	290	262	312	39	18	21	47	48	412
Interpreted	492	410	663	45	22	27	68	44	649

Table 5.13 Normalized average of each group

Group	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
Compiled	0.29	0.24	0.24	0.22	0.25	0.07	0.22	0.23	0.20	3	31
Interpreted	0.30	0.16	0.17	0.18	0.22	0.09	0.32	0.46	0.22	8	23

Table 5.14 Normalized average of each language

PL	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	0.17	0.17	0.11	0.46	0.27	0.05	0.17	0.24	0.12	4	19
C#	0.21	0.11	0.15	0.06	0.23	0.08	0.20	0.13	0.13	1	21
Java	0.48	0.46	0.48	0.15	0.24	0.09	0.31	0.31	0.34	4	53
Python	0.27	0.07	0.06	0.23	0.25	0.06	0.43	0.47	0.23	8	18
PHP	0.39	0.29	0.33	0.22	0.04	0.05	0.28	0.54	0.24	10	31
Perl	0.24	0.12	0.12	0.08	0.35	0.15	0.24	0.38	0.19	9	19

Table 5.15 Normalized standard deviation of each group

Group	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
Compiled	0.22	0.24	0.24	0.27	0.23	0.17	0.18	0.18	0.18	4	27
Interpreted	0.19	0.17	0.17	0.20	0.25	0.15	0.20	0.23	0.14	8	41

Table 5.16 Normalized standard deviation of each language

PL	LOC	OR	OD	PR	IO	FE	EL	VE	FG	CCM	HCM
C	0.16	0.10	0.06	0.32	0.24	0.11	0.20	0.15	0.10	3	13
C#	0.15	0.09	0.12	0.07	0.24	0.17	0.12	0.13	0.10	3	21
Java	0.21	0.28	0.27	0.13	0.19	0.22	0.19	0.19	0.21	6	30
Python	0.10	0.11	0.05	0.22	0.27	0.11	0.23	0.24	0.10	10	61
PHP	0.23	0.20	0.18	0.22	0.14	0.10	0.16	0.21	0.12	10	32
Perl	0.16	0.09	0.09	0.11	0.22	0.21	0.14	0.21	0.18	7	17

Table 5.12 shows the range of the complied language group and the interpreted language group. Tables 5.13 and 5.14 list the normalized average of each group and each language. The corresponding normalized standard deviation of each group and language are shown in Tables 5.15 and 5.16.

