Reporting the Implementation of a Framework for Measuring Test Coverage based on Design Patterns

Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa
Dept. Computer Science and Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo 1698555, Japan
kazuu@ruri.waseda.jp, washizaki@waseda.jp, fukazawa@waseda.jp

***Abstract*—Fault-free software is highly desirable, and so sufficient software testing plays an important role in attempts to realize a fault-free state. Test coverage is an important indicator of whether software has been tested sufficiently. However, existing measurement tools are associated with several problems, such as the cost of new development, the cost of maintenance, and inconsistent and inflexible measurement. In this paper, we propose a consistent and flexible test coverage measurement framework that supports multiple programming languages. We implemented our framework based on design patterns such as Template Method pattern and Macro Command pattern. Thus we report the success of the implementation of our framework based on design patterns, and we confirm the benefit of design patterns.**

***Keywords*-Design pattern; Framework; Software testing; Test coverage; Code coverage; Metrics**

I. Introduction

Test coverage (code coverage) is an important measure used in software testing. It refers to the degree to which the source code of a program has been tested and is an indicator of whether software has been tested sufficiently. Design pattern is an important software pattern which is a general reusable solution to a commonly occurring problem in software design. Pattern formulates the know-how of solution to a commonly occurring problem to be reused by people. There are multiple levels in test coverage, such as statement coverage, decision coverage and condition coverage. Developers select a suitable level according to the purpose of their software testing[1].

Measurement tools are necessary in order to measure the coverage of various programs accurately, and test coverage measurement tools have become widely available. Many measurement tools are offered for major languages such as C or Java. However, measurement tools for legacy languages such as COBOL and minor languages such as Lua are not readily available and only at some considerable expensive. Moreover, it is more difficult to have access to measurement tools for newly defined languages and for existing languages with some language specification changes because each existing tool is specific to a certain language specification. Such a situation drives the need for the development of some framework or tool that corresponds to a variety of languages including new languages in the future.

In this paper, we propose a consistent and flexible test coverage measurement framework that supports multiple languages. Our framework extracts commonalities among multiple languages, and disregards variability by focusing on the syntax of the languages. We implemented our framework based on design patterns such as Template Method pattern and Macro Command pattern, thus we confirm the benefit of design patterns.

Our framework is now freely available via the Internet[2].

II. Problems in conventional approaches

The following summarizes the problems with existing measurement tools. The problems are in cost of new development, in cost of maintenance, in inconsistent measurement, in inflexible measurement and in Incomplete measurement but we focus only the cost of new development.

The variety of languages is becoming more diverse. Moreover, coverage measurement tools are often unavailable for a number of legacy and/or minor languages due to a lack of community or non-commercial efforts. So, measurement tools for these languages are necessary. A measurement tool consists of the following 4 functions: a syntactic analyzer that interprets syntax from source code, a semantic analyzer that interprets the meaning of syntax such as a statement and a conditional branching, a measurement function for test coverage, and a display function for measurement results. Generally, it is difficult to implement these functions. Therefore, the cost necessary for development is high.

III. Coverage measurement framework supporting multiple programming languages

We propose a test coverage measurement framework that supports multiple languages, and which will solve and alleviate the problems described above.

The framework is a reusable software architecture and provides a generic design as some similar applications. The application can be implemented by adding application-specific code as user code to the framework[4].

