A Pluggable Tool for Measuring Software Metrics from Source Code

Yoshiki Higo, Akira Saitoh, Goro Yamada, Tatsuya Miyake, Shinji Kusumoto, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {higo,a-saitoh,g-yamada,t-miyake,kusumoto,inoue}@ist.osaka-u.ac.jp

***Abstract*—This paper proposes a new mechanism to measure a variety of source code metrics at low cost. The proposed mechanism is very promising because it realizes to add new metrics as necessary. Users do not need to use multiple measurement tools for measuring multiple metrics. The proposed mechanism has been implemented as an actual software tool MASU. This paper shows how using MASU makes it easy and less costly to develop plugins of the CK metrics suite.**

***Keywords*-Software Metrics, Measurement**

I. Introduction

Software metrics are measures for various reasons such as evaluating software quality, and they are measured from software produces such as source code [1]. At present, there are a variety of software metrics. The lines of code is the simplest and the most popular source code metric that we use. The CK metrics suite [2], which indicates features of object-oriented systems, is also widely used. The cyclomatic complexity [3], which represents an internal complexity of software modules, is a good indicator to assume buggy modules. Software metrics are defined on conceptual modules of software systems, that is, they can be measured from any programming languages that have the same conceptual modules.

However, it is extremely costly to obtain such conceptual information from the source code because it requires deep source code analyses. At present, there are many software tools such as compiler-compilers for supporting source code analysis. Unfortunately, their support is up to syntax analysis like building abstract syntax trees. If we need more deep information such as def-use relationship or call relationship, we have to implement such semantic analyses by hand.

Most of available measurement tools handle only a single programming language [4], [5], [6], [7], [8]. If we want to measure source code metrics from multiple programming languages, we have to prepare measurement tools for every of them. However, every tool measures different predefined source code metrics. Also, different tools calculate different values for the same metrics because the details of the metrics are differently interpreted by the tools [9]. In order to make the situation better, it is desirable to realize an unified metrics measurement from multiple programming languages. The proposed mechanism has the following features:

- it handles the source code of multiple programming languages,
- it realizes an unified metrics measurement from the languages,
- it takes plugins for new metrics, all users have to do is writing the metrics measurement logics. They do not have to implement semantic analysis that they need.

II. Related Work

A. Source Code Analysis tools

Baroni et al. developed a language-independent engineering platform, MOOSE [17]. MOOSE takes the source code of software systems and outputs software metrics as well as MASU. However, MOOSE does not directly analyze the source code. It analyzes intermediate representations such as CDIF and XMI, which are generated from the input source code. That means MOOSE provides model-based information to users. The model-based information is suitable to visualization, however the amount of it is smaller than the information directly-extracted from the source code. Also, MOOSE depends on third party tools in generating CDIF and XMI. Consequently, it does not have the same extendability and changeability as MASU.

Collard proposed srcML, which contains the source code information in XML format [18]. srcML handles C/C++ and Java source code. srcML provides information extracted by syntax analysis. Deep information extracted by semantic analysis is not included in srcML. Consequently, it is less suitable to software metrics measurement than MASU.

Antoniol et al. developed XOgastan, which is a multilingual source code analysis tool based on GCC [19]. In principle, XOgastan can handle all the programming languages that GCC handles. However, the amount of information provided by XOgastan is not enough to software metrics measurement. It does not provide the information about expressions used in every module. Also, their paper does not express about artifices for making it easy to add new analyzers for getting new information.

Fukuyasu et al. developed a CASE tool platform, Sapid based on fine-grained software repository [20]. Chen et al. developed a C program abstraction system [21]. These tools analyze more detail information than MOOSE, srcML, and XOgastan and provide the same kind of information as MASU. They can be used as metrics measurement platforms. The difference between MASU and these tools are followings:

- MASU provides an unified way to measure metrics from multiple programming languages;
- logics of metrics measurements are separated from the main module, which realizes ease of adding new measurement logics.

It is necessary to analyze templates in order to obtaining precious information from C/C++ source code. However, analyzing templates is very complicated and only a small number of analysis tools can analyze templates richly. Gschwind et al. developed TUAnalyzer, which uses GCC for realizing templates analysis [22]. We have a plan to extend MASU for handing C/C++. Using GCC as a preprocessor for analyzing templates is an option.

III. Preliminaries

A. Software Metrics

Software metrics are measures used for a variety of purposes such as evaluating software quality or predicting development/maintenance cost [1]. The representative metrics should be the CK metrics suite [2] and the cyclomatic complexity [3]. Most of software metrics are defined on the conceptual modules of software systems such as file, class, method, function and data flow. That means those metrics can be measured from any programming languages if they share the same conceptual modules.

Herein, we introduce every member of the CK metrics suite and the cyclomatic complexity because they are used in the latter of this paper.

- **WMC (Weighted Methods per Class)**: This metric is the sum of the complexities of the methods defined in the target class. Up to now, several methods measuring method complexity have been proposed, and the cyclomatic complexity [3] and the Halstead complexity measurement [10] are commonly used. Sometimes, this metric is simply the number of methods defined in the class.
- **DIT (Depth of Inheritance Tree)**: This metric represents the depth of the target class in the class hierarchy.
- **NOC (Number Of Children)**: This metric represents the number of classes directly derived from the target class.
- **CBO (Coupling Between Object classes)**: This metric represents the number of classes coupled with the target class. In the definition of this metric, there is a coupling between two classes if and only if a class uses methods or fields of the other class.
- **RFC (Response For a Class)**: This metric is the sum of the number of local methods and the number of remote methods. A local method is a method defined in the target class, and a remote method is a method invoked in local methods with the exception that local methods invoked in local methods are not counted as remote methods.
- **LCOM (Lack of Cohesion in Methods)**: This metric represents how much the target class lacks cohesion. This metric is calculated as follows: takes each pair of methods in the target class; if they access disjoint set of instance variables, increases P by one; if they share at least one variable access, increases Q by one.

$$LCOM = \begin{cases} P - Q & (if P > Q) \\ 0 & (otherwise) \end{cases}$$

- **CYC (CYClomatic complexity)**: This metric indicates the number of paths from the method enter to the method exit. the larger the value is the more complex it is.

