AUTOMATING THE PROCESS OF MEASURING COMPLEXITY OF JAVA PROGRAMMING ASSIGNMENT

MAWARNY BINTI MD. REJAB

ROHAIDA ROMLI

FACULTY OF INFORMATION TECHNOLOGY
UNIVERSITI UTARA MALAYSIA

2005

PENGAKUAN TANGGUNGJAWAB (DISCLAIMER)

Kami, dengan ini, mengaku bertanggungjawab di atas ketepatan semua pandangan, komen teknikal, laporan fakta, data, gambarajah, illustrasi, dan gambar foto yang telah diutarakan di dalam laporan ini. Kami bertanggungjawab sepenuhnya bahawa bahan yang diserahkan ini telah disemak dari aspek hakcipta dan hak keempunyaan. Universiti Utara Malaysia tidak bertanggungan terhadap ketepatan mana-mana komen, laporan, dan maklumat teknikal dan fakta lain, dan terhadap tuntutan hakcipta dan juga hak keempunyaan.

We are responsible for the accuracy of all opinions, technical comments, factual reports, data, figures, illustrations and photographs in this report. We bear full responsibility for the checking whether material submitted is subject to copyright or ownership right. UUM does not accept any liability for the accuracy of such comments, reports and other technical and factual information and the copyright or ownership rights claims.

PROJECT LEADER:

……
Name: Mawarny Md. Rejab

MEMBER:

……
Name: Rohaida Romli

ACKNOWLEDGEMENT

The authors wish to express their gratitude to the Faculty of Information Technology for the financial support given to undertake this research project. We also would like to thank to all our colleagues and individuals for their support and assistance.

ABSTRACT

Programming is a complex intellectual activity and a core skill for first year IT students. Several researches have shown that most students often write programs without considering the quality of the program. Due to this matter, an automatic assessment system has become one of the most important tools to evaluate and grade programming assignment including judgments of the quality of programming solutions. Besides considering the correctness of the output program, the automatic assessment system also focuses on the complexity factor in ensuring the consistency and accuracy of the hand-marking programming assignments and improve the quality of students' programming solution. This study is proposed as an effort to assist lecturers of the Introductory Java Programming course in evaluating and grading program assignments by considering the complexity factor. Besides selected traditional software metrics such Lines Of Code and Cyclomatic Complexity, several object-oriented metrics are adopted to measure the program complexity namely, Respond For a Class (RFC), Number of properties (SIZE2), Number of classes (NCL), Operation Complexity (OpCom), Operation Argument Complexity (OAC) and Attributes Complexity (AC). Specific score and weight will be given for each selected metric as a measurement of the program complexity. The summary of report that contains a complexity analysis and complexity mark awarded to the student will be generated automatically using a developed prototype. Thus, this approach will be implemented to provide a tool in order to improve the process of evaluating the Introductory of Java programming assignment for the Faculty of Information Technology.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

This chapter consists of an overview of the research study, problem statement, objective, scope and significance of the study.

1.1 An overview of the research study

Measurement of software complexity has been of great interest to several researchers in software engineering. Software complexity has been shown to be one of the major contributing factors in developing software. According to the Lake and Cook (1994), software complexity is defined as an objective measure of how difficult it may be for a programmer to perform common programming tasks, such as understanding, testing, or maintaining, on a piece of software. Measurement of software complexity does not measure the complexity itself, but instead measures the degree to which those characteristics though to lead complexity exist within the code. For example, a program may be considered complex to test if it has complicated control flows and many different execution paths. Hence, a possible complexity measure will be the number of conditional and looping statements.

Ideally, complexity measures should have both descriptive and prescriptive components (Watson and McCab, 1996). Descriptive measures identify software that is error-prone, hard to understand, hard to modify, hard to test, and so on. Prescriptive

measures identify operational steps to help control software, for example splitting complex modules into several simpler ones, or indicating the amount of testing that should be performed on given modules.

A large number of software metrics have been proposed over the last decade for measuring the complexity of programs. Hundreds of traditional software complexity metrics and a large number of proposed object-oriented programming metrics have been defined in measuring complexity of software. As discussed by Lake and Cook (1994), traditional software complexity metrics are usually divided into classes namely, lines of code (LOC), data structures metrics, control flow metric (cyclomatic complexity), information flow metric, and software science metric (based on four parameters: number of unique operators, number of unique operands, total number of operators, total number of operands).

Meanwhile, an Object-Oriented (OO) complexity metrics can be divided into several categories such as class related metrics, method related metrics, inheritance metrics, metrics measure coupling and metrics measure general (system) software production characteristics (Xenos et. al, 2000). Measurement of OO system complexity requires the understanding of several constructs and the relationships between those constructs. According to Tegarden and Sheetz (1992), a model of OO system complexity consists of the system complexity, structural complexity, and perceptual complexity constructs.

Complexity measures are also important in the assessment of students programs. The complexity measures in program assessment can provide a way to assist lecturers in ensuring the consistency and accuracy of handmarking programming assignments and improve the quality of students programming solutions. Nowadays, automatic assessment systems focus not just on the correctness of a program's output, but also analyze the output, the style of writing, the complexity and other factors that depend on the scheme of the program (Zarina, 1999). This study is proposed as an effort to assist lecturers of introductory Java programming course in improving the consistency and accuracy of the marking standard.

1.2 Problem Statement

Programming is a complex intellectual activity and a core skill for first year IT students. Research has shown that most students are able to write programs; however, their programs are often poorly constructed because they do not consider different solutions to a program (Truong et. al, 2004). Novice students often try to solve as quickly as possible without thinking about the quality of their programs (Vizcaino et. al, 2000). Thus, this research attempts to solve the difficulties in ensuring consistency and accuracy of the marking standard in terms of measuring the complexity of students' program. Besides, this study is also an effort to enhance a previous study by Rohaida et.al (2004), which did not include the complexity factor in the students' program assessment. Furthermore, there is no proper tool to automate the process of measuring the complexity of students' Java programming assignments in the faculty of Information Technology, Universiti Utara Malaysia. Thus, as an effort to underlying this situation, an automation of measuring the complexity for the students' Java programming assignments is proposed.

1.3 Objective

The objective of this study is to automate the process of measuring the complexity for students' Java programming assignments in maintaining a uniform marking standard.

1.4 Scope of the Study

This study focuses on the measurement of complexity of the students' programming assignments of a course, *Introduction To Programming* (TIA 1013). This measurement contributes an important part in the assessment of students' Java programs. The measurement of a program complexity focuses on the area of basic object-oriented programming concepts. Based on a preliminary study shown in Appendix A, selected traditional software metrics such as Lines Of Code and Cyclomatic Complexity and several object-oriented metrics are used namely, Respond For a Class (RFC), Number of properties (SIZE2), Number of classes (NCL), Operation Complexity (OpCom), Operation Argument Complexity (OAC) and Attributes Complexity (AC).

1.5 Significance of the Study

This study will improve consistency in evaluating the complexity of the students' Java programming assignments. The complexity analysis based on different program abstraction levels can provide a way to maintain the marking standard.

1.6 Conclusion

This chapter gives an overview of the study including the problem statement, objective, scope and significance of the study. The following chapter will review on the background of software complexity and related studies on the program complexity measurement.

CHAPTER TWO

LITERATURE REVIEW

This chapter will focus on reviews on software complexity, static analysis, software complexity metrics, and related work on assessment of program complexity.

2.1 Software Complexity

Software complexity has been defined and interpreted in many ways over the years. Basili (1980) defines the term software complexity as " ….a measure of the resources expended by another system while interacting with a piece of software. If the interacting system is people, the measures are concerned with human efforts to comprehend, to maintain, to change, to test, etc, that software". Ramamoorthy (1985) pointed out the definition of software complexity as the degree of difficulty in analysis, design, implementation and testing of software.

Curtis (1979) has suggested a definition of complexity that refers to the characteristic of a software, which makes it difficult to understand or work with. In the development phase, complexity strongly influences the effort required to debug and test the program modules and subsystems. In the maintenance phase, complexity determines how difficult it will be located and corrected undetected implementation errors, and also how much effort will be required to modify programs modules to incorporate specification changes (Curtis, 1985). According to Zuse (1991), software complexity is

the difficulty to maintain, to change and understand software. It primarily deals with the characteristics of software that affect the program performance.

Programming behaviors are very complex and can be influenced by the experience and ability of the programmer and the programming environment. Referring to Yourdan (1979), most problems in programming occur because human beings make mistakes without considering the limitation of the complexity capacity.

Based on several definitions, complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software. The term software complexity is a measure of difficulty of performing tasks that have been applied to the interaction between a program and programmer.

2.2 Static Analysis

There is a strong connection between software complexity and testing. Complexity is a common source of error in software. The term software complexity is used to identify software that is error-prone, hard to understand, hard to modify, and so on. As such static analysis has been selected as a testing approach for the assessment of program complexity.

Referring to Coward (1988), static analysis is a testing technique that does not involve the execution of the software with data. The program source code structure and syntax are inspected so as to highlight static errors and produce statistical information for the programmer. Based on Truong et.al (2004), static analysis is a process of examining

source code without executing the program. It is used to locate problems in code including potential bugs, unnecessary complexity and high maintenance areas.

According to Sommerville (2004), static analysis is an automated technique of program analysis where the program is analyzed in detail to find potentially errorness conditions. The stages involved in static analysis include:

a) Control flow analysis – This stage identifies and highlights loops with multiple exits or entry points and unreachable codes. An unreachable code is a code that is surrounded by unconditional goto statements or that is in a branch of a conditional statement where the guarding condition can never be true.

b) Data use analysis – This stage highlights how variables in the program are used. It detects variables that are used without previous initialization, variables that are written twice without an intervening assignment and variables that are declared but never used. Data use analysis also discovers ineffective tests where the test condition is redundant. Redundant conditions are conditions that are either always true or always false.

c) Interface analysis- This analysis checks the consistency of routine and procedure declarations and their use. Interface analysis can also detect functions and procedures that are declared and never called or function results that are never used.

d) Information flow analysis – This phase of the analysis identifies the dependencies between input and output variables. While it does not detect anomalies, it shows how the value of each program variable is derived from other variable values.

e) Path analysis – This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed in that path. It essentially unravels the program's control and allows each possible predicate to be analyzed individually.