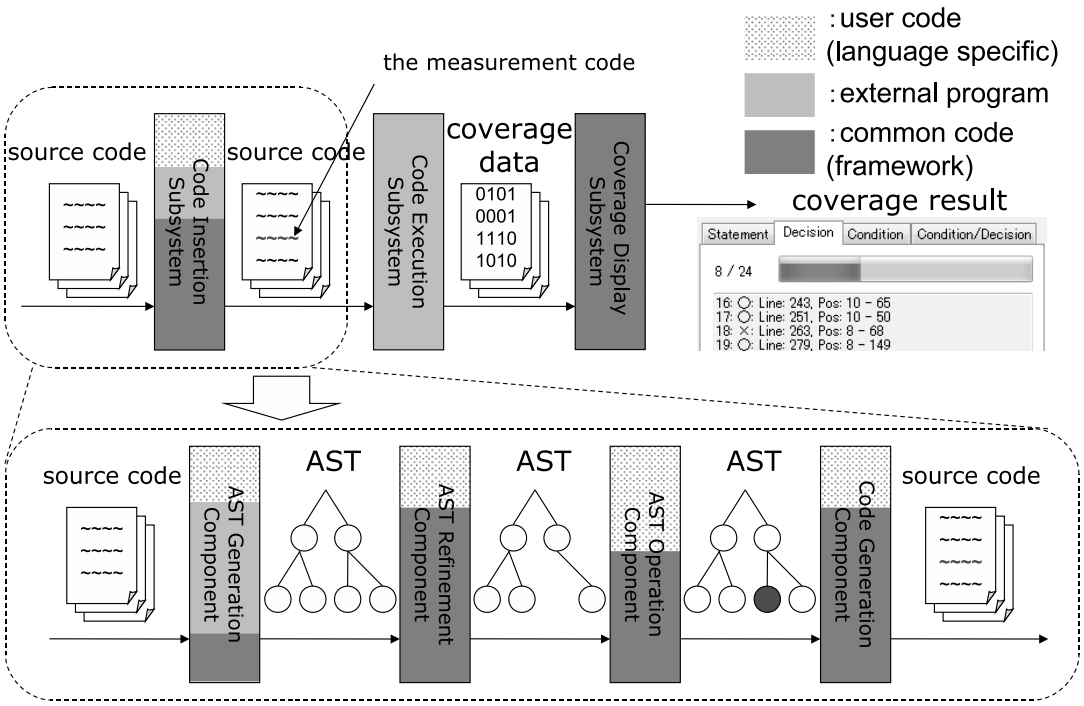

Figure 1. The entire design of our framework

The entire design of our framework and the processing flow is shown in Figure 1. Our framework consists of three subsystems: the code insertion subsystem, the code execution subsystem and the coverage display subsystem. Moreover, the code insertion subsystem consists of four components: the AST (Abstract Syntax Tree) generation component, the AST refinement component, the AST operation component and the code generation component. We implemented their with design patterns, so we get high reusability and reduce the cost of new development.

The process of the coverage measurement is as follows.

1) Generation of AST from source code
2) Insertion of code for measurement on AST
3) Generation of source code from AST
4) Execution of generated source code and collection of measurement information
5) Display of measurement results from test coverage

Our framework inserts the measurement code into the source code, and the test coverage is measured by executing the program. When our framework inserts the measurement code, it collects information such as the location information of the measurement elements in the source code.

Our framework is designed as an object-oriented framework with object-oriented programming and design patterns. Our framework provides common code for language independent processing and also provides structure to help to write user code for language dependent processing. Moreover, the insertion on AST simplifies the insertion processing. In this way, our framework reduces the cost of new development and maintenance. However, our framework targets only procedure-oriented languages due to the mechanism used for measurement which involves inserting the measurement code.

IV. Implementation of our framework

We implemented our framework in .NET Framework 3.5 SP1. Our framework enables the implementation of language specific processing by adding user code such as assembly files that run in .NET Framework 3.5 SP1 or older, or script files in languages supported by Dynamic Language Runtime (DLR)[13]. DLR is .NET library that provides language services for several different dynamic languages. In this way, our framework helps to add user code.

We now show sample code as a sample measurement tool implementation that measures test coverage in Java, C and Python by using our framework.

A. Code insertion subsystem

The code insertion subsystem consists of the following components: the AST generation component, the AST refinement component, the AST operation component and the code generation component.

1) AST generation component: converts the obtained source code into an AST as an XML document. In this sample, this component consists of two functions: AST builder and the caller of AST builder. AST builder is user code which is deployed as an external program. AST builder

is implemented using compilers such as SableCC[5] for Java, ANTLR[6] for C and Python standard library for Python. The caller of AST builder is common code which is designed by using Template Method pattern[7].

The Template Method pattern reorganizes the processing steps between the coarse-grained process flow and fine-grained concrete processing steps. The former is placed in a superclass method and the latter is placed in subclass methods. The latter is triggered by the former by calling superclass abstract methods which are actually implemented in subclasses.