There are many software metrics that do not have complete definitions. For example, WMC, which is a member of the CK metrics suite, represents the sum of weighted methods count. However, how methods are weighted is not defined in WMC. Such ambiguities in metrics definitions unintentionally have impacts on metrics measurement.

Also, a present metric may not be the best for the specified purposes. There are often happens that modified metrics are proposed for the existing metrics or new metrics are proposed for the same purposes. For example, a member of CK metrics LCOM is known that it cannot indicate the lack of cohesion appropriately in the specified situations. At present there are multiple LCOMs that have different definitions [11], [12].

B. Java bytecode Analysis Tools

- **Soot** is a Java bytecode optimization framework that was developed as a project in McGill University [15]. Soot provides a variety of information extracted from bytecode for optimization, which can be used for not only optimization but also other kinds of analyses.
- **WALA** is a byte code analysis library developed by IBM Research [16].

Bytecode analysis tools provides useful information extracted from actually-executed code. However, there is information that can be extracted from only the source code, such as the kind of conditional blocks. Bytecode is a executable code that are generated from compilers, and not bytecode but source code is the target of maintenance. Software metrics are used for evaluating human-made products or predicting maintenance cost that human has to do. Consequently, in these contexts, bytecode analysis tools are not appropriate to be used.

C. Metrics Measurement

Measuring source code metrics generally consists of the following two steps:

- **STEP1**: analysis the source code to extract information required for measuring the specified metrics,
- **STEP2**: measure the metrics by using the extracted information.

As mentioned in Subsection III-A, software metrics can be measured from multiple programming languages that have the same conceptual modules. However, in STEP1 of the metrics measurement, we have to prepare source code analyzers for every of programming languages.

At present, there are many measurement tools [4], [5], [6], [7], [8]. However, it it difficult to measure multiple metrics from multiple programming languages in an unified way. That is caused by the following reasons.

- Implementing a source code analyzer requires much effort. However source code analyzers have to be prepared for every programming language. Most of existing measurement tools handle only a single programming language.
- If we measure metrics from multiple programming languages, we have to prepare measurement tools for every of the languages. However, most of software metrics have ambiguous definitions, so that measurement results are quite different from measurement tools [9].
- It is difficult to reuse source code analyzers of existing measurement tools for measuring new metrics because every software metric requires different information.

All the above problems are in the STEP1, that is, if we improve STEP1, metrics measurement will be conducted at lower cost.

D. Source Code Analysis

Source code analysis is a technique to automatically extract required information from the source code. Extracted information is used for measuring software metrics or other activities for the evaluating software systems. However, many software metrics require deep information in the source code such as def-use relationship or call relationship. At present, there are a variety of compiler-compilers such as JavaCC [13] or ANTLR [14] to support implementing source code analyzers. Unfortunately, their supports are up to syntax analysis. If we need deeper information, we have to implement semantic analyzers by hand, which requires much efforts.

On the other hand, opportunities of source code analysis are increased, so that source code analysis is of growing importance in software development and maintenance. For example, software companies began to investigate developing and maintaining software systems with software engineering methodologies for QA (Quality Assurance). As a part of QA, source code analysis is performed for metrics mesurement or other activities. However, activity of source code analysis is far from satisfactory because of its high cost.

The high cost on source code analysis is caused by difficulty of implementation of source code analyzer and shortfall in human resources for implementing it. That means, engineers who have to evaluate software systems are suffered from not implementing algorithms of software metrics measurement but implementing source code analyzers used for extracting required information from the source code.

IV. Required Measurement Tools

Herein, we describe functional requirements for measurement tools and our policy for designing and implementing the functionalities.

A. Functional Requirements

The followings should be functional requirements for effective and efficient metrics measurements.

Multilingualization: A measurement tool should handle multiple programming languages that are widely-used.

Unified Definition of a Metric: There should be an unified definition for every metric. Such unified definitions realize to measure metrics from multiple programming languages with the exactly same logics.

Pluggable Interface: Different metrics are used in different organizations and different contexts, which means that a measurement tool should have a pluggable interface for measuring any kinds of software metrics.

B. Policies for Designing and Implementing a Measurement Tool

To realize the functional requirements described in Subsection IV-A, the following policies should be appropriate for designing and implementing a measurement tool.

Separation of Source Code Analyzer and Metrics Measurement Module: This separation realizes to use a single metric measurement module for multiple programming languages, so that it is possible to use unified definitions of software metrics for multiple languages.

Absorbing Differences between Languages: In the development of a measurement tool, building source code analyzer requires more cost than building metrics measurement module. Consequently, in order to reduce the cost, a framework for reusing modules in the source code analyzers must be prepared. To realize this framework, we absorb the differences between programing languages at the early stage of source code analysis. After the absorption, common analysis modules can be used for multiple languages.

One-on-one Correspondence between Measurement Module and Metric: A measurement module should measure only a single metric. This design is suitable to add/remove measurement modules into/from the measurement tool. Adding/removing a measurement module does not have any impact on other measurement modules.

