Towards a Unified Source Code Measurement Framework Supporting Multiple Programming Languages

Reisha Humaira, Kazunori Sakamoto, Akira Ohashi, Hironori Washizaki, Yoshiaki Fukazawa
Dept. Computer Science and Engineering
Waseda University
Tokyo, Japan
(reisha@fuji, kazuu@ruri, akira-radiant@akane).waseda.jp / (washizaki, fukazawa)@waseda.jp

Abstract—**Software metrics measure various attributes of a piece of software and are becoming essential for a variety of purposes, including software quality evaluation. One type of measurement is based on source code evaluation. Many tools have been developed to perform source code analysis or to measure various metrics, but most use different metrics definitions, leading to inconsistencies in measurement results. The metrics measured by these tools also vary by programming language. We propose a unified framework for measuring source code that supports multiple programming languages. In this paper, we present commonalities of measurable elements from various programming languages as the foundation for developing the framework. We then describe the approach used within the framework and also its preliminary development. We believe that our approach can solve the problems with existing measurement tools.**

Keywords-source code measurement; metrics; framework; multiple languages

I. Introduction

Quantitative measurements have become essential in software development for a variety of purposes, especially the evaluation of software quality. Software metrics as a measure of properties of a piece of software are commonly used by software developers and are based on software processes, products, or resources [1]. There are various kinds of measurements that can be conducted based on the goal of the metric.

One type of measurement is performed based on source code. There are already a large number of source code measurement tools, but most of them are only able to handle one programming language [2], [3], [4], [5], [6]. Some, such as [7] and [8], are able to handle multiple programming languages, but we still need to specify which programming language should be measured, which means these tools cannot handle software written in more than one programming language. In addition, some measurement tools cover only one metric category. For example, [9] and [10] only assess size by measuring lines of code, so if we want to measure complexity we have to find another measurement tool. When different tools are used together, there can be inconsistency in measurement results due to differences in measurement definitions and standards among the tools [11].

It would be worthwhile to have a unified source code measurement system for multiple programming languages. However, the implementation would not be easy since processing each programming language and metric would entail significant expense. In this paper, we present the preliminary efforts related to our proposed measurement framework. We extract the commonalities of various elements needed to measure source code metrics regarding the features of a particular programming language. The results are the foundation for developing a lower-cost framework, and it will be easier to define or modify the measurement for use with multiple programming languages.

The remainder of this paper is organized as follows. Section 2 discusses problems with existing measurement tools, and also metrics that can be measured from source code. Section 3 discusses commonalities of metrics for several programming languages. Section 4 provides an overview of the proposed framework for source code measurement supporting multiple programming languages. Section 5 discusses a case study. Section 6 considers related works. Finally, Section 7 addresses conclusions and future work.

II. Background

A. *Problems with Existing Approaches*

The following summarizes the main problems with existing tools.

1) *Cost of New Development*

The implementation of a measurement tool supporting multiple programming languages would be very expensive, mainly due to the difficulty in implementing an analyzer to obtain information from the source code of every language. In addition, since developers increasingly use object-oriented programming languages, metric tools are built mainly to support that paradigm (in addition to the basic measurement of lines of code) [11]. Tools are often unavailable for other paradigms. However, procedural or functional languages are still popular, and tools compatible with these languages are therefore necessary.

2) *Incomprehensive Measurement*

Most existing source code measurement tools only assess software size and cannot evaluate software written in multiple languages. JSUnit [12] is an example of a program that uses more than one programming language. JSUnit is a client-server-type test framework that performs tests on web browsers and displays the results on the client side. The client side of JSUnit is implemented in JavaScript and the server side is implemented in Java. With existing tools, the measurement of JSUnit metrics requires that the Java and JavaScript code be separated and that two measurements be performed.

3) Inconsistent Measurement

Different measurement tools have different metrics definitions that may produce different results [11]. For example, we evaluated several Java code files within the package `net.jsunit.model` and some JavaScript code files within the `app` directory of JSUnit. We obtained the source code of JSUnit from its Github project page[1]. Table I shows the measurement results using different tools. In this example, lines of code (LOC) in Metrics 1.3.6 is defined as non-blank lines of code, but Source Monitor also counts blank lines when measuring LOC. Furthermore, Count Lines of Code (CLOC) skips blank lines and comment lines when measuring code lines.