In this study, an automated tool has been developed to examine the source code without executing the program in order to measure the value that will be used by the software complexity metrics.

2.3 Software Complexity Metrics

Software metrics are a well-known way to measure the quality of programs. Software complexity metrics have been developed to identify parts of program that are likely to be difficult to test, understand, or error-prone. A large number of software complexity metrics have been proposed over the last decade for measuring the complexity of programs. Hundreds of traditional software complexity metrics and the large number of proposed object-oriented software complexity metrics have been defined in measuring the complexity of software.

2.3.1 Traditional software complexity metrics

There are several traditional software complexity metrics that have been proposed by some researchers since 1976 such as Cyclomatic Complexity, Lines of Code, Software Science Metric and so on. However, in this study, only two traditional software complexity metrics are adopted in measuring the program complexity, namely:

a) Lines of code

According to Conte (1986), a line of code is defined as any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. This metrics quantifies the size of program, which does not take the coding style into account. The larger size of program, the more paths it contains and hence the more difficult it will be to work or to understand.

b) Cyclomatic Complexity

Referring to Mc Cabe (1976), cyclomatic complexity is a measure of module control flow complexity based on control flow graph. Control flow graphs describe structure of software modules, which the module corresponds to a single function or method. Each method or function can be represented into a flow graph that consists of nodes and edges. The nodes represent computational statements or expressions and the edge represent transfer of control between nodes. Based on the nodes and edges, the cyclomatic complexity calculation is defined as below:

$$v(G) = e - n + p$$

Based on the formula of cyclomatic complexity above, *e* is the number of edges, *n* is the number of nodes, and *p* is the number of connected components. Referring to Tegarden et.al (1992), connected components are the nodes in the module that can be reached from outside the graph or that can transfer control outside the graph. This corresponds to the number of entry and exit points for the module.

According to Rosenberg (1998), a method with a low cyclomatic complexity is generally better, although it may mean that decisions are differed through message passing, not that the method is not complex. The greater the cyclomatic complexity is the more execution paths there are through the method, and the harder to understand the method. Table 2.1 depicts the range of cyclomatic complexity value.

Table 2.1: Range Of Cyclomatic Complexity

Cyclomatic Complexity	Risk Evaluation
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21 – 51	complex, high risk program
greater than 50	very high risk, untestable program

2.3.2 Object-oriented software complexity metrics

The object-oriented paradigm for software development is different with the traditional procedural paradigm. As a result,, some researchers and practitioners suggest that traditional software metrics are inappropriate for measuring object-oriented programming complexity. There are several object-oriented concepts such as polymorphism, inheritance, and encapsulation that fail to be captured using traditional

metrics. Moreau and Dominick (1989) point out that many existing software metrics that have been utilized within conventional programming environments are inappropriate for evaluating object-oriented systems in certain circumstances.

On the other hand, some researchers suggest that several traditional software metrics can still be used for object-oriented paradigms. A valid reason for applying traditional software metrics is that most traditional metrics have been widely used, well understood, and have become accepted as a "standard" for traditional functional or procedural programs. As a result, Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center suggests only three traditional software metrics that are applicable to object-oriented programs namely, cyclomatic Complexity (McCabe), Lines of code (LOC), and Comment Percentage.

According to Chidamber and Kemerer (1995), six object-oriented metrics have been defined which are Weight Methods per Class (WMC), Response For Class (RFC), Lack Of Cohesion (LCOM), Coupling Between Object Classes (CBO), Depth of Inheritance Tree (DIT) and Number of Children (NOC). Furthermore, Li et. al (1995) defined ten metrics which include five out of six metrics that have been defined by Chidamber and Kemerer with addition of five more metrics: Message-Passing Coupling (MPC), data Abstraction Coupling (DAC), Number Of Methods (NOM), Number of Semicolons (SIZE1) dan Number of Propeties (SIZE 2).

Moreau and Dominick (1989) defined three metrics which are Message Vocabulary Size (MVS), Inheritance Complexity (IC) and Message Domain Size (MDS). According to Chen and Lu (1993), a new set of metrics has been proposed for

object-oriented design namely operation complexity, operation argument complexity, attribute complexity, operation coupling, and cohesion metrics.

Many researchers such as Henderson-Sellers and Brito e Abreu classify the object-oriented metrics based on different dimensions. The selected object-oriented metrics are primarily applied to the concepts of classes, coupling and inheritance and based on the levels of a software system. A brief description of object oriented terms for metrics is given in Table 2.2.

Table 2.2: Key Object-Oriented Terms for Metric

Term	Description
Attribute	Define the structural properties of classes, unique within a class, generally a noun
Class	A set of objects that share a common structure and common behavior manifested by asset of methods, the set serves as a template from which an object can be instantiated (created)
Cohesion	The degree to which the methods within a class are related to one another.
Coupling	Object X is coupled to object Y if and only if X sends a message to Y.
Inheritance	A relationship among classes, wherein an object in a class acquires characteristic from one or more other classes.
Instantiation	The process of creating an instance of the object and binding or adding the specific data.
Message	A request that an object makes of another object to perform an operation.
Method	An operation upon on object, defined as part of the declaration of a class.
Object	An instantiation of some class which is able to a save a state (information) and which offers a number of operations to examine or effect this state.

According to Tegarden (1992), the complexity of object-oriented systems can be represented by a set of measures defined at different levels. The levels are variable level, method level, object level and system level.

The variable level is associated with the definition and use of variables throughout the system. The method level refers to the defined operation of a class. At this level, the control flow graph model can be used to represent the control flow in a method. The object level combines variable and method complexity. On the other hand, the system level is associated with the classes in the system, object hierarchy, inheritance, message passing and methods defined in the system.

Even thought, several researchers and practitioners have proposed hundreds of software metrics, only selected metrics are applied to extend the selected concept in object-oriented and based on the levels of a software system. For example, Software Assurance Technology Center (SATC) applied both traditional metrics and object-oriented metrics for the object oriented system. Table 2.3 presents an overview of metrics applied by the SATC for object-oriented systems. The first three metrics in Table 2.3 are examples of traditional metrics and the next six metrics are especially applied to object-oriented system.

Table 2.3: SATC Metrics for Object-Oriented Systems

Source	Metric	Object-oriented construct
Traditional	Cyclomatic Complexity	Method
Traditional	Lines Of Code (LOC)	Method
Traditional	Comment Percentage (CP)	Method
Object-oriented	Weighted Method per class (WMC)	Class/Method
Object-oriented	Response for a class (RFC)	Class/Message
Object-oriented	Lack of cohesion of methods (LCOM)	Class/Cohesion
Object-oriented	Coupling between objects (CBO)	Coupling
Object-oriented	Depth of inheritance tree (DIT)	Inheritance
Object-oriented	Number of Children (NOC)	Inheritance

In this study, several object – oriented complexity software metrics are adopted in measuring software complexity namely Operation Complexity (OP), Attribute Complexity (AC), Operation Arguments Complexity (OAC), Number of Properties and Response For Class (RFC). All these software metrics are applied to measure the program complexity based on selected concepts in object-oriented, which are attribute, class, message, method and object. This measurement has been defined at the variable level, method level and object level of software systems. The detailed explanation of selected metrics will be discussed later in chapter 3.

2.4 Related Work on Assessment of Program Complexity

There are several studies done on automation program assessment that take into consideration on the complexity factor. The automatic assessment system focuses not on the correctness of the output program, but analyses the output, the style of writing, the complexity and other factors depending on the scheme of the program (Zarina, 1999). Some of researchers focus on one factor of the software quality, whereas others consider several combinations of quality factors. Referring to Jackson (1996), the University of Liverpool developed an automatic grading system, which measures the quality of students' program in five main areas, namely correctness, style, efficiency, complexity, and test data coverage. For complexity assessment, the system applied McCabe's metric in order to determine the value of cyclomatic complexity.

Zin and Foxley (1991) built an automatic assessment system, called *analyse*, to mark students' program in an introductory or intermediate programming course. There are five main components used in *analyse* to compute the score for program quality, which are maintainability, structural weakness, dynamic correctness, dynamic efficiency, and program complexity. Measurement of program complexity includes static analysis for the occurrence frequency of gotos, reserved words, operators, loop, conditional statements, assignment statements, function calls, complexity of expression, and methods of types.

Hung et.al (1993) developed ASSESS to mark factors in development effort, reliability, style, execution efficiency, and complexity. Hung's evaluation of a student's performance in programming is based on the use of four software metrics, which are programming skills, complexity, programming style, and programming efficiency.

Furthermore, Mengel and Yerramilli (1999) used the Verilog Logiscope WinViewer program, to automate static analysis of a student's program. This system was used to calculate values for a series of selected metrics such as McCabe Cyclomatic complexity, and number of function. The quality of the programs was primarily defined as the conformance to the requirements of the program assignment with a small program size, small complexity, and high modularity.

The Learning Technology Research (LTR) group at Nottingham University developed a coursework system, called *Ceilidh*. *Ceilidh* is designed for the assessment of a student's coursework in Computer Science and the administration of the corresponding courses. In order to identify the marking standard, several metrics were adopted including complexity metrics.

According to Foxley et.al (1996), *Ceilidh* was used widely with around 15 different programming languages, *Ceilidh* ran on a UNIX operating system and required knowledgeable system staff to install and maintain. Several limitations occurred in Ceilidh due to difficulties to understand, maintain, and support. At the end, Ceilidh was redesigned using object-oriented methods and re-implemented to come out with a new system, called CourseMaster.

Truong et.al (2004) introduced a static analysis framework which can be used to give novice students practice in writing better quality Java programs and to assist teaching staff in the marking process. This framework is integrated into the Environment for Learning to Program (ELP). ELP is an online , active, collaborative and constructive environment for learning to program, which provides functions for automatic assessment of students work in Java. In order to measure the quality of programs, cyclomatic complexity is adopted in the framework because it provides useful information about the structure of a program.