V. MASU: Metrics Assessment plugin platform for Software Unit

Based on the policies described in Section IV-B, we are developing a measurement tool **MASU**. Java language is used for implementing MASU. Currently, MASU consists of

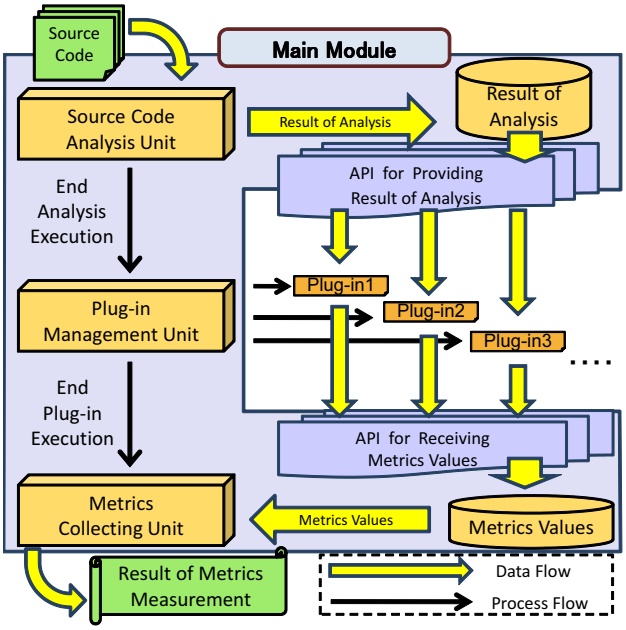


Figure 1. Architecture of MASU

519 source files and 102,000 LOCs (41,000 LOCs are automatically generated code by ANTLR). MASU takes source code of object-oriented programming language and bring out metrics measurement result. Currently, MASU is under construction, however it can handle Java full-grammar and C# partial grammar. MASU is managed in SourceForge, and it is freely downloadable from http://sourceforge.net/projects/masu/.

Figure 1 shows the architecture of MASU. MASU consists of a **main module** and **plugins**. Subsections V-A and V-B describes the main module and plugins respectively.

A. Main Module

The main module is responsible for all the functionalities of MASU except metrics measurement logics. It includes three units, **Source Code Analysis Unit**, **Plugin Management Unit**, and **Metrics Collecting Unit**.

The execution of MASU proceeds as follows:

1) Source Code Analysis Unit analyzes the source code and builds language-independent data, which is correspond to conceptual modules of programming languages.
2) Plugin Management Unit executes plugins that are registered to the main module in advance. All the plugins obtain information, which is required for metrics measurement, from the main module by using APIs. After metrics measurement, the plugins send the measurement results to the main module by using APIs.
3) Metrics Collecting Unit collects metrics measurement results and outputs them in the CSV format.

The remainder of this subsection focuses on Source Code Analysis Unit because it includes technological implementations to absorb the differences between programming languages. Other two units are rather simple implementations.

Source Code Analysis Unit consists of AST Building Part and AST Analysis Part. Firstly AST Building Part builds language-independent ASTs from the input source code, then AST Analysis Part extract data by analyzing the ASTs.

1) AST Building Part: In Source Code Analysis Unit, firstly language-independent ASTs are built from the input source code. Figure 2 shows an example of a language-independent AST. In the left side of Figure 2, there are three code portions of Java, VisualBasic, and C#. Every of them has a different syntax from one other meanwhile their semantics are the same. In the right side of Figure 2, there is a language-independent AST from the three code portions. In the AST, all the syntax differences between the code portions are absorbed. Building language-independent ASTs is quite suitable to reduce cost for developing source code analyzer because it is only once that we have to implement AST analyzers for semantic analysis. Note that AST builders have to be prepared for every programming language.

Absorbing the differences between programming languages is realized by the following operations:

Definition of Common Nodes: MASU defines a common AST nodes for preserved keywords that have the same semantics. In Figure 2, the inheritance relationship is presented by “extends”, “:”, and “inherits” in the three languages. In AST, this relationship is presented by “INHERITANCE” node.

Insertion of Definition Nodes: MASU inserts definition nodes that represent correspondences between its sub nodes and source code elements. This technique can absorb the different orders of program elements between different programming languages. In Figure 2, the order of formal parameter definition in VisualBasic is different from Java and C#. In the language-independent AST, MASU inserts “TYPE” and “NAME” nodes for absorbing the different orders in the formal parameter defintion.

Moving and Deleting Nodes: MASU moves or deletes some nodes for absorbing relationships between nodes such as parent-child or sibling. For example, in Java language, nodes representing namespaces have sibling relationships with nodes representing class definitions. On the other hand, in C# language, they have parent-child relationships. In MASU, the relationship between namespace and class definition is defined as a sibling relationship.

2) AST Analysis Part: This part analyzes ASTs built in AST Building Part and constructs data that has language-independent structures. Herein, a language-independent structure is a conceptual element shared by multiple programming languages. For example, *file*, *class*, *method*, *field*, *data flow*, *namespace*, and *type* are typical language-independent data structures.