TABLE I. DIFFERENCES BETWEEN METRICS TOOLS

File Name	Metrics 1.3.6 [5]	Source Monitor [7]		CLOC [10]	
	Lines	Lines	Statement	Blank	Code
AbstractResult.java	73	88	57	15	73
BrowserResult.java	197	248	147	51	197
BrowserSource.java	6	10	5	4	6
TestCaseResult.java	128	160	97	32	128
TestPageResult.java	45	61	32	16	45
TestRunResult.java	184	223	147	39	184
jsUnitCore.js	n/a	977	283	107	504
jsUnitParams.js	n/a	117	62	24	93
jsUnitTracer.js	n/a	48	33	9	39

B. Source Code Metrics

This section reviews the source code measurement systems that are most commonly used. We focus on size and complexity because they are the most important measurements [13].

1) Size Measurement

Lines of code (LOC) are the traditional measurement of software size. We define: the number of physical lines of target source code for measurement (*total lines of code*); the number of lines that contain the instructions necessary to execute a program (*effective lines of code*); the number of lines that contain only comments (*comment lines of code*); and the number of the lines that contain only spaces, tabs, or newline character(s) (*blank lines of code*). Measuring the number of "statements", which are logical measures based on the specification of the programming language, could also be a component of size measurement.

[1] https://github.com/pivotal/jsunit

2) Complexity Measurement

The following are several complexity measurements [13].

- Cyclomatic complexity: the complexity measurement proposed by McCabe, which is a measure of the number of control flows within a module.
- Halstead's software science: complexity metrics based on the number of operators and operands in a program.
- Information flow metrics: these measure the information flow into and out of modules, and indicate the cohesion of the program.
- CK metrics: a set of object-oriented design metrics by Chidamber and Kemerer, consisting of six metrics that measure class size and complexity, use of inheritance, coupling between classes, class cohesion, and collaboration between classes.

III. ASSESSING SOURCE CODE METRICS AND PROGRAMMING LANGUAGES

To define the base of our framework, we conducted assessments of source code metrics and programming language features. These assessments produce the commonalities of source code's measurable elements. From these measurable elements, we can define a metric more easily and even customize a new metric by utilizing the predefined measurable elements.

We analyzed the metrics described in subsection II-B to determine what elements of a program are used by these metrics. The size measurements are simply related to the

TABLE II. MEASURABLE ELEMENTS USED BY METRICS

Measurable Elements	STMT[a]	Halstead[b]	CK Metrics[c]					
			WMC	DIT	NOC	CBO	RFC	LCOM
statement	√							
operator		√						
operands		√						
Entities								
class			√	√	√	√	√	
method			√				√	√
attribute								√
Relationships								
method calls method						√	√	
class inherits from class				√				
class is a child of class					√			
class uses instance of other class						√		
class is parent to class					√			
class/method uses attribute								√

a. Number of statements. b. Halstead's software science.

c. The CK metrics: weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), coupling between object (CBO), response for class (RFC), and lack of cohesion in methods (LCOM) physical lines of source code. The number of statements is based on the "statement" definition in the programming language. The cyclomatic complexity is measured based on control flow, and Halstead's software science is related to operators and operands. Reference [14] has developed a data model of information from which most of the proposed object-oriented design metrics can be computed. The data model lists 12 entities and 17 relationships that are used by the metrics. We found that at the implementation level, the "class sends message to class/method" relationship is similar to "method calls method", so we combined them. In addition, we also combined the "class uses attribute of class" and "class/method uses attribute" relationships. Table II summarizes the measurable elements that are used by number of statements, Halstead's software science metrics, and CK Metrics.

We expanded this preliminary measurable elements list so we could assess other features of programming languages. We also wanted to cover measurable elements provided by non-object-oriented languages.

We included 20 programming languages in our analysis, based on [15]. Table III summarizes the measurable elements that could be quantified in each programming language. Table III consists of two parts. The first part describes the paradigm used by each programming language, based on the definition for each paradigm as presented in [16]. The checkmarks ("√") show the supported paradigms.

The second part depicts the measurable elements. A checkmark indicates that the measurable element could be quantified in the corresponding programming language. Every programming language has statements, operators, and operands that we could count. Therefore, we inferred that the number of statements and Halstead's software science metrics could be calculated for each language. For other elements, the results varied by programming language.