2.5 Conclusion

In brief, this chapter has highlighted the concepts of software complexity, the relationship between software complexity and static analysis and reviewed in detail aspects of software complexity metrics. Besides, the related works of program complexity assessment have also been discussed. The following chapter will explain the methodology used for this study.

CHAPTER THREE

METHODOLOGY AND PROTOTYPE DESIGN

This chapter introduces the methodology that was used throughout the study and discusses the prototype design. After considering several aspects in choosing a suitable methodology for this study, the Customized System Development Research Methodology, recommended by Nunamaker et.al (1991) was adopted. Four main phases were involved in this study, namely:

i. Constructing a conceptual issue

ii. Prototype design

iii. Prototype development

iv. Prototype testing

3.1 Construct a Conceptual Issue

This initial phase involved two main activities, namely:

3.1.1 Requirements Gathering

A preliminary study was conducted in order to gather and capture all related requirements of the study. The preliminary study consists of a set of questions that was distributed among experienced programming lecturers. The study was meant to gauge to what extent lecturers teach introductory programming course such as *Pengaturcaraan Awalan* (TIA1013) evaluate the program source code in marking the Java programming

assignment. In addition, the preliminary study provides a better understanding on the development of a prototype by identifying the following items:

- Evaluation items for program source code such as lines of code, number of classes, and data types of variables in order to identify suitable software metrics.
- Marking schema for each evaluation items.

A sample of the preliminary study is shown in the Appendix A.

3.1.2 Requirements Analysis

Based on the result analysis of preliminary study shown in the Appendix B, all the captured requirements are represented by using the Unified Modeling Language (UML). UML is a graphical language for visualizing, specifying, constructing and documenting the deliverables of software product (Booch,1998). The modeling language of UML is used in representing the outcome of this phase. Use case diagram and use case specifications have been produced in representing the captured requirements.

In this study, several use cases were defined. Each use case shows how the actor interacts with the system and what the system does. The use case has a set of sequence actions and performs observable results to a particular actor, who interacts with the system. The use cases that have been defined in this study are:

- Set weight value of metrics
- Manage program complexity

Figure 3.1 depicts the use case diagram for this study. As shown, there is only one actor involved in this study, namely the lecturer. Lecturer is a person who teaches Java programming course and plays an important role in preparing and managing the

source needed in processing the Java program assessment, such as student's Java programming assignment, program schema, and weight value for measuring the complexity of program.

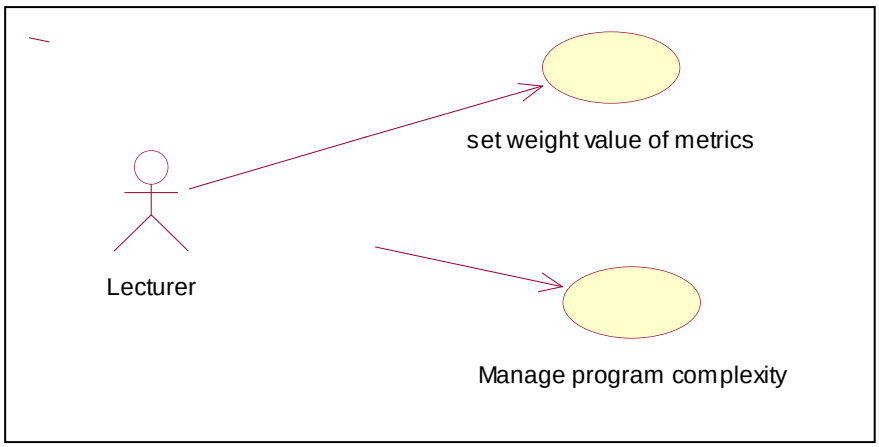


Figure 3.1: Use Case Diagram

In addition, use case specifications are also identified in order to provide a description of the interaction between actors and use case. The use case specification for Set Weight Value of Software Metrics is depicted in Table 3.1 and Table 3.2 depicts the use case specification of Manage Program Complexity.

Table 3.1: Use Case Description for Set Weight Value of Metric Use Case

Use case name	Set weight value of metrics		
Primary actor	Lecturer		
Brief description	This use case enables the lecturer to set weight value of selected software metrics that have been adopted in this study in order to measure the complexity of the student's Java programming assignment.		
Pre-condition	None		
Flow of events	Step	Actor Action	System Response
	1.	This use case begins when the lecturer presses on the 'set weight value of metric' button on the main menu.	Menu 'Set weight value of metric' will be displayed.
	2.	The lecturer will set the weight value between range 1 to 5 for the following software metrics: ▪ Lines of Code (LOC) ▪ Number of Classes ▪ Number of Attributes ▪ Number of Methods ▪ Cyclomatic Complexity ▪ Value of Arguments ▪ Value of Attributes ▪ Respond for Class	The system will save the selected value for each software metric in order to use them as a weight of measuring the program complexity.
	3.	This use case end when the lecturer presses OK button.	

Table 3.2: Use Case Description for Manage Program Complexity Use Case

Use case name	Manage program complexity		
Primary actor	Lecturer		
Brief description	This use case enables the lecturer to access student's programming assignment and program schema in order to measure the program complexity and grade the assignment in considering the complexity factor.		
Pre-condition	▪ Student's java programming assignment and program schema must be up loaded into a specific directory on the PC. ▪ Weight values are set.		
Flow of events	Step	Actor Action	System Response
	1.	This use case begins when the lecturer presses the 'check program complexity' button on the main menu.	Menu 'Check program complexity' will be displayed.
	2.	The actor will select the number of classes and press 'OK' button.	Based on the number of classes given, the system will display the list of files that will be downloaded.
	3.	The actor will select and download the following files based on the defined number of classes: a) student's Java file (user-defined class) b) student's Java file (testing class) c) Schema file (user-defined class) d) Schema file (testing class)	

	4.	The actor will press the "Analyze complexity" button.	▪ The system shall measure the complexity value for the program schema and the student's program. ▪ The system shall display the complexity analysis of the program schema and the student's program. ▪ The system shall provide the score for each software metric given and display total mark of the complexity aspect after comparing the complexity value for the program schema with the student 's program.

3.2 Prototype Design

During this phase, sequence diagrams were produced and the complexity checking process, the program complexity measurement, and the program specification were designed. A sequence diagram is a graphical view of the scenario that can be seen as a detailed specification of the use case. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry

out the functionality of the scenario (Quatranit, 2000). The sequence diagram for set weight value of metric use case is depicted in figure 3.2 and figure 3.3 illustrated the sequence diagram for Manage program complexity use case.