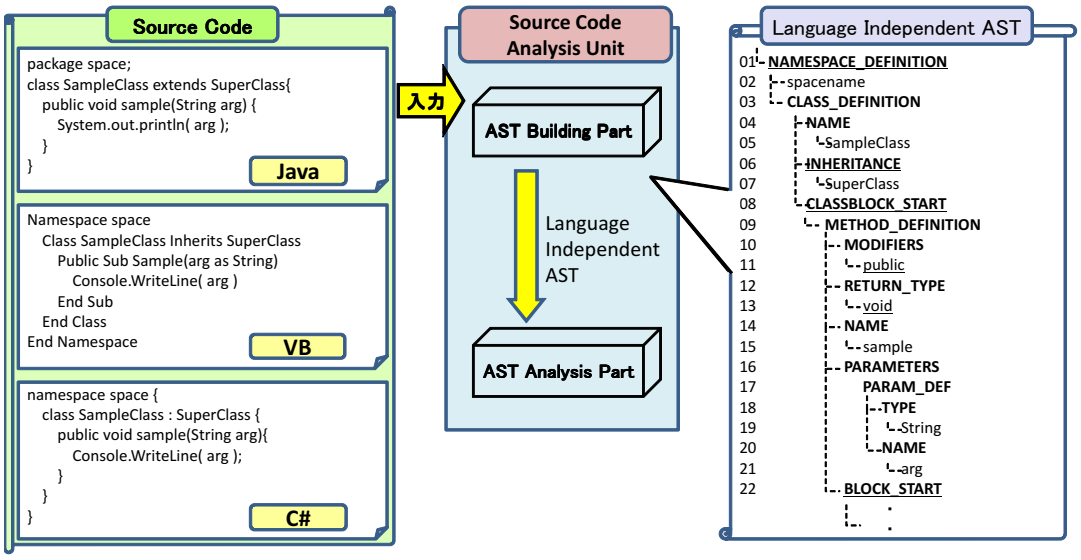


Figure 2. An Example of Building Language-Independent AST

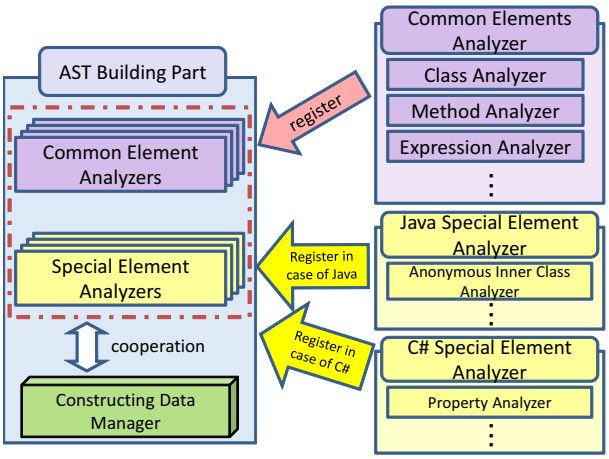


Figure 3. Mechanism of AST Analysis Part

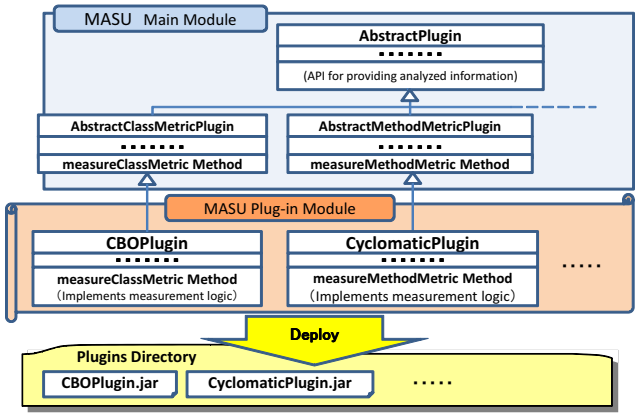


Figure 4. Class Hierarchy Related to Plugins

Figure 3 shows a simple example of the mechanism of AST Analysis Part. This part consists of multiple analyzers and a constructing data manager. Every analyzer is prepared for a single element in a programming language, and analyzers extract information for constructing language-independent data in cooperation with the constructing data manager.

For example, the method analyzer analyzes only methods in a software system. It cooperates with the expression analyzer and the identifier analyzer to extract the detail information within every method. The method analyzer cooperates with the constructing data manager to getting its owner class information.

Syntax differences between different programming languages have already absorbed in AST Building Part, so that a single analyzer can be used for multiple programming languages. This architecture reduces the implementation cost.

B. Plugin

Logics of metrics measurements are contained in not the main module but plugins. Metrics measurements are completely separated from source code analysis in MASU. This separation realizes to add new metrics measurements and remove unnecessary metrics measurements.

A plugin contains a logic of a single metric measurement. Plugins obtain information required for measuring metrics from APIs of the main module. Then, they calculate and push the metrics values to the main module by using the APIs.

Figure 4 shows the class hierarchy related to plugins. MASU provides abstract classes having definitions of APIs for data exchanges between the main module and plug-ins. Plugins are implemented as subclasses of the abstract classes. There are 4 abstract classes and each of them is correspond to metrics measurement unit. The 4 units are *class*, *method*, *field*, and *file*. If a user implements plugins of the CK metrics suite, s/he uses AbstractClassMetricPlugin, which is an abstract class for *class* because the definitions of the CK metrics suite are on *class*. The logic of class metric measurement is implemented in method measureClassMetric.

Implemented plugins are archived as Jar files and placed on directory "plugins", then the main module automatically identifies them and measures the metrics by using the measurement logics defined in the plugins.

C. Security

Data in the main module is shared by multiple plugins. This means, if a plugin intentionally or unintentionally changes the data wrongly, other plugins may not measure metrics correctly. In order to avoid this problem, in MASU, every plugin is executed by a different thread. MASU has a security manager that controls accessibilities to the main module from plugins on thread level. This security manager can prevent the above problem from occurring.

VI. Evaluation

A. Performance

In order to evaluate the performance of MASU, we actually analyzed source code with MASU. In this evaluation we used a personal workstation with the following equipment.

- **CPU**: PentiumIV 3.00 GHz
- **Memory**: 2.00 GBytes
- **OS**: Windows XP

In this evaluation, we analyzed two systems.

Java Platform Standard Edition 6: This is a Java software system. The number of file is 3,852 and the lines of code is about 1,200,000. The analysis time was 392 seconds, and the maximum memory usage was 838 MBytes.