Non-object-oriented programming languages do not utilize the class and inheritance concepts. Alternatively, they use other concepts that can be measured, such as modules, functions, and procedures. We can possibly use these elements to produce complexity measurements such as numbers of functions, numbers of procedures, etc.

In comparison, languages designed mainly for object-oriented programming such as Java, C#, C++, and Python incorporate measurable elements that permit us to assess object-oriented metrics. Nevertheless, historically imperative languages that have been extended with some object-oriented features such as Perl and Ada yield different results, especially in association with the class concept. JavaScript is also a distinctive case.

Measurable elements are readily distinguished based on language syntax. In Perl and Ada, we can declare a class using the `package` keyword. However, the same keyword is also used to declare namespace, subprogram, or data types. Therefore, we cannot determine that all `package` keywords indicate the presence of classes. In spite of this, Perl has the special keyword `@ISA` and Ada uses `tagged` to indicate inheritance, both of which allow us to count the measurable elements related to inheritance. We could possibly designate packages associated with the inheritance keyword as "classes" instead of "packages", but we would perhaps incorrectly measure the number of classes since classes without inheritance would not be counted. In other words, we could partially count the "class" number. We note such cases with dots ("●") in Table III.

JavaScript is actually a prototype-based scripting language. Although it supports the object-oriented paradigm, the identification of classes or inheritance is complicated by the fact that we utilize `function` to declare a class. Therefore, in JavaScript we interpreted `function` syntax as representing a "function" although semantically it could possibly represent a "class".

The above results show that object-oriented design metrics could not be applied to every object-oriented programming language due to the availability of the measurable elements. Based on Table II, we required the "class" entity in order to measure CK Metrics. We are able to measure WMC, DIT, NOC, CBO, and RFC for Java, C#, C++, Objective-C, PHP, (Visual) Basic, Python, Delphi, and Ruby, but we cannot measure them for JavaScript since we are unable to precisely identify its classes.

IV. OVERVIEW OF PROPOSED FRAMEWORK

We propose a framework to measure source code that supports multiple languages and will diminish the problems previously described in subsection II-A. The following are the main features of the measurement framework:

- The framework should provide an easy way to measure a metric.
- The framework should be able to evaluate the metrics of software written in more than one programming language.
- The framework should use the same metric definition standard.

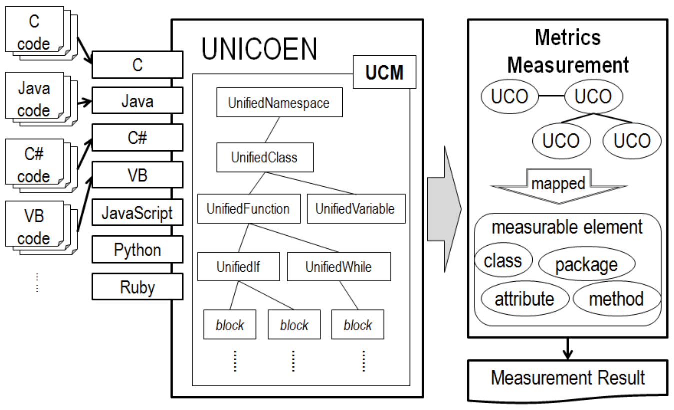


Figure 1. Overview of proposed framework.

Figure 1 shows the overview of our proposed framework. In order to reduce the cost of developing the framework, we used UNICOEN, a unified framework for code engineering, supporting multiple programming languages currently being developed by [17]. UNICOEN supplies a common