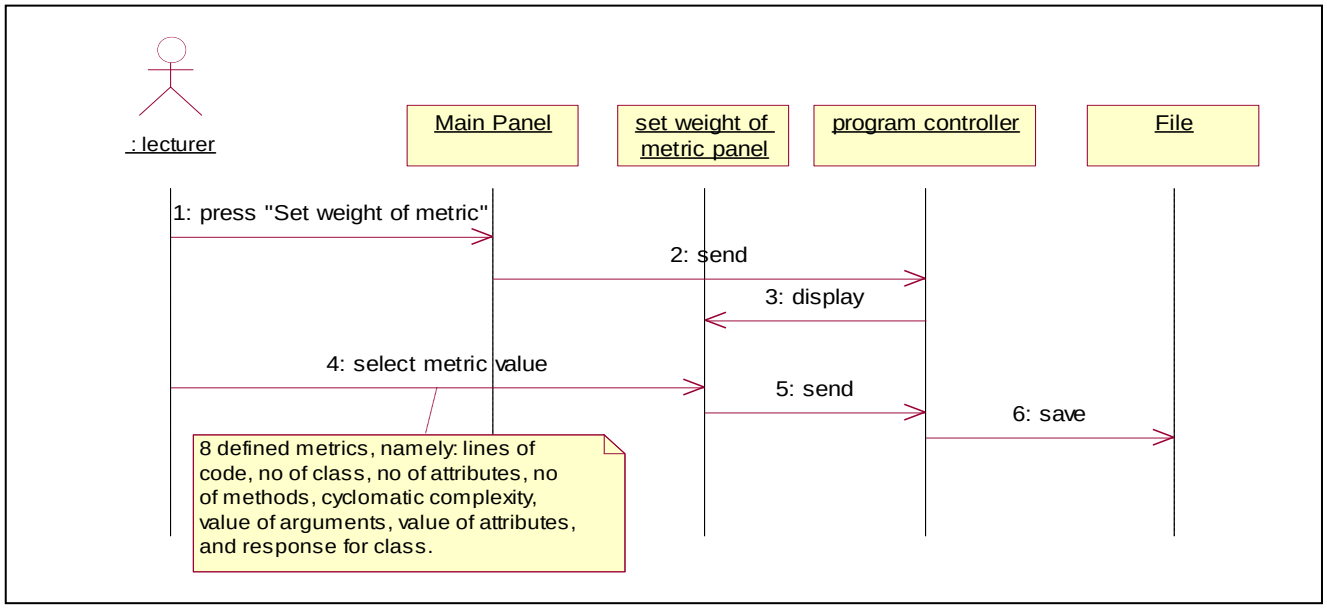


Figure 3.2: Sequence Diagram for Set Weight Value of Metric Use Case

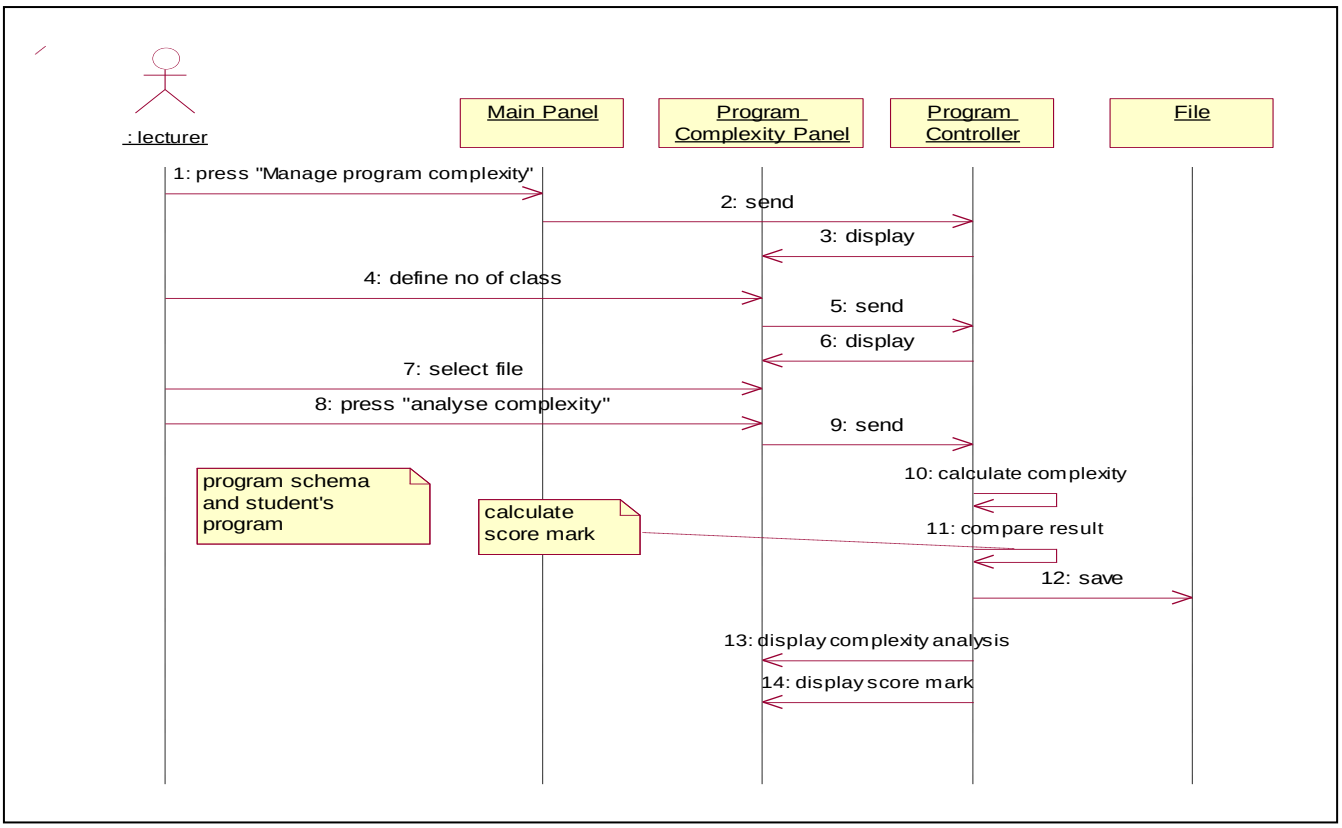


Figure 3.3: Sequence Diagram for Manage Program Complexity Use Case

3.2.1 Program Complexity Measurement Design

In this study, the complexity of a student's program is measured by using selected software metrics (Abounader and Lamb, 1997; Xenos et al, 2000), namely:

i) Number of classes (NCL) metric

- This metric is proposed by Sheetz, Tegarden and Monarchi. NCL metric measures all the number of classes.

ii) Number of properties (SIZE2) metric

- This metric is proposed by Moreau and Dominick. SIZE2 metric counts the number of attributes plus the number of local methods.

iii) Attributes Complexity (AC) metric

- This metric is proposed by Chen and Lu. AC metric defined as $\sum R(i)$, where R(i) is the value of each attribute in the class. Summing all R(i) in the class gives this metric value. The value of each attribute is evaluated based on values in Table 3.3.

Table 3.3: Attribute/Argument Value

Type	Value
Boolean and integer	1
Char	1
Real (Float, double)	2
Array	3 – 4
Object	6 – 9

iv) Cyclomatic Complexity metric

- This metric is proposed by McCabe's. Cyclomatic complexity metric measures the amount of decision logic in a single software module. Cyclomatic complexity is defined to be **e – n + 2**, where *e* and *n* are the number of edges and nodes in the control flow graph, respectively. This cyclomatic complexity is measured for each method in class.

v) Operation Complexity (OpCom) of a class metric

- This metric is proposed by Chen and Lu. The definition for operation complexity is $\sum O(i)$, where O(i) is operation *i*'s complex value. Summing up the O(i) in for each operation *i* in the class gives their metric value. The operation complexity value is evaluated based on values in Table 3.4.

Table 3.4: Operation Complexity Value

Rating	Complexity Value
Null	0
Very Low	1-10
Low	11-20
Nominal	21-40
High	41-60
Very High	61-80
Extra Hight	81-100

vi) Operation Argument Complexity (OAC) metric

- This metric is proposed by Chen and Lu. OAC metric defined as $\sum P(i)$, where P(i) is the value of each argument *i* in each operation in the class. The value of each argument is also evaluated based on values in Table 3.4.

vii) Respond For Class (RFC) metric

- This metric is proposed by Chidamber and Kemerer. RFC metric is the number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class.

viii) Lines Of Code (LOC) metric

- This metric is a traditional software metric. LOC metric measures the size of a module: which is the number of statements including comments (Xenos et. al, 2000).

The process of measuring program complexity is done by implementing a static analysis of the program complexity for a student's program and program schema. Then, the process of analysis and comparison complexity values of student's program and program schema will be done to identify the equivalence of complexity values between both programs. The weight value and score will be given for each selected metric and calculation will be done to define the complexity mark awarded for the student's program. Figure 3.4 depicts an overview of the complexity checking process.

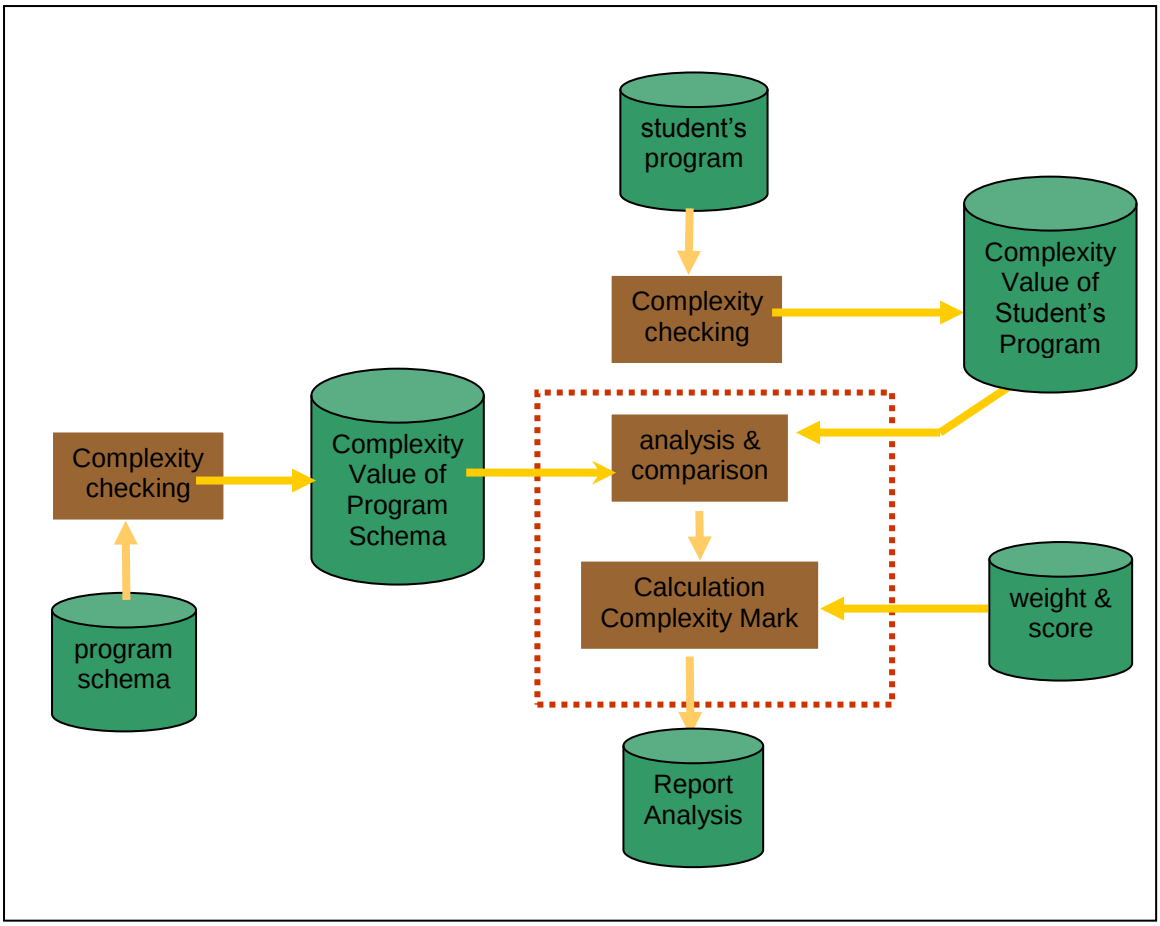


Figure 3.4: Overview of Complexity Checking Process

The measurement of the program complexity was made by assigning appropriate weight and score for each selected software metric. Each metric is given the same score with a value of '10' in order to simplify the process of calculation complexity mark. The weight value for each metric is given on a scale value of 1 to 5. Therefore, the lecturer can choose the specified value according to a level of prioritizing the importance of each metric in the evaluation criteria. The value of '1' is a low priority , whereas the value of '5' is a high priority. The purpose of selecting a scale value of 1 to 5 is to determine less range value of accuracy. The schema of the given scores and weights for each metric for this study is depicted in Table 3.5.