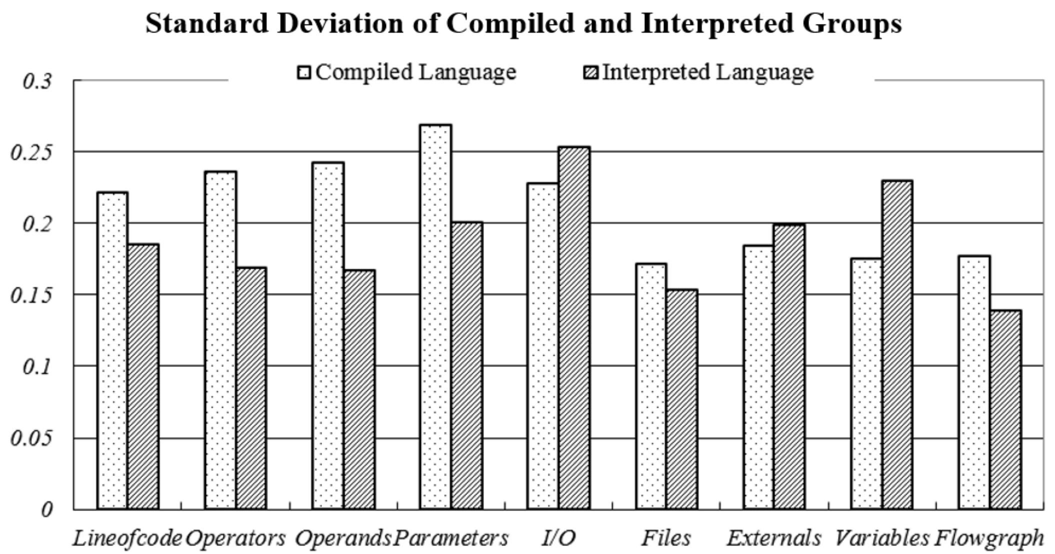

Figure 5.7 Standard deviation of Compiled and Interpreted groups

Figure 5.7 shows the proportional distribution of standard deviation of the compiled and interpreted groups. It can be seen that source level of the compiled language group has more difficulty than the interpreted one in terms of Lines of Code, Operators, Operands, Parameters, Files, and Flow graph. Nevertheless, I/O, Externals, and Variables are slightly less than those of the interpreted group.

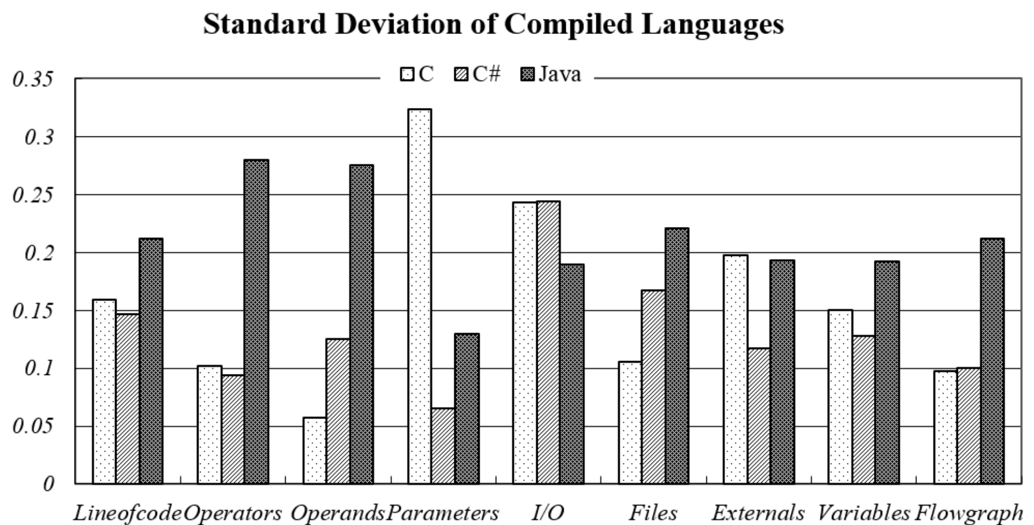

Figure 5.8 Standard deviation of Compiled Language: C, C# and Java

Figure 5.8 shows the proportional dispersion of different compiled languages, which are C, C#, and Java. Of the nine indicators, C language has the highest difficulty in terms of Parameters and Externals; C# has the highest difficulty in I/O; while Java has the highest difficulty in Lines of Code, Operators, Operands, Files, Variables, and Flow graph. This experiment shows that Java program exhibits the highest discernible difficulty variations of the three compiled programming languages. The average of these nine metrics shows that C is more complicated than C# and C# exhibits the least discernible difficulty variations in the compiled programming languages.

From the lesser measure standpoint, the results also show that C has the least Operands, Files, and Flow graph; C# has the least Lines of Code, Operators, Parameters, Externals, and Variables; while Java has the least difficulty in I/O.