CSgL: This is a C# software system. The number of file is 61, and the lines of code is about 30,000. The size of CSgL is much smaller than the Java software. However, CSgL is enough to check that language-independent AST generated from C# correctly works. The analysis time was 59 seconds and the maximum memory usage was 51 MBytes.

The performance evaluation showed that MASU can be applied to large-scale software systems with a personal workstation.

B. Plugin Development

In order to show the ease of plugin development with MASU, we developed all the members of the CK metrics suite and the cyclomatic complexity. Table I shows the size of developed plugins and their development time. All the plugins are developed by a single developer. He is a master's student and a member of MASU development team. At the time of the experiment, his programming experience was about 2.5 years. He was familiar with APIs provided by MASU because he is a development member of MASU. The column "total LOC" means the total lines of code of the developed plugins. The numbers inside parentheses are ones including brank lines and comments. The column "logic LOC" shows the lines of the method implementing the metrics measurement logics.

Figure 5 shows the entire source code of the RFC plugin. By using MASU, the measurement logic of RFC was implemented with only several lines. We do not have to implement any source code analyzer at all. Also, there are several methods that have to be implemented in plugins. The annotation @Override shows the methods. However, every of them has only a single instruction, which is very easy to implement.

VII. Conclusion

This paper presents a metrics measurement platform, MASU. MASU has the following features for realizing metrics measurements at lower costs:

- MASU handles multiple programming languages,
- MASU has an unified way to measure metrics from them,
- we only have to write logics of metrics that we want to measure.

At present, MASU is tenable for practical usage, indeed it is used in a variety of research related to source code analysis and metrics measurement. At present, MASU handles Java and C# source code and we are going to extend it for handling other languages such as C/C++, Visual Basic. Not only source code but also other kinds of products are target of MASU extensions. For example, it is interesting to use execution histories as a input of MASU for realizing metrics measurement based on dynamic data.

Acknowledgment

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, sup-

Table I
Size of Plugins for CK metrics suite and the cyclomatic complexity

Metrics	total LOC	logic LOC	time
WMC	31 (74)	2	10
DIT	35 (81)	8	20
NOC	36 (73)	1	10
CBO	61 (121)	29	20
RFC	56 (117)	7	15
LCOM	114 (221)	48	40
CYC	52 (115)	21	25

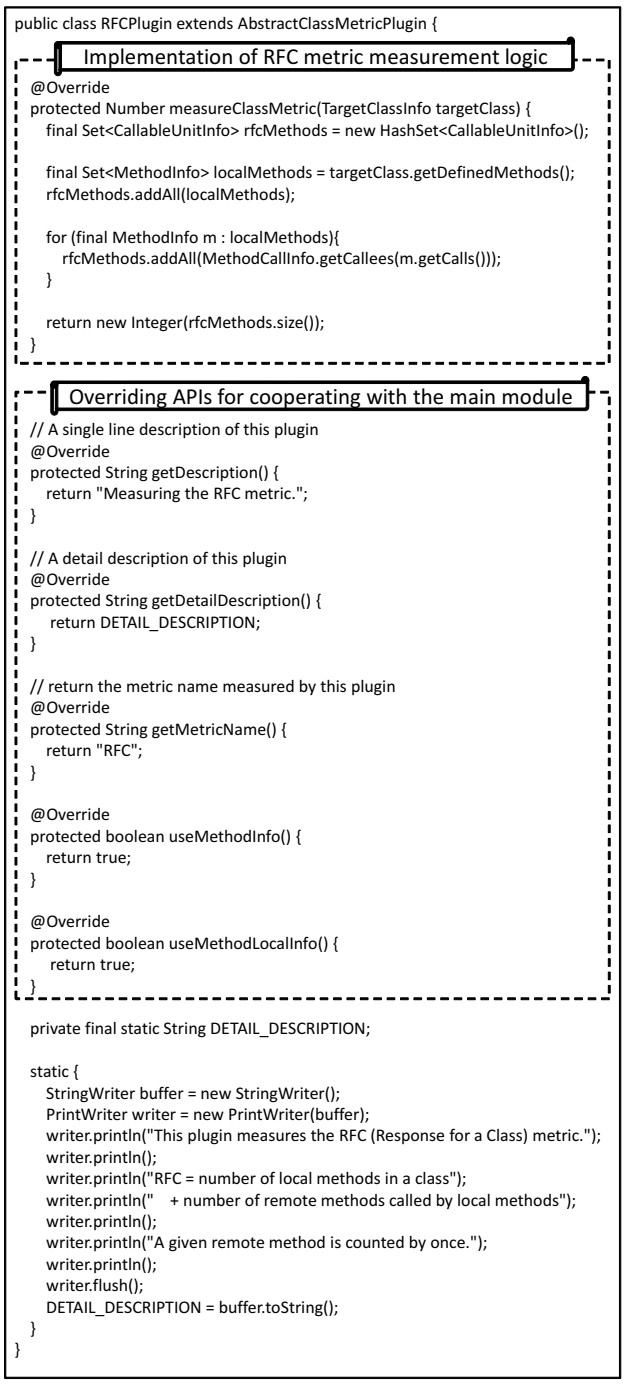


Figure 5. Entire Source Code of RFC Plugin

ported by Ministry of Education, Culture, Sports, Science and Technology. It has been performed under Grant-in-Aid for Scientific Research (A) (21240002) and Grant-in-Aid for Exploratory Research (23650014) supported by the Japan Society for the Promotion of Science.

REFERENCES

[1] P. Oman and S. L. Pleeger, *Applying Software Metrics*. IEEE Computer Society Press, 1997.