Table 3.5: The Schema of Given Score and Weight for Selected Metric

Selected Metric		Weight value	Score value
i)	NCL	Selection of integer values between 1 to 5	10
ii)	SIZE2		
iii)	AC		
iv)	Cyclomatic Complexity		
v)	OpCom		
vi)	OAC		
vii)	RFC		
viii)	LOC		

The complexity mark of a student's program is calculated by adding the total marks, which are acquired for each metric defined in Table 3.5. The complexity mark is presented in the format of percentage value. The formula for the complexity mark calculation is shown as follows:

$$\text{Complexity Mark} = \frac{\sum_{i=1}^{n} \text{Weight value} \times \text{Score value}}{\sum_{i=1}^{m} \text{Weight value} \times \text{Score value}} \times 100\ \%$$

where,

m = number of selected metric

n = number of metrics that meets schema of output, and $n \leq m$

3.2.2 Program Specification

In order to ensure all the selected software metrics will be measured correctly, there are several program specifications that should be followed as stated below:

- The main method should be declared separately with user-defined classes.
- The program should not contain the blank lines.
- The condition statement should not have more than two conditional operators in single statement.
- The access modifier should be used for all methods and variables in class.
- The instructions of program solving should be defined clearly, in terms of input and output, number of classes, number of attributes, number of methods, control statements that will be used and the arguments used in a method.

3.3 Conclusion

This chapter explained the methodology and how it was used to develop the prototype for the study. This study involved four phases, namely construct a conceptual issue, design the prototype, build the prototype and evaluate the prototype. The UML technique was adopted to analyze the prototype requirement. Besides that, this chapter also discussed prototype design including the complexity checking process, program complexity measurement, and program specification. The next chapter will discuss on the prototype development and testing phase.

CHAPTER FOUR

DEVELOPMENT AND TESTING

This chapter discusses on prototype development and testing phase. The development section will discuss on the related topics of prototype development meanwhile, the testing section will discuss on the prototype testing including an approach for testing used in this study and the static analysis of complexity result.

4.1 Prototype Development

In the implementation phase, a prototype was developed to automate the process of measuring the complexity of a student's program. The Java complexity measurement prototype developed is referred to as **JCoM** (Java Complexity Measurement). During this phase, the system architecture defined in the design phase is transformed into codes using selected software. The software tools are depicted in Table 4.1.

Table 4.1: Software Tools

Type of software	Purpose
Kawa version 3.22	Editor for prototype development
Jdk 1.3	Java Compiler
NotePad	Text File

4.1.1 Description of JCoM Prototype

JCoM is developed to provide an environment to assist lecturers of *Introduction to Programming* course to automate the process of measuring the complexity of a student's Java programming assignments. Besides providing lecturers with an environment to implement the complexity checking of student's programs, **JCoM** also provides another two functions, namely:

- Calculate the complexity mark of a student's program
- Produce comments of an analysis complexity that has been generated.

JCoM is also developed as a support tool to improve the process of evaluating correctness of the Java programming assignments proposed by Rohaida et. al (2004). The lecturer who is the main user of JCoM should assign a weight value for each selected metric before checking the complexity of a student's program. These values will be stored in the text file and which will be used in the process of measuring the program complexity. There are three sub processes, which will be done sequentially during the implementation of measuring program complexity, namely:

- Checking the complexity of student's program and program schema. The produced complexity values for both programs will be stored into a text file.
- Analysis and comparison of complexity values of student's program and program schema are implemented by using the complexity values produced in the previous process.
- Calculation of complexity mark for student's program is done by using weight value and score that have been defined.

A report that consists of the complexity analysis and the complexity mark of student's program will be generated as a final output for this prototype.

4.1.2 JCoM Interfaces

Interfaces of **JCoM** prototype are produced based on the use cases that have been defined at the initial phase. As mentioned in section 3.1, there are two use cases defined for this study, namely:

- Set Weight Value of Metrics
- Manage Program Complexity

There are two main interfaces in **JCoM** prototype, namely:

a. Set Weight Value of Metric Interface

The interface of Set Weight Value of Metric is depicted in Figure 4.1.

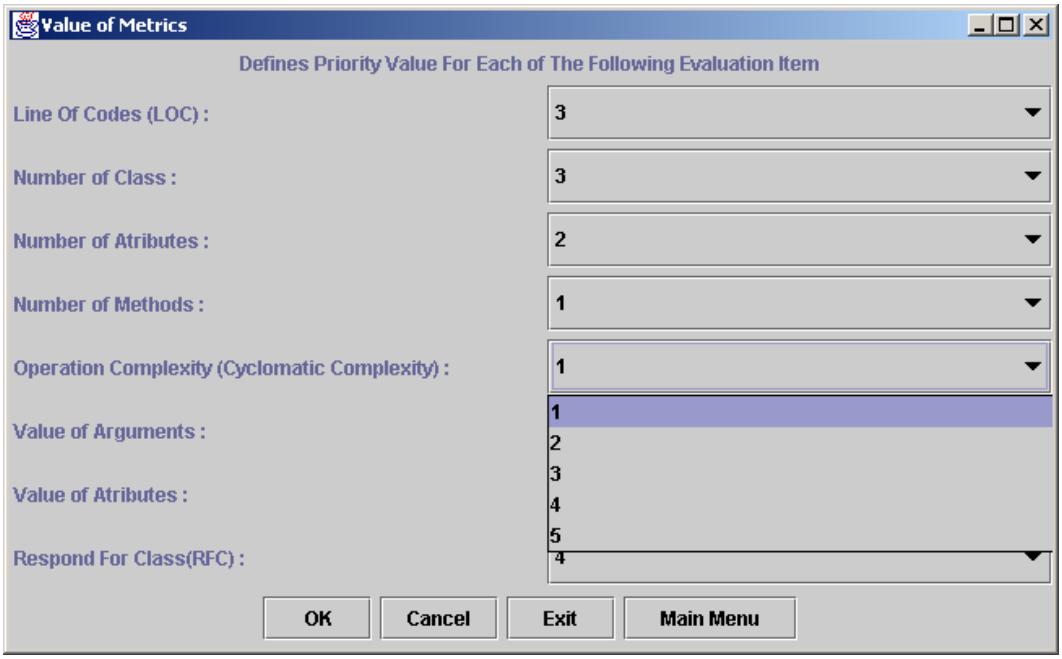


Figure 4.1: Set Weight Value of Metric Interface

Set Weight Value of Metric interface is used by a lecturer to assign a weight value for each metric that is listed in the interface. The value given is based on the level of the importance of each metric in the evaluation criteria. The weight value is given in a scale value from 1 to 5. The selection of weight value can be done by choosing one of the values that are listed in the combo box. All the weight values will be stored into the text file after the user presses the 'OK' button. These weight values will be used in the process of calculating the complexity mark for a student's program.

b. Manage Program Complexity Interface

The interface of Manage Program Complexity is depicted in Figure 4.2.

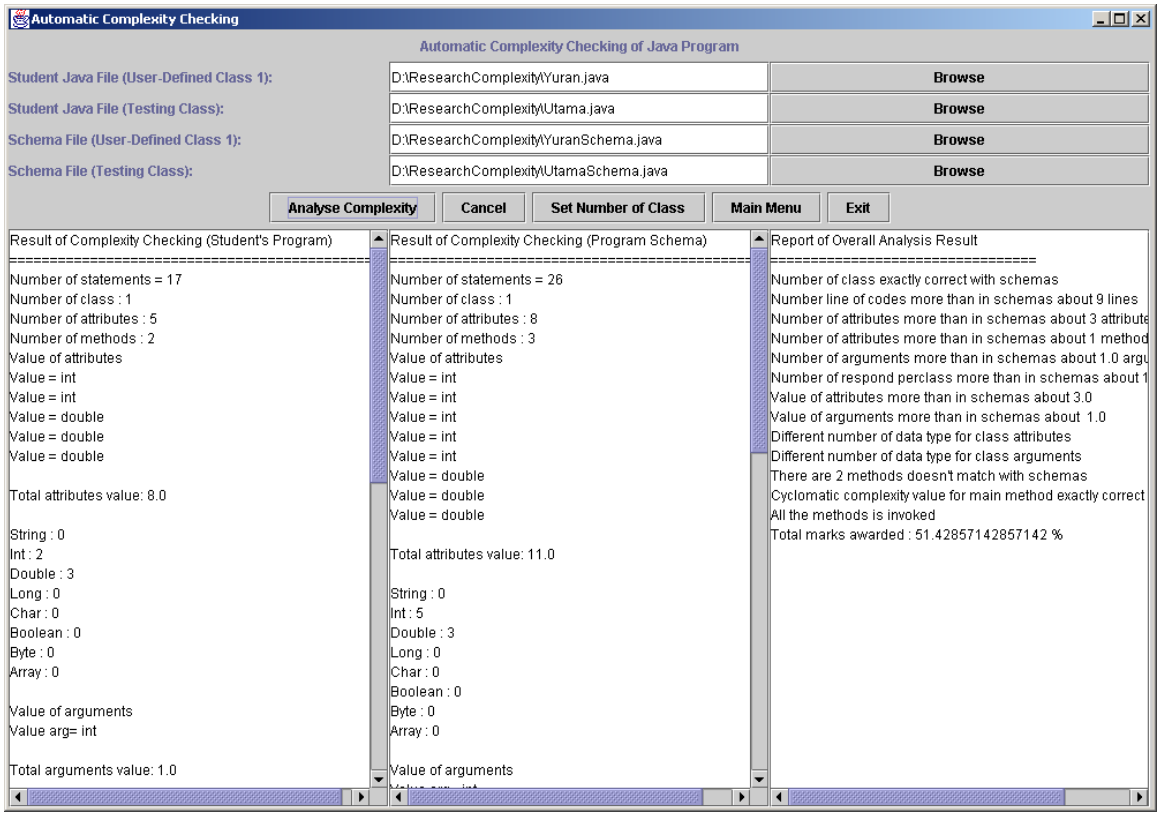


Figure 4.2: Manage Program Complexity Interface

The Manage Program Complexity interface is used to implement the process of the static analysis of program complexity for both student's program and program schema. The lecturer will select and download the student's program and program schema that consist of user-defined class and testing class. The Testing class is referred as a main class. Therefore, the user should define the number of user-defined class involved in the program schema before the process of static analysis of the program complexity can be implemented. The Set Number of Class interface is used to insert the number of user-defined class needed in the solution of program. The maximum number of user-defined class for this prototype was limited to three classes. When a user presses the 'OK' button, the Manage Program Complexity interface will be displayed. The interface of Set Number of Class is depicted in Figure 4.3.