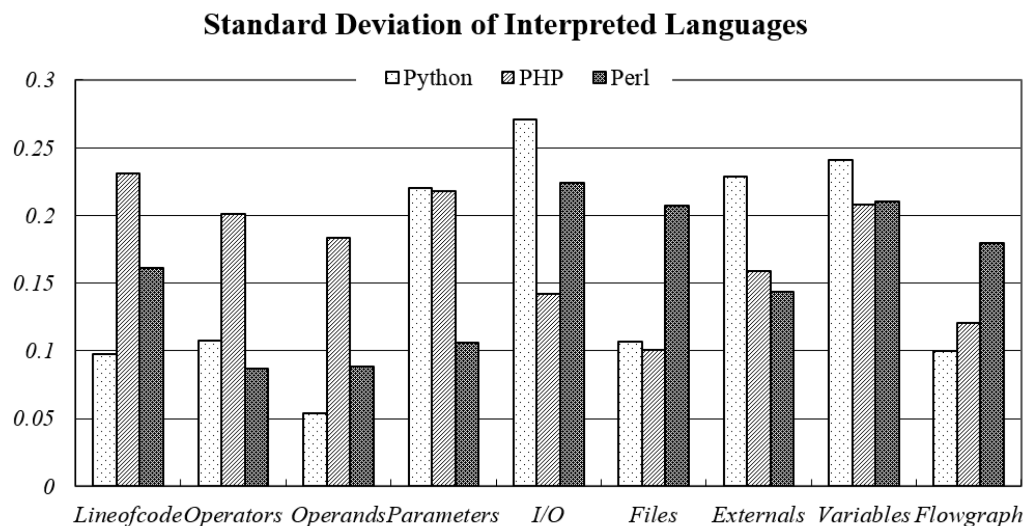

Figure 5.9 Standard deviation of Interpreted Language, Python, PHP and Perl

Figure 5.9 shows standard deviation of Python, PHP, and Perl. Of the nine metrics, Python has higher difficulty than others in terms of Parameters, I/O, Externals and Variables; PHP has the highest difficulty in Line of code, Operators, and Operands; while Perl has more difficulty than others in Files, and Flowgraph.

The results also show that Python has the least Lines of Code, Operands, and Flow graph; PHP has the least I/O, Files, and Variables; while Perl has the least Operators, Parameters, and Externals. Thus, code difficulty of these three interpreted programming languages is not easily discernable from the standard values alone. With the help of average complexity value in Table 5.13, it can be seen that Python is the simplest language of the interpreted group, while PHP exhibits higher difficulty than Perl in the combined average and standard consideration.

5.3 Discussion

Consider the two comprehensive measures, namely, CCM and HCM, the compiled group has lower CCM than its interpreted counterpart, while the HCM value jumps higher due solely to Java. However, measurements of he compiled group as their control flow and relationships among the constituent metrics are closely related. Table 5.8-5.16 confirm this finding. The compiled group, with the exception of Java, is highly compact, having fewer metrics statistics than their interpreted counterpart except Python. For example, the CCM measure that is based primarily on CFG exhibits close relationship with program difficulty for the compiled group. On the other hand, the HCM complexity measure is somewhat out of proportion owing to Java richness of operators and operands. In other words, the value of OR and OD of Java are approximately 5-6 and 6-10 times higher than its peers C and C#, respectively. PHP exhibits similar traits among its peers Python and Perl. Interestingly, this relationship comparison demonstrates just the opposite to actual implementation of programming language selection, that is, Java and PHP are the two most popularly used languages within their peer group.

Table 5.17 shows the normalized average difficulty among the programming languages under specific functional applications. For example, C and Python exhibit the least difficulty based on their respective group in the string matching problem, while C# and Python are the best in loop comparison. Table 18 shows the overall degree of difficulty help decide suitability of language selection for the program written for specific functional applications.