TABLE III. MEASURABLE ELEMENTS IN EACH PROGRAMMING LANGUAGE

Programming Languages		**Java**	**C**	**C#**	**C++**	**Objective-C**	**PHP**	**(Visual) Basic**	**Python**	**Perl**	**JavaScript**	**Delphi**	**Ruby**	**Lisp**	**Pascal**	**Transact-SQL**	**PL/SQL**	**Ada**	**Logo**	**R**	**Lua**
Paradigm	imperative	√	√	√	√		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	object-oriented	√		√	√	√	√	√	√	√	√	√	√					√		√	
	functional										√			√					√	√	√
	logic															√	√				
Measurable elements: entities																					
statement		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
operator		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
operand		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
namespace				√	√		√	√				√								√	
package		√								√							√	√			
class		√		√	√	√	√	√	√	●		√	√					●		√	
module			√												√						√
method		√		√	√	√		√	√	√		√	√							√	
procedure			√												√	√	√	√			
function			√				√				√			√	√	√	√	√			√
structure/record			√	√	√	√				√		√			√			√			
union			√		√	√												√			
attribute/variable		√	√	√	√	√	√	√	√	√	√	√	√		√			√		√	
method arguments		√		√	√	√		√	√	√		√	√							√	
procedure arguments			√												√	√	√	√	√		
function arguments			√				√				√			√	√	√	√	√	√		√
return value		√	√	√	√	√	√	√	√	√	√	√	√	√		√	√	√		√	√
global variable							√		√				√								
abstract class		√		√	√		√	√	√									√			
interface		√		√			√											√			
Measurable elements: relationships																					
method calls method		√		√	√	√		√	√	√		√	√							√	
class inherits from class		√		√	√	√	√	√	√	√		√	√					√		√	
class is a child of class		√		√	√	√	√	√	√	√		√	√					√		√	
class uses instance of other class		√		√	√	√	√	√	√	√		√	√					√		√	
class is parent to class		√		√	√	√	√	√	√	√		√	√					√		√	
class/method uses attribute		√		√	√	√	√	√	√	√		√	√					√		√	
procedure/function calls procedure/function			√				√				√			√	√	√	√	√			√

representation of source code for different programming languages called the unified code model (UCM), and objects generated from the source code according to the UCM are called unified code objects (UCOs). At present, UNICOEN supports seven programming languages: C, Java, C#, VB, JavaScript, Python, and Ruby.

The basic purpose of the framework was to define the measurable elements we obtained in Section III. These elements were obtained by mapping UCOs generated by UNICOEN to our definitions. UNICOEN generates UCOs from the input source code. After the mapping process, we could easily count each element and then utilize it to evaluate the corresponding metrics.

The mapping process is described as follows. The partial structure of the UCM is depicted in Figure 1. Each node in the UCM will constitute one UCO block, and relationships among the blocks will form a tree structure. A UCO block contains primary information such as UnifiedNamespace, UnifiedClass, UnifiedVariable, UnifiedFunction, etc., whose names refer to the types of entities they are. There is also additional information such as modifier, type, extend-constrain, identifier, etc., which describe the details of each entity.

Based on this information, for example, we can map a UnifiedNamespace to the "package" entity for Java code or to the "namespace" entity for C# code. A UnifiedFunction is then mapped to "method" for Java code. For C code, UnifiedFunction could be mapped to "function" if the block contains UnifiedReturn information or "procedure" if it contains no UnifiedReturn. Currently the UCOs can cover all of the measurable elements for the seven programming languages mentioned previously.

V. Case Study

In this section, we illustrate the evaluation of metrics using our approach. For the case study, we analyzed the same files listed in Table I. We describe the measurement of total and blank lines of code, number of statements, and number of children.

A. *Measuring Total and Blank Lines of Code*

These measurements are directly related to the physical lines of source code, so did not require any mapping from UCOs. The framework read each file line-by-line and counted the number of times a line was read. This resulted in total lines of code (TLOC). Blank lines of code (BLOC) was defined as the number of lines that contained only spaces, tabs, or newline character(s). It was calculated by reading each line and tallying if the line has only these character types. The results of our measurements are summarized in Table IV. To obtain the number of lines excluding blank lines, we can easily subtract BLOC from TLOC.

B. *Measuring Number of Statements*

While the lines of code are generically called "the physical lines of code", the number of statements (NOS) is called "the logical lines of code". NOS is measured based on UCOs. Each block in each UCO is basically mapped to a statement. Therefore, the NOS value is exactly the same as the total number of blocks in the UCOs.

```
package net.jsunit.model;
import java.util.List;
public interface BrowserSource {
    Browser getBrowserById(int id);
    List<Browser> getAllBrowsers();
}
```

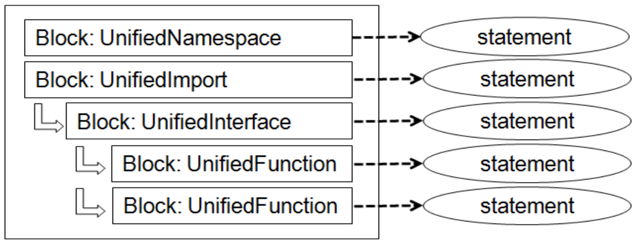


Figure 2. Mapping of BrowserSource.java to UCO and then to measurable elements (statements).