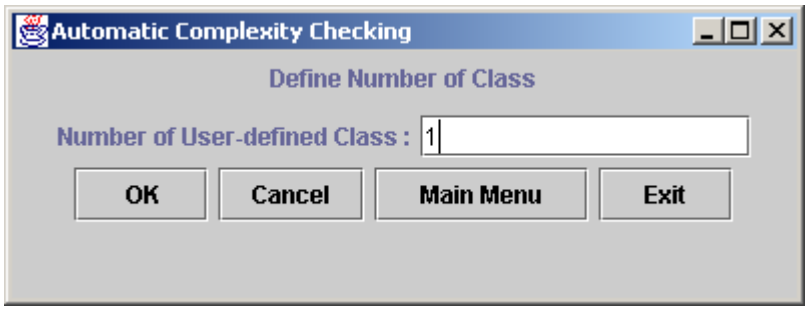


Figure 4.3: Set Number of Class Interface

Referring to the Manage Program Complexity interface, the files of the student's program and program schema will be accessed from the current directory by pressing the 'Browse' button. Right after pressing this button, the interface of open current files will be displayed. This interface is depicted in Figure 4.4.

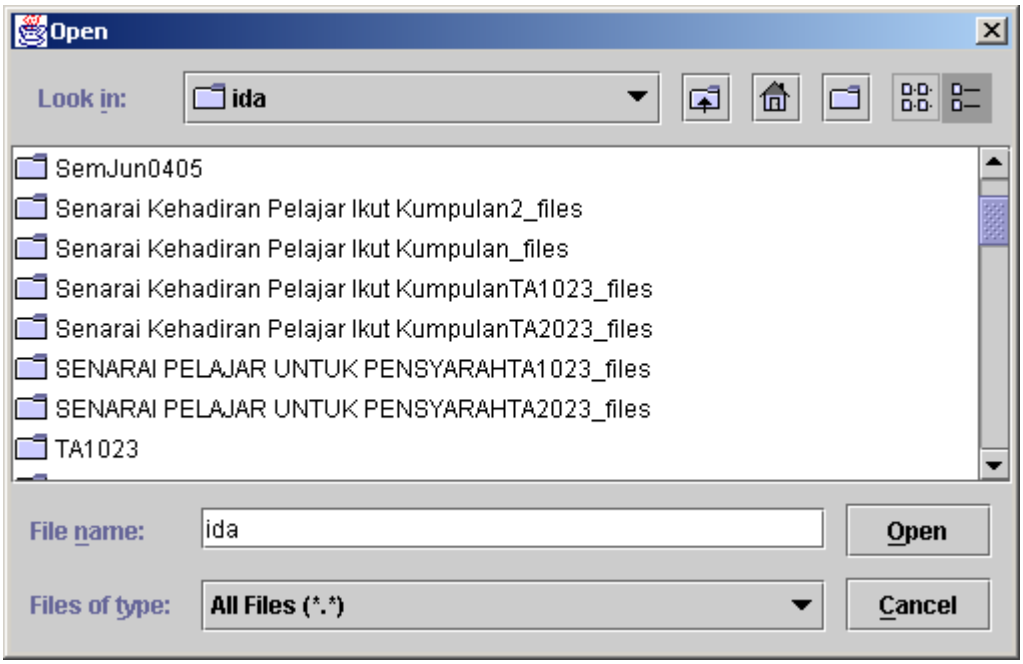


Figure 4.4: Open File Interface

All the classes involved in the solution of a program assignment should be accessed from current directories in order to implement the process of static analysis of the program complexity. The complexity analysis for the student's program and program schema will be stored into different text files. These complexity values will be used in the process of analysis and comparison between the student's program and the program schema in order to determine the equivalence of their complexity values. This process can be done by pressing the 'Analysis complexity' button. The results of the complexity checking for both student's program and program schema also will be displayed in the text areas that are contained in the interface. Furthermore, the details of the analysis complexity and complexity mark for student's program will be also displayed in the text area on this interface. The analysis report of the complexity checking is presented in the format of listing of comments.

4.2 Prototype Testing

After the implementation phase, the prototype will be tested in order to validate how well the prototype performs. The test conducted focused on the requirements that have been defined during the analysis phase. In conducting the testing process, the following hardware and software must be prepared:

a. Hardware preparation

The testing will be executed using desktop computers in windows operating system environment.

b. Software preparation

JCoM prototype is used during the testing phase. The description of required software is depicted in Table 4.2.

Table 4.2: Description of Required Software

No.	Item	Description
1.	JDK 1.3	To compile Java program
2.	Notepad	Files

4.2.1 Testing Approach

JCoM is used to examine the Java program source code without executing the program. As mentioned in the previous chapter, static analysis is selected as a testing approach for assessment of program complexity. This approach involves a process of examining a program without executing it. In this study, the testing activity is conducted based on the defined use cases. Therefore, a testing script is used to describe all the necessary steps to conduct a particular test.

The testing script for Set Weight Value of Metrics use case is depicted in Table 4.3 and Table 4.4 depicts the testing script for the Manage Program Complexity use case.

Table 4.3: Testing Script for Set Weight Value of Metrics Use Case

No.	**Use Case**	**Description**
1	**Set weight value of metrics**	This use case enables the lecturer to set the weight of the selected software metrics that has been adopted in this study in order to measure the complexity of the student's Java programming assignment.
Pre-conditions		None
Post-conditions		Weight values of selected software metrics have been defined.

Test Steps	**Test Scenarios**	**Expected Output**
1.	The lecturer presses on 'Set weight value of metrics' button in main menu.	Menu 'Set weight of metric' will be displayed.
2.	The lecturer will set the weight value between the range 1 to 5 for the following software metrics: ▪ Lines of Code (LOC) ▪ Number of Classes ▪ Number of Attributes ▪ Number of Methods ▪ Cyclomatic Complexity ▪ Value of Arguments ▪ Value of Attributes ▪ Respond for Class	
3.	Press on 'OK' button.	

Table 4.4: Testing Script for Manage Program Complexity Use Case

No.	**Use Case**	**Description**
2	**Manage Program Complexity**	This use case enables the lecturer to access student's programming assignment and program schema in order to measure the program complexity and grade the assignment in considering the complexity factor.
Pre-conditions		Student's Java programming assignment and program schema must be up loaded into specific directory on PC. Weight values are set.
Post-conditions		The expected output consists of complexity analysis and total mark will be displayed.

Test Steps	**Test Scenarios**	**Expected Output**
1.	The lecturer presses 'check program complexity' button on the main menu.	Menu 'Check program complexity' will be displayed.
2.	Select number of classes and press 'OK' button.	Based on number of classes given, the system will display the list of files that will be downloaded.
3.	Select and download the following files based on the defined number of classes: ▪ student's Java file (user-defined class) ▪ student's Java file (testing	

	class) ▪ Schema file (user-defined class) ▪ Schema file (testing class)	
4.	Press on the "Analyze complexity" button.	▪ The complexity analysis for program schema and student's program will be displayed separately in text field. ▪ Report of overall complexity analysis result will be displayed. ▪ The total mark for the complexity aspect will be displayed.

4.4.2 Static Analysis of Complexity Result

As mentioned in section 3.4.2, static analysis is used as a testing approach and the results are based on the selected software metrics. JCoM is used to measure the complexity of the program schema and student's program. In this study, two sets of programs have been chosen in order to figure out the complexity result as follows:

a) Set 1 consists of a student's program that is exactly similar with the program schema. Figure 4.5 and figure 4.6 depicts the program schema and the student's program respectively and both programs consist of user defined classes and testing classes.

b) Set 2 consists of a student's program that differs with the program schema. Figure 4.7 depicts the program schema and figure 4.8 depicts the student's program and both programs consist of the user defined class and the testing class.

The reason for choosing only two sets of program to be tested is because the number of similar or dissimilar programs with the program schema does not affect the process of static analysis and calculating the complexity mark, even though the result produced is different.

```
Import java.io.*;
public class UtamaSchema
{
  public static void main(String arg[]) throws IOException
  {
    int count=1;
    int bilSubject=0;
    String name;
    InputStreamReader read = new InputStreamReader (System.in);
    BufferedReader input = new BufferedReader(read);
    YuranSchema subject = new YuranSchema();
    System.out.println("**********SCORE A1 TUITION CENTRE*************");

    while(count==1)
    {
      System.out.print("Name:");
      name= input.readLine();

      System.out.print("Number of subject:");
      bilSubject = Integer.parseInt(input.readLine());
      subject.calculateFee1(bilSubject);
      subject.calculateFee2(bilSubject);
      System.out.print("Do you want to continue (press 1):");
      count= Integer.parseInt(input.readLine());
      System.out.println();
    }
    subject.printFee();
    }
}

public class YuranSchema
{
  private int bil, count=1, c, b, n ;
  private double registrationFee=100.00,fee=0.0, totalFee=0.0;
  public void calculateFee1(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *30.00);
    else
          fee = registrationFee +((bil * 30.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
  }
  public void calculateFee2(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *25.00);
    else
          fee = registrationFee +((bil * 25.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
  }
  public void printFee(){
      System.out.println("Total Fee: RM" + totalFee);
  }
 }
```

Figure 4.5: Program Schema

```
import java.io.*;
public class Utama
{
  public static void main(String arg[]) throws IOException
  {
    int count=1;
    int bilSubject=0;
    String name;
    InputStreamReader read = new InputStreamReader (System.in);
    BufferedReader input = new BufferedReader(read);
    YuranSchema subject = new YuranSchema();
    System.out.println("**********SCORE A1 TUITION CENTRE*************");

    while(count==1)
    {
      System.out.print("Name:");
      name= input.readLine();

      System.out.print("Number of subject:");
      bilSubject = Integer.parseInt(input.readLine());
      subject.calculateFee1(bilSubject);
      subject.calculateFee2(bilSubject);
      System.out.print("Do you want to continue (press 1):");
      count= Integer.parseInt(input.readLine());
      System.out.println();
    }
    subject.printFee();
    }
}

public class Yuran
{
  private int bil, count=1, c, b, n ;
  private double registrationFee=100.00,fee=0.0, totalFee=0.0;
  public void calculateFee1(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *30.00);
    else
          fee = registrationFee +((bil * 30.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
  }
  public void calculateFee2(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *25.00);
    else
          fee = registrationFee +((bil * 25.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
  }
  public void printFee(){
      System.out.println("Total Fee: RM" + totalFee);
  }
 }
```

Figure 4.6: Student's Program

In the first set, both programs were examined and Table 4.5 depicts the expected result, which consists of the complexity analysis and the rewarded mark. The rewarded mark is based on the defined weight value of software metrics, which is depicted in Table 4.6.