Table 5.17 Result summary of language difficulty under specific functional applications

Language functional applicability		C	C#	Java	Python	PHP	Perl
Matching	String Matching	0.36	0.43	0.51	0.54	0.67	0.59
	Array Copying	0.40	0.28	0.56	0.84	0.53	0.47
I/O	File open/close	0.33	0.33	0.48	0.53	0.59	0.61
	File read/write	0.41	0.23	0.68	0.39	0.71	0.39
	I/O append/update	0.40	0.21	0.73	0.32	0.50	0.53
Computation	Numerical Computation	0.50	0.30	0.57	0.49	0.63	0.54
Comparison	Condition	0.47	0.26	0.49	0.35	0.70	0.52
	Branch	0.24	0.15	0.50	0.47	0.76	0.64
	Loop	0.46	0.06	0.44	0.36	0.64	0.58

Table 5.18 Summary of language difficulty under the specific functional application

Language functional applicability		C	C#	Java	Python	PHP	Perl
Matching	String Matching	√					
	Array Copying		√				
I/O	File open/close	√	√				
	File read/write		√				
	I/O append/update		√				
Computation	Numerical Computation		√				
Comparison	Condition		√				
	Branch		√				
	Loop		√				

Another noteworthy result is that all languages in the same group exhibit alternate high difficulty across the measuring metrics of the underlying application domain. The statistical summary of language applicability in Table 5.18 shows that given the functions under study, one language may be the most viable candidate of implementation choice than others based on the findings.

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

CHAPTER VI

CONCLUSION

With the rapid advancement in software industries, software metrics become the basis for software management and are crucial to the accomplishment of software development. A straightforward method is proposed to measure the difficulty of source code written in C, C#, Java, Python, PHP, and Perl by means of measuring metrics, namely, operators, operands, parameters, inputs and outputs, file operations, external functions or libraries, variable declarations, and flow graph. This study focuses on four types of problems to compare the complexity of different language implementation. The findings in this thesis reveal that source code written in compiled languages is inherently more complex than interpreted languages. The reasons may depend primarily on the nature of application to be performed by the target software, as interpreted software is likely to be smaller and less involved than its compiled counterpart. Each language exhibits its source level difficulty through the aforementioned metrics, CCM, and HCM. The results of metric measurements show that no one language is suitable for all types of problems. Anyhow, C# and Python dominate their respective language group in implementation complexity, particularly when the issue of source code difficulty is concerned. This primarily depends on individual programmer's experience in programming language. As such, the findings may assist developers in language selection that is most appropriate yet least difficult for the task.

Despite the significant role played by software metrics, studies and researches in this field are still immature. New paradigms and programming languages are being invented, in particular, design patterns, automated code generation, 5GL, and user computing. Unfortunately, these techniques bring about accidental and inherent complexities [12] that grow exponentially out of control. The effect renders software project management to inevitably fall behind technology in terms of productivity measurement, cost estimation, project planning, and the like. In addition, there are no adequate international standards to warrant the software products being distributed. More researches need to be done to fill in the blank of a firm theoretical foundation and assurance of methods and metrics. Such endeavors will foster the development of software applications that could serve the insatiable needs of the evolving modern digital world.