[2] S. Chidamber and C. Kemerer, "A Metric Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 476–493, June 1994.

[3] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[4] I. inc., "CodePro AnalytiX," http://www.instantiations.com.

[5] C. consulting inc., "JDepend," http://www.clarkware.com.

[6] V. Machinery, "JHawk," http://virtualmachinery.com.

[7] A. Cain, "JMetric," http://www.it.swin.edu.au/projects/jmetric/products/jmetric/default.htm.

[8] Aqris, "RefactorIT," http://www.aqris.com.

[9] R. Lincke, J. Lundberg, and W. Lowe, "Comparing Software Metrics Tools," in *Proc. of International Symposium on Software Testing and Analysis*, July. 2008, pp. 131–141.

[10] M. H. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[11] B. Henderson-Sellors, *Object-Oriented Metrics: measures of Complexity*. Prentice Hall, 1996.

[12] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Ojbect-Oriented Systems," in *Proc. of International Symposium on Applied Corporate Computing*, Oct. 1995, pp. 78–84.

[13] "JavaCC," http://javacc.dev.java.net/.

[14] "ANTLR," http://www.antlr.org/.

[15] S. R. Group, "Soot: a Java optimization Framework," http://www.sable.mcgill.ca/soot/.

[16] I. T. W. R. Center, "WALA," http://wala.sourceforge.net/wiki/index.php/Main_page.

[17] A. L. Baroni and F. B. Abreu, "An OCL-based formalization of the MOOSE Metric Suite," in *Proc. of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2003.

[18] M. L. Collard, J. I. Maletic, and A. Marcus, "Supporting Document and Data Views of Source Code," in *Proc. of ACM Symposium on Document Engineering*, 2003, pp. 34–41.

[19] G. Antoniol, M. D. Penta, G. Masone, and U. Villano, "XOgastan: XML-Oriented GCC AST Analysis and Transformations," in *Proc. of 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, 2003, pp. 173–182.

[20] N. Fukuyasu, S. Yamamoto, and K. Agusa, "An OCL-based formalization of the MOOSE Metric Suite," in *Proc. of International Workshop on Principles of Software Evolution*, 2003, pp. 43–47.

[21] Y. F. Chen, M. Y. Nishimoto, and C. V. Ramaoorthy, "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 325–334, Mar. 1990.

[22] T. Gschwind, M. Pinzger, and H. Gall, "TUAnalyzer – Analyzing Templates in C++ Code," in *Proc. of 11th Working Conference on Reverse Engineering*, 2004, pp. 48–57.

APPENDIX

We show all the source code of CK metrics plugins for presenting their simplicities. Figures 6, 7, 8, 9, and 10 show all the CK metrics plugins except RFC bacause it is already shown in Figure 5.

```
public class DITPlugin extends AbstractClassMetricPlugin {
  Implementation of DIT metric measurement logic
@Override
  protected Number measureClassMetric(TargetClassInfo targetClass) {

    ClassInfo classInfo = targetClass;
    for (int depth = 1;; depth++) {

      final List<ClassTypeInfo> superClasses = classInfo.getSuperClasses();
      if (0 == superClasses.size()) {
        return depth;
      }
      classInfo = superClasses.get(0).getReferencedClass();
    }
  }
  Overriding APIs for cooperating with the main module
@Override
  protected String getDescription() {
    return "Measuring the DIT metric.";
  }

@Override
  protected String getMetricName() {
    return "DIT";
  }

@Override
  protected boolean useFieldInfo() {
    return false;
  }

@Override
  protected boolean useMethodInfo() {
    return false;
  }
}
```

Figure 6. Entire Source Code of DIT Plugin

```
public class WmcPlugin extends AbstractClassMetricPlugin {
  private final static String DETAIL_DESCRIPTION;
  Implementation of WMC metric measurement logic
  @Override
  protected Number measureClassMetric(final TargetClassInfo targetClass) {
    int wmc = 0;

    for (final MethodInfo m : targetClass.getDefinedMethods()) {
      wmc += this.measureMethodWeight(m).intValue();
    }

    return new Integer(wmc);
  }

  protected Number measureMethodWeight(final MethodInfo method) {
    int weight = this.measureCyclomatic(method);
    return weight;
  }

  private int measureCyclomatic(final LocalSpaceInfo block) {
    int cyclomatic = 1;
    for (final StatementInfo statement : block.getStatements()) {
      if (statement instanceof BlockInfo) {
        cyclomatic += this.measureCyclomatic((BlockInfo) statement);
        if (!(statement instanceof ConditionalBlockInfo)) {
          cyclomatic--;
        }
      }
    }
    return cyclomatic;
  }
  Overriding APIs for cooperating with the main module
  @Override
  protected String getDescription() {
    return "Measuring the WMC metric.";
  }

  @Override
  protected String getDetailDescription() {
    return DETAIL_DESCRIPTION;
  }

  @Override
  protected String getMetricName() {
    return "WMC";
  }

  @Override
  protected boolean useMethodInfo() {
    return true;
  }

  @Override
  protected boolean useMethodLocalInfo() {
    return true;
  }

  static {
    StringWriter buffer = new StringWriter();
    PrintWriter writer = new PrintWriter(buffer);

    writer.println("This plugin measures the WFC (Response for a Class) metric.");
    writer.flush();

    DETAIL_DESCRIPTION = buffer.toString();
  }
}
```