Table 4.5: Expected Result of Similar Programs

Item	Selected Metrics	Expected Result
Complexity Analysis (Program Schema)	▪ Lines Of Code ▪ Number of classes ▪ Number of properties ▪ Attributes Complexity ▪ Operation Arguments Complexity ▪ Cyclomatic Complexity ▪ Operation Complexity of classes ▪ Cyclomatic Complexity for testing class ▪ Response for Class	26 1 11 11 2 5 very low 2 3
Complexity Analysis (Student's program)	▪ Lines Of Code ▪ Number of classes ▪ Number of properties ▪ Attributes Complexity ▪ Operation Arguments Complexity ▪ Cyclomatic Complexity ▪ Operation Complexity of classes ▪ Cyclomatic Complexity for testing class ▪ Response for Class	26 1 11 11 2 5 very low 2 3
Total Mark	▪ Program Schema ▪ Student's program	100% 100%

Table 4.6: Weight Value of Software Metrics

Software metrics	Weight Value
Line Of Codes	2
Number of Class	4
Number of Attributes	3
Number of Methods	4
Operation Complexity (Cyclomatic Complexity)	4
Value of Arguments	3
Value of Attributes	4
Respond for class (RFC)	4

```
import java.io.*;
public class UtamaSchema
{
  public static void main(String arg[]) throws IOException
  {
    int count=1;
    int bilSubject=0;
    String name;
    InputStreamReader read = new InputStreamReader (System.in);
    BufferedReader input = new BufferedReader(read);
    YuranSchema subject = new YuranSchema();
    System.out.println("**********SCORE A1 TUITION CENTRE*************");

    while(count==1)
    {
      System.out.print("Name:");
      name= input.readLine();

      System.out.print("Number of subject:");
      bilSubject = Integer.parseInt(input.readLine());
      subject.calculateFee1(bilSubject);
      subject.calculateFee2(bilSubject);
      System.out.print("Do you want to continue (press 1):");
      count= Integer.parseInt(input.readLine());
      System.out.println();
    }
    subject.printFee();
    }
}
public class YuranSchema
{
  private int bil, count=1, c, b, n ;
  private double registrationFee=100.00,fee=0.0, totalFee=0.0;
  public void calculateFee1(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *30.00);
    else
          fee = registrationFee +((bil * 30.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
  }
  public void calculateFee2(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *25.00);
    else
          fee = registrationFee +((bil * 25.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
  }

public void printFee(){
      System.out.println("Total Fee: RM" + totalFee);
```

```
   }
 }
```

Figure 4.7: Program Schema

```
import java.io.*;
public class Utama
{
  public static void main(String arg[]) throws IOException
  {
    int count=1, bilSubject;
    String name;
    InputStreamReader read = new InputStreamReader (System.in);
    BufferedReader input = new BufferedReader(read);
    Yuran subject = new Yuran();
    System.out.println("**********SCORE A1 TUITION CENTRE*************");

    while(count==1)
    {
      System.out.print("Name:");
      name= input.readLine();

      System.out.print("Number of subject:");
      bilSubject = Integer.parseInt(input.readLine());
      subject.calculateFee(bilSubject);
      System.out.print("Do you want to continue (press 1):");
      count= Integer.parseInt(input.readLine());
      System.out.println();
    }
    subject.printFee();
    }
}
public class Yuran
{
  private int bil, count=1 ;
  private double registrationFee=100.00,fee=0.0, totalFee=0.0;
  public void calculateFee(int bilSubject){
    bil = bilSubject;
    if (bil <3)
          fee= registrationFee + (bil *30.00);
    else
          fee = registrationFee +((bil * 30.00) *0.85);
    System.out.println("Fee: RM" + fee);
    totalFee +=fee;
    }
  public void printFee(){
      System.out.println("Total Fee: RM" + totalFee);
    }
 }
```

Figure 4.8: Student's Program

In the same manner, for the 2nd set, both programs were examined by using JCoM and the expected results consisted of the complexity analysis and rewarded marks were produced and depicted in Table 4.7.

Table 4.7: Expected Result of Different Programs

Item	Selected Metrics	Expected Result
Complexity Analysis (Program Schema)	▪ Lines Of Code	26
	▪ Number of classes	1
	▪ Number of properties	11
	▪ Attributes Complexity	11
	▪ Operation Arguments Complexity	2
	▪ Cyclomatic Complexity	5
	▪ Operation Complexity of classes	very low
	▪ Cyclomatic Complexity for testing class	2
	▪ Response for Class	3
Complexity Analysis (Student's program)	▪ Lines Of Code	17
	▪ Number of classes	1
	▪ Number of properties	7
	▪ Attributes Complexity	8
	▪ Operation Arguments Complexity	1
	▪ Cyclomatic Complexity	3

	▪ Operation Complexity of classes ▪ Cyclomatic Complexity for testing class ▪ Response for Class	very low 2 2
Total Mark	▪ Program Schema ▪ Student's program	100% 51.43%

4.5 Testing Results and Conclusion

Based on the implementation and prototype testing of **JCoM**, we have found that the prototype is able to implement the following processes automatically:

- The static analysis of the program complexity for the tested program and the program schema.
- The Process of analysis and comparison complexity values of the tested program and program schema.
- The Complexity mark awarded for the tested program.
- The report analysis that consists of the complexity analysis and the complexity mark of the tested program.

Due to **JCoM**'s tested and proven ability, it shows that the prototype developed has met the requirements for this study.

CHAPTER FIVE

CONCLUSION

This chapter explains the findings of this study. It also includes suggestions and recommendations for future work.

5.1 Result Findings

This study is focused on the automation of measuring the complexity of Java programming assignment in terms of maintaining a uniform marking standard for the *Introduction To Programming* course. Results of this study shows that, the prototype developed referred as **JCoM** is able to automate the process of measuring the complexity of student's Java programming assignment. However, the program specification defined in section 3.2.2 should be followed to ensure all the selected metrics are measured correctly.

Furthermore, the selection of object-oriented metrics which are used to measure the complexity of a student's Java programming assignment for this study are mostly covered by the basis evaluation items in the current manual assessment of the student's Java source code. This is indicated in the results of preliminary study that shown in Appendix B.

This study is also provides an environment in the prototype to allow the lecturer to select the appropriate scale of the weight value used to measure the complexity of student's program. It gives a choice to the lecturer in order to prioritize the weight value based on the importance of selected metric in the evaluation criteria.

Meanwhile, the result analysis produced as a final output for this prototype can provide an information guideline to the students in terms of identifying whether or not their program followed the program requirement. However, there are a few limitations to this study. These are:

- This study focused on only one of the maintainability quality factor, which is the complexity.
- Selected object-oriented metrics used to measure program complexity did not cover evaluation items for advanced object-oriented programming.
- The number of user-defined classes was limited for three classes only.

5.2 Future Works

The following are several recommendations for future work due to the limitation described in section 5.1:

- There is another factor of maintainability, which is typographic arrangement that describes the way a program source code is presented and provides full measuring of programming style of program source code.
- Others advanced object-oriented metrics such as class cohesion, coupling between objects, class coupling, depth of inheritance tree, method inheritance factor and polymorphism factor can be used to measure the program complexity of advanced object-oriented programming.
- The user (lecturers) should have an authority to determine the distinct number of user-defined class involved in the programming solution.

5.3 Conclusion

As a conclusion, the prototype developed is an initial effort to automate the process of measuring the complexity of students' Java programming assignments. Even though, this prototype does not fully measure the complexity of advanced object-oriented programming, the selected software metrics that has been adopted in this study mostly covers the basic evaluation item in marking the "Introduction To Java Programming" assignments. Furthermore, based on **JCoM**'s ability finding, it can improve consistency and time in the marking process, in terms of the complexity measurement.

REFERENCES

Abounader, J. R. & Lamb, D.A. 1997. A Data Model for Object-Oriented Design Metrics. Retrieved May 26, 2005. http://citeseer.ist.psu.edu/abounader97data.html

Basili, V.R. (1980). Qualitative Software Complexity Models: A summary in Tutorial on Models and Methods for Software Management and Engineering. *IEEE Computer Society Press*.

Booch, G., Jacobson, I. & Rumbaugh, J. (1998*). The Unified Software Development Process.* Massachusetts. Addison Wesley.

Chen, J.Y. & Lu, J.F. (1993). A new metrics for Object-oriented design. *Information and Software Technology.* 35 (4), 232-240.

Chidamber, S.R. & Kemerer, C.F. (1995). A Metrics suite for Object-Oriented design. *IEEE Transactions on Software Engineering*..21(3), 265.

Conte, S.D., Dunsmore, H.E. & Shen, V.Y. (1986). *Software Engineering Metrics and Models.* California :Benjamin /Cummings Publisher

Coward, P.D. (1988). A review of software testing. *Information and Software Technology.* Vol 30, 187-197

Curtis, B. (1985). Tutorial: Human Factors in Software Development. *IEEE Computer Society*.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A. and Love, T. Measuring the psychological complexity of Software Maintenance Tasks with the Healstead and McCabe metrics. *IEEE Trans. Software Engineering*, SE-5, 2, 96-104.