REFERENCES

1. T. Honglei, S.W., and Z Yanan, *The Research on Software Metrics and Software Complexity Metrics.* Proc. IEEE International Forum on Computer Science-Technology and Application Jan. 2009: p. 131-136.
2. Zhou, S.Y.a.S., *A Survey on Metric of Software Complexity.* Proc. IEEE International Conference on Information Management and Engineering, 2010: p. 352-356.
3. D. I. De Silva, N.K., and H. Perera, *Applicability of Three Complexity Metrics.* Proc. IEEE International Conference on Advances in ICT for Emerging Regions 2012: p. 82-88.
4. McCabe, T.J., *A Complexity Measure.* IEEE Transactions on Software Engineering, 1976: p. 308-320.
5. Halstead, M.H., *Elements of Software Science.* Elsevier Science, 1977.
6. Shao., Y.W.a.J., *Measurement of the Cognitive Functional Complexity of Software.* Proc. IEEE International Conference on Cognitive Informatics, 2003: p. 67-74.
7. A. Tahir, S.G.M., *A Systematic Mapping Study on Dynamic Metrics and Software Quality.* Proc. IEEE International Conference on Software Maintenance 2012: p. 326-335.
8. M. K. Debbarma, S.D., N. Debbarma, K. Chakma, and A. Jamatia, *A Review and Analysis of Software Complexity Metrics in Structural Testing.* International Journal of Computer and Communication Engineering: p. 129-133.
9. S. Sabharwal, R.S., and P. Kaur, *Software Complexity: A Fuzzy Logic Approach.* Proc. IEEE International Conference on Communication, Information & Computing Technology, 2012: p. pp. 1-6.
10. A. Shukla, P.R., *A Generalized Approach for Control Structure based Complexity Measure.* Conf. on Recent Advances in Information Technology, 2012.
11. A. Capiluppi, P.F., and C. Boldyreff, *Code Refactoring: Evaluating the Effectiveness of Java Obfuscations.* 19th Working Conference on Reverse Engineering, 2012: p. 71-80.
12. S. Sarala, P.A.J., *Information Flow Metrics and Complexity Measurement.* Computer Science and Information Technology(ICCSIT) of IEEE International Conference, 2010: p. 575-578.
13. J.J. Vinju, M.W.G., *What does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric.* Source Code Analysis and Manipulation(SCAM) of IEEE 12th International Working Conference, 2012: p. 154-463.

14. O. Hummel, S.B., *A Pragmatic Means for Measuring the Complexity of Source Code Ensembles.* 4th International Workshop on Emerging Trends in Software Metrics(WETSoM), 2013: p. 76-79.
15. S. Misra, I.A., and M. Koyuncu, *An inheritance complexity metric for object-oriented code: A cognitive approach.* Sadhana, 2011: p. 317-337.
16. L. Prasad, A.N., *Experimental analysis of different metrics(object-oriented and structural) of software.* CICSYN'09. First International Conference on Computational Intelligence, Communication Systems and Networks, 2009: p. 235-240.
17. R.G. Kula, K.F., N. Yoshida, and H. Lida, *Experimental study of quantitative analysis of maintenance effort using program slicing-based metrics.* APSEC '12 Proc. 19th Asia-Pacific Software Engineering Conference, 2012: p. 50-57.
18. P. Singh, S.S., and J. Kaur, *Tool for generating code metrics for C# source code using abstract syntax tree technique.* ACM SIGSOFT Software Engineering Notes, 2013. **38**: p. 1-6.
19. A. Abusasd, I.M.A., *The correlation between source code analysis change recommendations and software metrics.* ICICS '12 Proc. 3rd International Conference on Information and Communication Systems, 2012.
20. Y. Sasaki, T.I., K. Hotta, and H. Hata, *Preprocessing of metrics measurement based on simplifying program structures.* 19th Asia-Pacific Software Engineering Conference, 2012.
21. Norman E. Fenton, S.L.P., *Software Metrics: A rigorous approach.* 2003: p. 18-33.
22. Arockiam, A.A.a.L., *A Survey on Metric of Software Cognitive Complexity for OO Design.* World Academy of Science, Engineering and Technology,, 2011. **58**.
23. RoberE.Park, W.G., WillianA.Florac, *Goal Driven Software Measurement A Guide Book.* Software Engineering Institute, CarnegieMellonUniversity, 1996.
24. M.K. Debbarma, N.K., and A. Saha, *Static and Dynamic Software Metrics Complexity Analysis in Regression Testing.* International Conference on Computer Communication and Informatics, 2012.

APPENDIX

VITA

Liu Xiao received a Bachelor degree in Computer Science and Technololgy from the Department of Computer Science, Faculty of Science, Northwestern Polytechnical University in 2010.

Publication

Liu Xiao and PeraphonSophatsathit, “An Empirical Study of Source Level Complexity”,Proceedings of the 5th International Conference on Multimedia Information Networking and Security (MINES 2013), Beijing, China, November 1-3, 2013, pp. 472-475.

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University