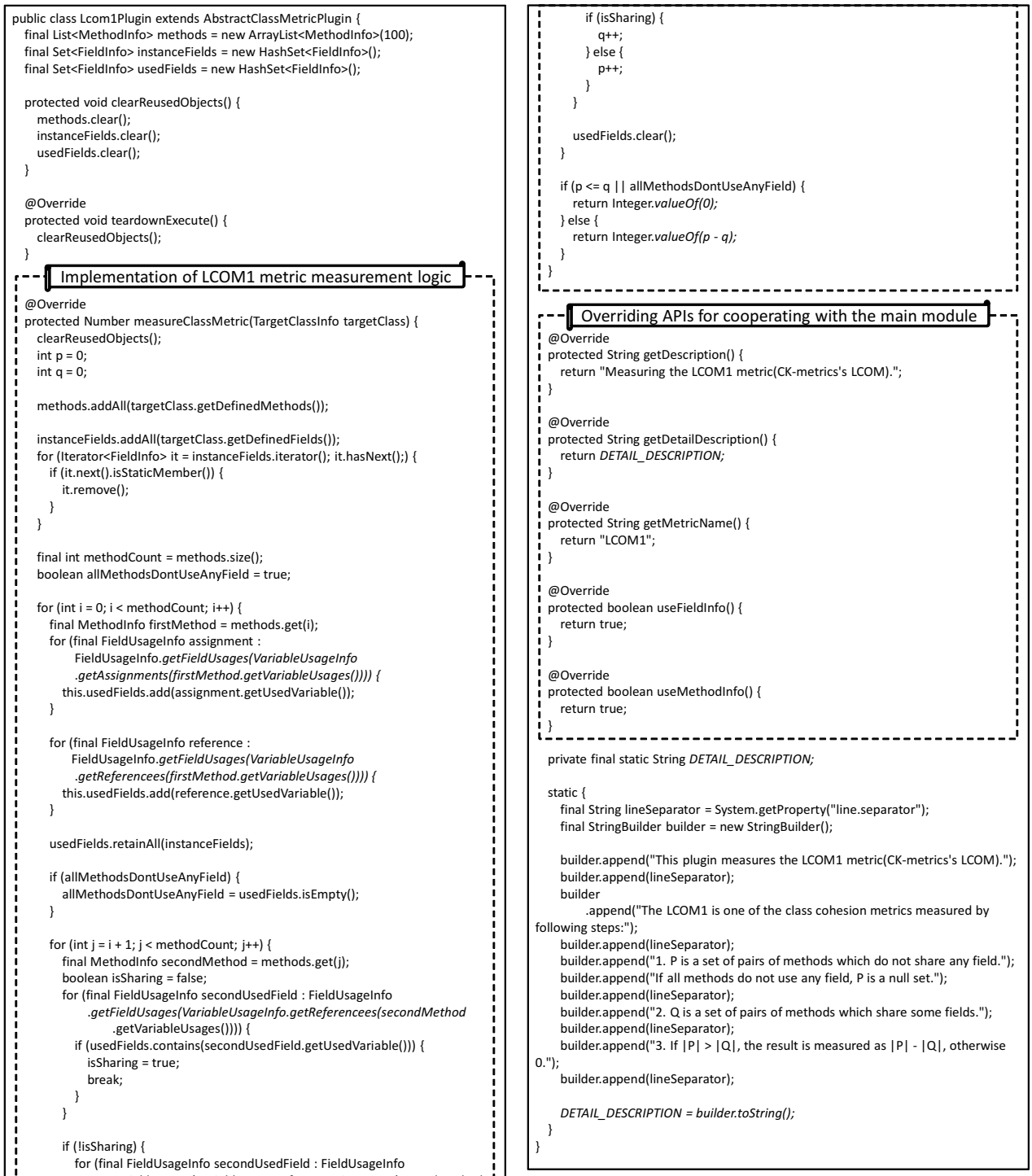
Figure 7. Entire Source Code of WMC Plugin

```
public class NocPlugin extends AbstractClassMetricPlugin {
   private final static String DETAIL_DESCRIPTION;

   Implementation of NOC metric measurement logic
   @Override
   protected Number measureClassMetric(TargetClassInfo targetClass) {
      return targetClass.getSubClasses().size();
   }

   Overriding APIs for cooperating with the main module
   @Override
   protected String getMetricName() {
      return "NOC";
   }

   @Override
   protected String getDescription() {
      return "measuring NOC metric.";
   }

   @Override
   protected String getDetailDescription() {
      return DETAIL_DESCRIPTION;
   }

  @Override
   protected boolean useMethodInfo() {
      return true;
   }

   @Override
   protected boolean useMethodLocalInfo() {
      return true;
   }

   static {
      StringWriter buffer = new StringWriter();
      PrintWriter writer = new PrintWriter(buffer);

      writer.println("This plugin measures the NOC metric.");
      writer.println();
      writer.println("NOC = number of children(subclasses) in a class");
      writer.println();
      writer.flush();

      DETAIL_DESCRIPTION = buffer.toString();
   }
}
```