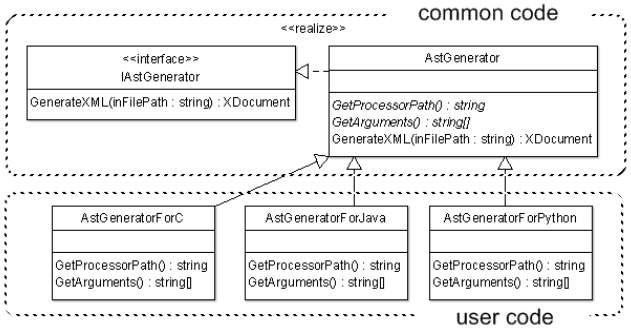

Figure 2. The class diagram of the AST generation component

The class diagram of UML[8] related to this component is shown in Figure 2. The `AstGenerator` is an abstract class that is designed by applying the Template Method pattern. The sample user code of this component for Java is shown in List 1.

List 1. AstGeneratorForJava.cs

```
using System.ComponentModel.Composition;

namespace CoverageFramework.AstGenerator.Java {
  [Export(typeof(IAstGenerator))]
  public class AstGeneratorForJava : AstGenerator {
    private static readonly string[]
        _arguments = new[] {
      "-jar", "../Java/Java.jar",
    };
    protected override string FileName {
      get { return "java"; }
    }
    protected override string[] Arguments {
      get { return _arguments; }
    }
  }
}
```

Therefore, the use of the compiler compilers and common code eases the implementation of this component.

2) AST refinement component: changes the structure of AST to operate AST easily. In the sample, this component removes the unnecessary nodes of AST such as nonterminal nodes which have only nonterminal nodes as child nodes. Moreover, this component converts single-line if statements into multi-line if statements. Our framework provides the almost processing as common code.

3) AST operation component: has roughly three functions: the enumeration of subtrees, the generation of subtrees and the replacement of subtrees. The enumeration function locates the position in which the measurement code is inserted. For example, this function locates the position of all atomic logical terms in conditional expressions in Python. Our framework provides a large part of this function as common code which is designed by using the Template Method pattern.

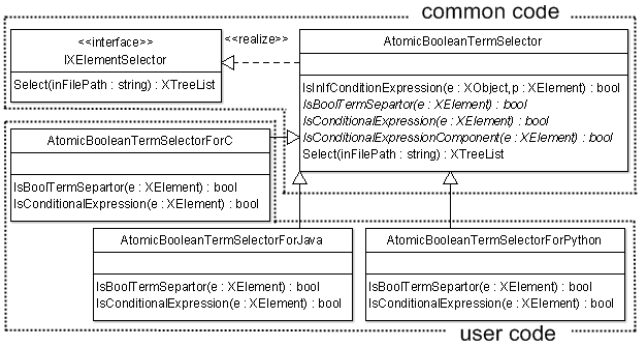

Figure 3. The class diagram of the AST operation component

The class diagram related to this function is shown in Figure 3. The `AtomicBooleanTermSelector` is an abstract class that is designed by applying the Template Method pattern. The sample user code that enumerates the atomic logical terms for Python is shown in List 2.

List 2. AtomicBooleanTermSelectorForPython.cs

```
using System.Linq;
using System.Xml.Linq;
using System.ComponentModel.Composition;

namespace CoverageFramework.Element.Selector.Python {
  [Export(typeof(IXElementSelector))]
  public class AtomicBooleanTermSelectorForPython
      : AtomicBooleanTermSelector {
    private static readonly string[]
      _condComponentNames = new[]
    { "or_test", "and_test", };
    private static readonly string[]
      _condNames = new[]
    { "or_test", "and_test", };
    private static readonly string[]
      _condOpValues = new[]
    { "or", "and", };
    protected override bool
        IsBoolTermSepartor(XElement e) {
      return !e.HasElements &&
        _condOpValues.Contains(e.Value);
    }
    protected override bool
        IsConditionalExpression(XElement e) {
      return _condNames.Contains(e.Name.LocalName);
    }
    protected override bool
        IsConditionalExpressionComponent(XElement e) {
      return _condComponentNames
        .Contains(e.Name.LocalName);
    }
  }
}
```

By implementing processing that judges whether the given

node is the measurement element, this code enumerates the atomic logical terms.

In addition, our framework provides some other classes as common code, such as the `XElementSelectorUnion` class, which integrates some enumeration results, and the `XElementSelectorPipe` class, which enumerates subtrees in other enumeration results. These classes are designed by applying the Command pattern[7].

The Command pattern is the design pattern that encapsulates a request and the parameters in an object. A command object that is combined with certain other command objects is called a Macro Command.