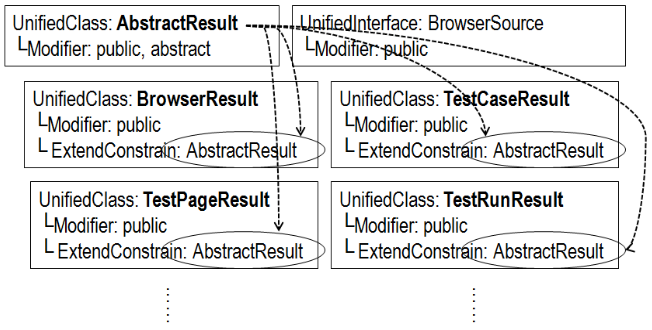
TABLE IV. TLOC, BLOC, and NOS Measurement Results

File Name	TLOC	BLOC	NOS
AbstractResult.java	89	16	56
BrowserResult.java	249	52	145
BrowserSource.java	11	5	5
TestCaseResult.java	160	32	95
TestPageResult.java	62	17	32
TestRunResult.java	224	40	142
jsUnitCore.js	978	108	361
jsUnitParams.js	118	25	62
jsUnitTracer.js	48	9	29

Figure 2 describes how the code in BrowserSource.java is represented using UCOs. Each block is considered as one statement, so the NOS for BrowserSource.java is 5. The mapping mechanism for JavaScript code is the same as for Java code. The NOS results are summarized in Table IV.

C. *Measuring Number of Children*

The number of children (NOC) is defined as the number of immediate subclasses subordinated to a class in the class hierarchy. To measure the NOC we distinguished the relationships "class is a child of class" and "class is parent to class". Determining these relationships based on UCOs is not as simple as mapping UCOs to entities such as packages, classes, etc. A class that inherits another class will contain extend-constrain information in its UnifiedClass block. To find the inheritance relationship, we traced whether the "extend-constrain" name of a class was identical to that of any class name within the project. Using our sample code, this approach is demonstrated in Figure 3.



UnifiedClass: AbstractResult
Modifier: public, abstract
UnifiedInterface: BrowserSource
Modifier: public
UnifiedClass: BrowserResult
Modifier: public
ExtendConstrain: AbstractResult
UnifiedClass: TestCaseResult
Modifier: public
ExtendConstrain: AbstractResult
UnifiedClass: TestPageResult
Modifier: public
ExtendConstrain: AbstractResult
UnifiedClass: TestRunResult
Modifier: public
ExtendConstrain: AbstractResult

Figure 3. Tracing the children of the AbstractResult class.

In this case, we obtained the following relationships. The TestPageResult, TestRunResult, TestCaseResult, and BrowserResult classes were children of the AbstractResult class. The AbstractResult class was the parent of the TestPageResult, TestRunResult, TestCaseResult, and BrowserResult classes. We could infer, then, that the NOC for the AbstractResult class was 4.

Using this approach makes it easier to map the UCOs to measurable elements and to determine relationships. Although currently UNICOEN supports only seven languages, it is easy to add new languages or update existing languages, so we could easily add measurements for other languages, thus reducing the cost of development. In addition, this approach allows us to fully measure various metrics for programs written in more than one programming language using only one framework. Regarding the problem with measurement inconsistencies demonstrated in Table I, we cannot say whether our results are more appropriate since there are no international standards prescribing the optimal way to measure a metric. However, our approach used the same metric definition for each programming language supported by our framework. Hence, we could obtain consistent results among all the languages, for instance between Java and JavaScript code as shown in Table IV.

VI. Related Works

This section discusses the related research approaches of Baroni et al. [18], Higo et al. [19], and Maneva et al. [20]. The goals of these research groups were similar to our own in terms of measuring software metrics.

Baroni et al. developed MOOSE as a language-independent platform that outputs object-oriented design metrics. MOOSE analyzes OCL and UML metamodel representations. Their approach is limited because the amount of information obtained from their representation is less than that which can be directly extracted from source code. The UCOs generated by UNICOEN contain more information than the representations used by MOOSE. For instance, with UCOs we can trace the “method calls method” and “class/method uses attribute” relationships, but MOOSE is unable to recognize them.