Figure 8. Entire Source Code of NOC Plugin

```
public class CBOPlugin extends AbstractClassMetricPlugin {

   Implementation of CBO metric measurement logic
   @Override
   protected Number measureClassMetric(TargetClassInfo targetClass) {

      SortedSet<ClassInfo> classes = new TreeSet<ClassInfo>();

      for (final FieldInfo field : targetClass.getDefinedFields()) {
         final TypeInfo type = field.getType();
         classes.addAll(this.getCohesiveClasses(type));
      }

      for (final MethodInfo method : targetClass.getDefinedMethods()) {

         {
            final TypeInfo returnType = method.getReturnType();
            classes.addAll(this.getCohesiveClasses(returnType));
         }

        for (final ParameterInfo parameter : method.getParameters()) {
            final TypeInfo parameterType = parameter.getType();
            classes.addAll(this.getCohesiveClasses(parameterType));
         }

         for (final VariableInfo<? extends UnitInfo> variable : LocalVariableInfo
               .getLocalVariables(method.getDefinedVariables())) {
            final TypeInfo variableType = variable.getType();
            classes.addAll(this.getCohesiveClasses(variableType));
         }
      }

      classes.remove(targetClass);

      return classes.size();
   }

   private SortedSet<ClassInfo> getCohesiveClasses(final TypeInfo type) {

      final SortedSet<ClassInfo> cohesiveClasses = new TreeSet<ClassInfo>();

      if (type instanceof ClassTypeInfo) {
         final ClassTypeInfo classType = (ClassTypeInfo) type;
         cohesiveClasses.add(classType.getReferencedClass());
         for (final TypeInfo typeArgument : classType.getTypeArguments()) {
            cohesiveClasses.addAll(this.getCohesiveClasses(typeArgument));
         }
      }

      return Collections.unmodifiableSortedSet(cohesiveClasses);
   }

   Overriding APIs for cooperating with the main module
   @Override
   protected String getDescription() {
      return "Measuring the CBO metric.";
   }

   @Override
   protected String getMetricName() {
      return "CBO";
   }

   @Override
   protected boolean useFieldInfo() {
      return true;
   }

   @Override
   protected boolean useMethodInfo() {
      return true;
   }
}
```

Figure 9. Entire Source Code of CBO Plugin




```
public class Lcom1Plugin extends AbstractClassMetricPlugin {
   final List<MethodInfo> methods = new ArrayList<MethodInfo>(100);
   final Set<FieldInfo> instanceFields = new HashSet<FieldInfo>();
   final Set<FieldInfo> usedFields = new HashSet<FieldInfo>();

   protected void clearReusedObjects() {
      methods.clear();
      instanceFields.clear();
      usedFields.clear();
   }

   @Override
   protected void teardownExecute() {
      clearReusedObjects();
   }

   @Override
   protected Number measureClassMetric(TargetClassInfo targetClass) {
      clearReusedObjects();
      int p = 0;
      int q = 0;

      methods.addAll(targetClass.getDefinedMethods());

      instanceFields.addAll(targetClass.getDefinedFields());
      for (Iterator<FieldInfo> it = instanceFields.iterator(); it.hasNext();) {
         if (it.next().isStaticMember()) {
            it.remove();
         }
      }

      final int methodCount = methods.size();
      boolean allMethodsDontUseAnyField = true;

      for (int i = 0; i < methodCount; i++) {
         final MethodInfo firstMethod = methods.get(i);
         for (final FieldUsageInfo assignment :
               FieldUsageInfo.getFieldUsages(VariableUsageInfo
               .getAssignments(firstMethod.getVariableUsages()))) {
            this.usedFields.add(assignment.getUsedVariable());
         }

         for (final FieldUsageInfo reference :
               FieldUsageInfo.getFieldUsages(VariableUsageInfo
               .getReferencees(firstMethod.getVariableUsages()))) {
            this.usedFields.add(reference.getUsedVariable());
         }

         usedFields.retainAll(instanceFields);

         if (allMethodsDontUseAnyField) {
            allMethodsDontUseAnyField = usedFields.isEmpty();
         }

         for (int j = i + 1; j < methodCount; j++) {
            final MethodInfo secondMethod = methods.get(j);
            boolean isSharing = false;
            for (final FieldUsageInfo secondUsedField : FieldUsageInfo
                  .getFieldUsages(VariableUsageInfo.getReferencees(secondMethod
                        .getVariableUsages()))) {
               if (usedFields.contains(secondUsedField.getUsedVariable())) {
                  isSharing = true;
                  break;
               }
            }

            if (!isSharing) {
               for (final FieldUsageInfo secondUsedField : FieldUsageInfo
                     .getFieldUsages(VariableUsageInfo.getAssignments(secondMethod
                           .getVariableUsages()))) {
                  if (usedFields.contains(secondUsedField.getUsedVariable())) {
                     isSharing = true;
                     break;
                  }
               }
            }
            if (isSharing) {
               q++;
            } else {
               p++;
            }
         }

         usedFields.clear();
      }

      if (p <= q || allMethodsDontUseAnyField) {
         return Integer.valueOf(0);
      } else {
         return Integer.valueOf(p - q);
      }
   }

   @Override
   protected String getDescription() {
      return "Measuring the LCOM1 metric(CK-metrics's LCOM).";
   }

   @Override
   protected String getDetailDescription() {
      return DETAIL_DESCRIPTION;
   }

   @Override
   protected String getMetricName() {
      return "LCOM1";
   }

   @Override
   protected boolean useFieldInfo() {
      return true;
   }

   @Override
   protected boolean useMethodInfo() {
      return true;
   }

   private final static String DETAIL_DESCRIPTION;

   static {
      final String lineSeparator = System.getProperty("line.separator");
      final StringBuilder builder = new StringBuilder();

      builder.append("This plugin measures the LCOM1 metric(CK-metrics's LCOM).");
      builder.append(lineSeparator);
      builder
            .append("The LCOM1 is one of the class cohesion metrics measured by
following steps:");
      builder.append(lineSeparator);
      builder.append("1. P is a set of pairs of methods which do not share any field.");
      builder.append("If all methods do not use any field, P is a null set.");
      builder.append(lineSeparator);
      builder.append("2. Q is a set of pairs of methods which share some fields.");
      builder.append(lineSeparator);
      builder.append("3. If |P| > |Q|, the result is measured as |P| - |Q|, otherwise
0.");
      builder.append(lineSeparator);

      DETAIL_DESCRIPTION = builder.toString();
   }
}
```

Figure 10. Entire Source Code of LCOM1 Plugin