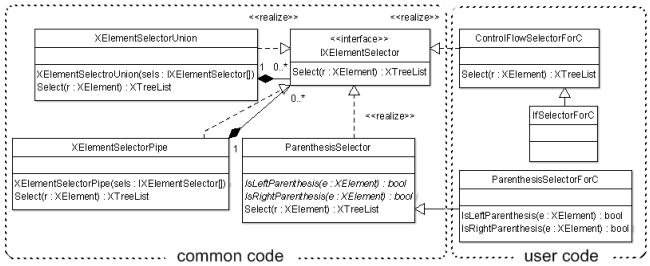

Figure 4. The class diagram according to enumeration subtrees

The class diagram related to the enumeration function is shown in Figure 4. The `XElementSelectorPipe` is a class as a Macro Command by applying the Command pattern. The usage of this class is shown in List 3.

List 3. The usage example of XElementSelectorPipe

```
1  var ifSelector = new XElementSelectorPipe(
2    new IfSelectorForC(),
3    new ParenthesisSelectorForC());
```

By combining the instance of the `IfSelectorForC` class, which enumerates the subtrees corresponding to the conditional sentence, and the instance of the `ParenthesisSelectorForC` class, which enumerates the subtrees corresponding to the parenthetic expression, this code enumerates the subtrees corresponding to the conditional expression for C.

Therefore, our framework reduces the size of the classes and promotes code reuse. Moreover, flexible measurement is achieved by adding processing that locates subtrees.

In addition, the generation functions are used to generate the subtrees corresponding to the measurement code. Our framework requires user code for this function. The replacement functions are used to insert the subtrees of the measurement code into the source code on AST. Our framework provides this function completely as common code.

4) Code generation component: converts the obtained AST into source code. When the AST has memorized almost all of the tokens corresponding text in source code, this component can be implemented simply by adding user code that outputs the tokens as they are without exception. Our framework provides common code that outputs the memorized tokens by applying the Template Method pattern.

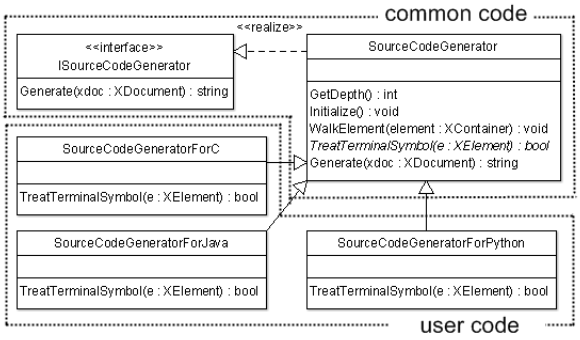

Figure 5. The class diagram of the code generation component

The class diagram related to this component is shown in Figure 5. The `SourceCodeGenerator` is an abstract class that is designed by applying the Template Method pattern. The sample user code of this component for Python is shown in List 4.

List 4. SourceCodeGeneratorForPython.cs

```
1  using System.Xml.Linq;
2  using System.ComponentModel.Composition;
3
4  namespace CoverageFramework.CodeGenerator.Python {
5    [Export(typeof(ISourceCodeGenerator))]
6    public class SourceCodeGeneratorForPython
7        : SourceCodeGenerator {
8      protected override bool
9          TreatTerminalSymbol(XElement element) {
10       switch (element.Name.LocalName) {
11       case "NEWLINE":
12         WriteLine();
13         break;
14       case "INDENT":
15         Depth++;
16         break;
17       case "DEDENT":
18         Depth--;
19         break;
20       default:
21         return false;
22       }
23       return true;
24     }
25   }
26 }
```

Neither the linefeed nor the indent is memorized in AST for Python. Accordingly, this component requires user code to output the linefeed and the indent to the corresponding terminal nodes.

Therefore, in our framework, most of this component is common code.

V. Evaluation

We evaluate our framework by comparing sample implementation that is developed by using our framework

with typical existing measurement tools, namely, Cobertura supporting Java and Statement coverage for Python[9] supporting Python.

We evaluate the new development cost by using the LOC (Lines of Code) of the program that inserts the measurement code and by the number of supporting coverage levels.