Higo et al. built MASU as a plug-in for measuring metrics of source code written in multiple programming languages. Their work focused primarily on developing a source code analyzer for each programming language. This analyzer built AST from the input source code of object-oriented programming languages. Currently MASU handles Java and C# source code. Our approach simplifies the measurement by assessing only the measurable elements. In addition, MASU was designed to work only with object-oriented languages, whereas our framework can handle non-object-oriented language such as C.

Maneva et al. proposed using a framework to evaluate source code. The key ideas are that a base set of metrics must be defined and the computation of the measures are distributed into distinct modules. They implement preprocessor functions to filter the source code artifacts from elements that render the computation of metric values inaccurate in some contexts. Their research does not mention a means of processing multiple languages, which is the main contribution of our approach.

VII. Conclusion and Future Work

This paper described our proposed unified source code measurement framework for multiple programming languages. We reported three main problems with existing source code metric tools: cost of new development, incomprehensive measurement, and inconsistent measurement. Based on size and complexity measurements, we assessed source code metrics and various programming languages and obtained the commonalities of measurable source code elements. The assessment results showed that we were able to measure size and some complexity metrics for every programming language. However, in some cases we could not measure object-oriented design metrics even though a language such as JavaScript supports the object-oriented paradigm, while in other cases, such as Perl and Ada, we could only partially conduct measurements. The next step in our approach is to identify measurable elements by mapping the representations produced by UNICOEN. As a case study, we explained the way our framework measures the total and blank lines of code, number of statements, and number of children.

We limit our work to several size and complexity metrics, so some other metrics were not considered. In addition, our framework is still early in development. Further work is needed to evaluate whether it can properly solve the problems we mentioned previously. But we believe that our approach can contribute to reducing the cost of developing a source code measurement tool, even for software written in more than one programming language. In addition, most current tools only perform static analysis of one version of software, and we will also therefore consider identifying distinctions between two versions of source code.

References

[1] N. Fenton, “Software measurement: a necessary scientific basis,” IEEE Transactions on Software Engineering, vol. 20, pp199–206, March 1994.

[2] Virtual Machinery, “JHawk,” http://www.virtualmachinery.com.

[3] Semantic Designs, Inc., “Java Source Code Metrics,” http://www.semanticdesigns.com.

[4] Clarkware Consulting, inc., “JDepend,” http://www.clarkware.com/software/JDepend.html.

[5] F. Sauer, “Eclipse Metrics plugin 1.3.6,” http://metrics.sourceforge.net.

[6] Chr. Clemens Lee, “JavaNCSS - A Source Measurement Suite for Java,” http://javancss.codehaus.org.

[7] Campwood Software, “SourceMonitor,” http://www.campwoodsw.com.

[8] M Squared Technologies, “Resource Standard Metrics,” http://msquaredtechnologies.com.

[9] D. A. Wheeler, “SLOCCount,” http://www.dwheeler.com/sloccount.

[10] Northrop Grumman Corporation, “CLOC - Count Lines of Code,” http://cloc.sourceforge.net.

[11] R. Lincke, J. Lundberg, and W. Lowe, “Comparing software metrics tools,” Proc. of International Symposium on Software Testing and Analysis, pp. 131–142, July 2008.

[12] E. Gamma and K. Beck, “JSUnit,” http://www.jsunit.net.

[13] L. M. Laird and M. C. Brennan, Software measurement and estimation: a practical approach, IEEE Computer Society, 2006.

[14] J.R. Abounader and D. A. Lamb, A data model for object-oriented design metrics. Kingston, ON: Queen’s University, 1997.

[15] TIOBE Software, “TIOBE Programming Community Index for January 2012,” http://www.tiobe.com.

[16] A. B. Tucker and R. E. Noonan, Programming Languages Principles and Paradigms, 2nd ed, McGraw-Hill, 2007.

[17] K. Sakamoto, A. Ohashi, D. Ota, and H. Iwasawa, “UNICOEN,” http://www.unicoen.net.

[18] A. L. Baroni and F. B. Abreu, “An OCL-based formalization of the MOOSE metric suite,” Proc. of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, July 2003.

[19] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, “A pluggable tool for measuring software metrics from source code,” Proc. of the Joint Conference of the 21th IWSM/MENSURA, November 2011.

[20] N. Maneva, N. Grozev, and D. Lilov, “A framework for source code metrics,” Proc. of 11th International Conference on Computer Systems and Technologies, June 2010.