Foxley, E. Higgins, C, & Gibbon, C. (1996). The Ceild System. Retrieved May 26, 2005. http://www.cs.nott.ac.uk/CourseMarker/more_info/html/Overview96.htm

George, J.F, Batra, D. & Valacich, J.S. & Hoffer, J.A. (2004*). Object-oriented Systems Analysis and Design.* New Jersey: Prentice Hall

Hung, S., Kwok, L. & Chan, R. (1993). Automatic Program Assessment . *Computers and Education, Computer and Education*,Vol 20, 183-190

Jackson, D. (1996). A Software System for Grading Student Computers Programs. *Computer Education*. 27(3/4), 171-180.

Lake A. and Cook C., (1994). Use of Factor Analysis to Develop OOP Software Complexity Metrics, *Proceeding of the Sixth Annual Oregon Workshop Software Metrics.*

Li, W., Henry, S., Kafura, D. & Schulman, R. (1995). Measuring Object-Oriented Design. *Journal of Object-Oriented Programming*. 48-55.

McCabe, T.J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, 2, 308-320.

Mengel, S.A. & Yerramilli V. (1999). A Case Study of Static Analysis of the Quality of Novice Student Programs. *The proceddings of the 13 SIGCSE Technical Symposium on Computer Science Education*, Vol 31, 1

Moreau, D.R. & Dominick, W.D. (1989). Object-oriented graphical Information Systems: Research plan and Evaluation Metrics. *Journal of System and Software.* Vol 10, 23-28.

NunaMaker, J., Chen, M. & Purdin, T. (1991). System Development In Information Systems Research. *Journal of Information Systems,* 7(3), 89-106

Quatrani, T (2000). *Visual Modeling with Rational Rose 2000 and UML*. Canada :Addison Wesley

Ramamoorthy, C.V., Tsai W.T, Yamaura T. & Bhide A. (1985). Metric Guided Methodology. *IEEE Trans. On Software Engineering*. Vol SE-11 No.5.

Rohaida, R., Cik Fazilah, H. & Mazni, O. (2004). *Correctness Assessment Of Java Programming Assignment.* Laporan Akhir Penyelidikan Geran Fakulti. Universiti Utara Malaysia.

Rosenberg, L.H. & Hyatt, L.E. (1997). Software Quality Metrics for Object-Oriented Enviroment. Retrieved Jun, 20, 2005, from http://satc.qsfc.nasa.gov/support/CROSS_APR97/oocross.pdf

Rosenberg, L.H. (1998). Applying and Interprenting Object-Oriented Metrics. Retrieved Jun, 2, 2005, from http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html

Sommerville, I. (2004). *Software Engineering. 7th Edition*. England: Addison Wesley

Tegarden, D.P., Sheetz, S.D. & Monarchi, D.E. (1992). Effectiveness of Traditional Software Metrics for Object-oriented Systems. *Proceedings of the Twenty-fifth Hawaii International Conference.* Vol 4, 359-368.

Truong, N., Roe, P. & Bancroft, P. (2004). Static Analysis of Student's Java Programs. *Conference in Research and Practice in Information Technology*, Vol 30

Vizcaino A., Contreras J., Favela J. and Prieto M., (2000). An Adaptive, Collaborative Environment to Develop Good Habits in Programming, *Proceeding of International Conference on Intelligent Tutoring System.*

Watson A. H. and McCabe T. J. (1996). Structural Testing: A Testing Methodology Using Cyclometic Complexity Metrics. *Technical Report of NIST Special Publication* 500-235.

Xenos M., Starrinoudis D., Zikouli K. and Christtodoulakis D. (2000). Object-Oriented Metrics – A Survey. Retrieved May 10, 2005, from http://citeseer.ist.psu.edu/528212.html

Yourdan, E. & Constantine, L. (1979). *Structured Design – Fundamentals of a discipline of computer programs and design.* Prentice Hall

Zarina, S. (1999). The Automatic Assessment of Z Specification. PhD Thesis. University Nottigham.

Zin, A.M. & Foxley, E. (1994). Automatic Program Assessment System. Retrieved May, 20, 2005, from http://www.cs.nott.ac.uk/CouseMarker/more_info/html/ASQA.htm

Zuse, H. (1991). *Software Complexity, Measures and Methods.* Walter de Gruyter. New York.

APPENDIX A

A PRELIMINARY STUDY

Universiti Utara Malaysia
Fakulti Teknologi Maklumat

7 June 2005

Sir/Madam

We are currently working on a study to automate the marking schema of the measuring complexity for Java programming assignment. This preliminary study is meant to gauge the extend to which those who are teaching the programming course, especially for Pengaturcaraan Awalan (TA1013) and/or Pengaturcaraan Lanjutan (TA1023) evaluate the program source code in marking the Java programming assignment.

We would really appreciate it if you could spare a moment of your time to help us fill in the form. Your cooperation is highly appreciated. Thank you.

Sincerely yours,

Mawarny Md. Rejab
Rohaida Romli

PRELIMINARY STUDY OF MARKING SCHEMA FOR JAVA PROGRAMMING ASSIGNMENT

INSTRUCTION : Please tick √ the appropriate answer for each of the following:

SECTION A : TEACHING BACKGROUND

1. Teaching experience in Java programming.

   ☐ 1 semester
   ☐ 2 semester
   ☐ 3 semester
   ☐ 4 semester
   ☐ more than 4 semester, please indicate _____.

2. Java programming courses that have been taught.

   ☐ TA1013 (Pengaturcaraan Awalan)
   ☐ TA1023 (Pengaturcaraan Lanjutan)
   ☐ TA2023 (Struktur Data dan Analisis Algoritma)
   ☐ Others, please indicate _____

SECTION B : MARKING SCHEMA OF JAVA PROGRAMMING ASSIGNMENT

1. Evaluation Items

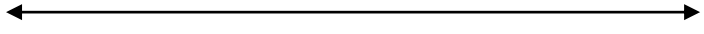
		YES	NO
1.1	Besides evaluating the program output, the program source code is also considered in marking the student's Java programming assignment. *(If not, please proceed to item no. 3)*	☐	☐
1.2	Evaluation items for program source code include:		
	a) Lines of code	☐	☐
	b) Number of classes	☐	☐
	c) Number of variables/attributes	☐	☐
	d) Data types of variables/attributes	☐	☐
	e) Number of methods	☐	☐
	f) Number of arguments in each method	☐	☐
	g) Data types of arguments in each method	☐	☐
	h) Control flow statements in each method	☐	☐
	i) Declared method which is not invoked by any object	☐	☐
	j) Class coupling	☐	☐
	k) Cohesion	☐	☐
	l) Depth of inheritance	☐	☐
	m) If needed, please state others appropriate evaluation items which is not mentioned above.		

_____
_____
_____
_____

2. Marking schema for evaluation items

2.1 Using the scale provided, please prioritize each evaluation item.

Low priority moderate High priority



1 2 3 4 5

	1	2	3	4	5
a) Lines of code	☐	☐	☐	☐	☐
b) Number of classes	☐	☐	☐	☐	☐
c) Number of variables/attributes	☐	☐	☐	☐	☐
d) Data types of variables/attributes	☐	☐	☐	☐	☐
e) Number of methods	☐	☐	☐	☐	☐
f) Number of arguments in each method	☐	☐	☐	☐	☐
g) Data types of arguments in each method	☐	☐	☐	☐	☐
h) Control flow statements in each method	☐	☐	☐	☐	☐
i) Declared method which is not invoked by any object	☐	☐	☐	☐	☐
j) Class coupling	☐	☐	☐	☐	☐
k) Cohesion	☐	☐	☐	☐	☐
l) Depth of inheritance	☐	☐	☐	☐	☐
m) Recommended evaluation items stated in Section B (1.2)	☐	☐	☐	☐	☐
_____	☐	☐	☐	☐	☐
_____	☐	☐	☐	☐	☐
_____	☐	☐	☐	☐	☐
_____	☐	☐	☐	☐	☐

2.2 Recommendation percentage for the program source code that should be contributed for the overall java programming assignment mark.

.

☐ 10%
☐ 20%
☐ 30%
☐ 40%
☐ 50%
☐ Other, please indicate _____

3. Other suggestions/comments

_____
_____
_____
_____
_____
_____

APPENDIX B:

RESULT ANALYSIS OF PRELIMINARY STUDY

Analysis Result for Preliminary Study in term of Marking Schema of Java Programming Assignment

1. Considering of program source code as an evaluation item

Test Item	Number of respondent
a. Yes	10
b. No	0

2. Current evaluation items for program source code

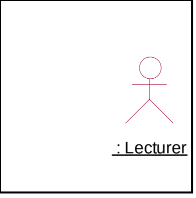
Test Item	Number of respondent	
	Yes	No
a. LOC	1	9
b. NOC	9	1
c. Number of variables/attributes	5	5
d. Data types of variables/attributes	9	1
e. Number of methods	9	1
f. Number of arguments in each method	4	6
g. Data type of arguments in each method	9	1
h. Control flow statement in each method	8	2
i. Declared method which is invoke by any object	5	5
j. Class coupling	4	6
k. Cohesion	3	7
l. Depth of inheritance	3	7
m. Others (programming style)	1	9

3. Suggestion of prioritizing an evaluation items

Test Item	Number of respondent				
	Scale				
	1	2	3	4	5
a. LOC	2	4	1	0	0
b. NOC	1	1	3	3	2
c. Number of variables/attributes	1	2	3	2	1
d. Data types of variables/attributes	0	0	2	4	3
e. Number of methods	0	2	2	3	2
f. Number of arguments in each method	1	3	2	3	1
g. Data type of arguments in each method	0	0	3	5	0
h. Control flow statement in each method	0	0	2	4	2
i. Declared method which is invoke by any object	0	3	1	0	2
j. Class coupling	1	2	2	3	0
k. Cohesion	0	0	1	2	0
l. Depth of inheritance	0	0	1	2	0

4. **Recommendation percentage mark (percentage contribute to total mark)**

Test Item	Number of respondent
a. 10 %	0
b. 20 %	1
c. 30 %	5
d. 40 %	1
e. 50 %	1
f. others	2


: Lecturer