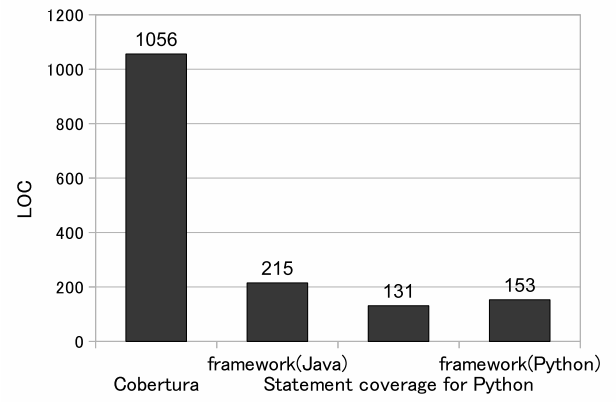

Figure 6. The LOC

Figure 6 shows the comparison results obtained with LOC. The LOC of Cobertura is 1056 lines, and the LOC of the sample implementation for Java is 215 lines. Cobertura uses BCEL[10] to insert byte code into the Java class file. BCEL is a library that conveniently provides users with the option to analyze, create, and manipulate (binary) Java class files. However, the sample implementation does not use the library except for our simple helper methods and the .NET standard library.

On the other hand, the LOC of Statement coverage for Python is 131 lines, and the LOC of the sample implementation with our framework for Python is 153 lines in Figure 6. Statement coverage for Python uses only the Python standard library. In addition, the LOC of the language independent and reusable part in the framework is 654 lines. Our framework can support new language with less cost than new development of the measurement tool.

Cobertura supports statement coverage and decision coverage, and Statement coverage for Python supports only the statement coverage. On the other hand, the sample implementation with our framework supports statement coverage, decision coverage, condition coverage and condition/decision coverage. The same functionality can be implemented with fewer LOCs.

Therefore, our framework succeeded in alleviating the problem of high cost for new development using design patterns and we confirm high reusability of design patterns.

VI. Related work

Here, we explain the ideas of Kiri et al.[12] as other researches that relate to the mechanism and purpose of our framework.

Kiri et al. propose the idea of developing a measurement tool which inserts the measurement code into source code. Their idea measures statement coverage, decision coverage and a special coverage called RC0. RC0 is special statement coverage for only the revised statement. However, their idea can measure only statement coverage and decision coverage because their idea measures coverage by simply inserting a simple statement. Moreover, though their idea can measure the coverage of four languages, including Java, C/C++, Visual Basic, and ABAP/4, it cannot support any other languages. Conversely, our framework cannot measure RC0. However, our framework can support new coverage such as RC0 easily by adding user code.

VII. Conclusion and future work

In this paper, we have proposed a coverage measurement framework for multiple languages and report the implementation of our framework based on design patterns. We achieved reduction of cost by reusing common code because we implemented our framework based on design patterns. Thus we conclude design patterns produce high reusability.

We plan to evaluate more completely our framework, achive more reusability using design patterns, and improve the framework in order to support languages other than procedure-oriented languages, such as functional programming languages.

References

[1] Lee Copeland, ”A Practitioner’s Guide to Software Test Design”, Artech House, 2003.

[2] Kazunori Sakamoto, Open Code Coverage Framework, http://sourceforge.jp/projects/codecoverage/.

[3] Cobertura, http://cobertura.sourceforge.net/.

[4] Mohamed Fayad and Douglas C. Schmidt, ”Object-Oriented Application Frameworks”, the Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[5] Etienne M. Gagnon, Ben Menking, Mariusz Nowostawski, Komivi Kevin Agbakpem and Kis Gergely, SableCC, http://sablecc.org/.

[6] ANTLR, http://www.antlr.org/.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, ”Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994.

[8] OMG, Unified Modeling Language (UML) specification, version 2.2, http://www.omg.org/spec/UML/.

[9] Gareth Rees, Statement coverage for Python, http://garethrees.org/2001/12/04/python-coverage/.

[10] Apache Software Foundation, The Byte Code Engineering Library, http://jakarta.apache.org/bcel/.

[11] Haruhiko Okumura, Houki Satoh, Kazuo Turu, Kazuyuki Shudo and Tutimura Nobuyuki, ”Algorithm cyclopedia by Java”(in Japanese), Gijutuhyoronsya, 2003.

[12] Takashi Kiri, Tatuya Miyoshi, Satoru Kishigami, Tatuo Osato, Tuyoshi Sonehara ”About the source code insertion type coverage tool”, The 69th Information Processing Society of Japan National Convention, 2003.

[13] Microsoft, http://dlr.codeplex.com/.