

Using Metaprogramming to Implement a Testing Framework

Hyun Cho

University of Alabama at Birmingham
Department of Computer and Information Sciences
Birmingham, AL 35294

robusta@cis.uab.edu

ABSTRACT

Testing plays an important role in improving software quality. However, test preparation, execution, and report analysis are labor-intensive tasks that require different activities and methods to obtain benefit from the testing process. This poster describes an investigation into the use of metaprogramming, which can reflect the internals of a system, to implement a common test framework that can be used to reduce the burden for test preparation in unit and system testing. The framework focuses on generating instrumented Java bytecode based on several sources of input (e.g., source code or bytecode, and information provided by each test case). Both load-time and compile-time metaprograms are used within the framework to inform the testing process.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging]: Testing tools

General Terms

Measurement, Design, Verification.

Keywords

Testing Framework, Metaprogramming, Transformation

1. INTRODUCTION

Software testing has played a major role in improving software quality. Although no one testing strategy has emerged yet, the strategy normally follows three steps: unit testing, integration testing, and system testing [1]. Unit testing is a popular form of white-box testing where developers write test cases with the understanding of internal logic and structure of the codes they have written. JUnit [2], CUnit [3], and CPPUnit [4] are commonly used for this purpose. Integration testing is a gray-box technique that focuses on finding defects during system integration. System testing usually takes the form of a black-box test that finds defects by running a system manually or using internally developed testing tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACMSE'09, March 19–21, 2009, Clemson, SC, U.S.A.

Copyright 2009 ACM 1-58113-000-0/00/0004 ...\$5.00.

The various types of testing may use different tools and methods, but they share common challenges and concerns. First, it is often required to instrument existing code to provide detailed information during testing. However, the challenge is that the instrumentation may need to be performed manually (e.g., embedded systems areas that may lack strong tool support), which takes much time and is error-prone. For example, in code coverage analysis and performance testing, additional code may be added as probes to different locations in the original source code of an application. Second, different tools and methods are often used for unit and system testing. JUnit is a standard popular unit testing tool for Java applications, but there is no such tool for system testing. Thus, an organization must duplicate their investment of time and effort to maintain different types of testing tools and training their engineers. As a third challenge of testing, test cases may be designed poorly, which negatively impacts the testing process and the quality of the tested software.

In this poster, a functional test framework that relies on metaprogramming is presented as a possible solution toward resolving these issues. Section 2 introduces concepts of metaprogramming and Section 3 describes the design of the test framework. Concluding remarks are presented in Section 4.

2. Background and Related Works

JUnit is widely used for unit testing Java applications, but it requires understanding source code to write test cases. JUnit originated from structural testing concerns and has shortcomings with respect to accommodating the increasing demands of functional testing, such as flexible test configuration and parameterization. To test behaviors of a unit during unit testing, Ribeiro et al [5] developed a Java bytecode instrumentation technique, but this method requires profound knowledge of Java bytecode. JFunc [6] extends JUnit to test the behavior of a system in a similar fashion to JUnit, but it has similar challenges with respect to functional testing, e.g., implementing test codes. In addition to unit testing, there is often a need to instrument source code for test coverage and performance analysis.

A metaprogram is a program that can make structural and/or behavioral changes to another program. A metaobject can be attached to classes and methods to adapt the behavior of a class at either compile-time or run-time. With metaprograms, it is possible to retrieve information about classes, methods, variables, or control structures without inserting additional code manually.

There exist two types of metaprograms for Java: compile-time and load-time. A compile-time metaprogram provides class metaobject APIs, which allow programmers to handle source code as language constructs. OpenJava [6] can be used to write a compile-time metaprogram. A load-time metaprogram manipulates Java bytecode to reflect a system's behavior during run-time. Javassist [8] and JMangler [9] are examples of load-time metaprogramming that provide libraries to manipulate Java bytecode without knowledge of its structure.

3. Implementation of Testing Framework

This poster describes a testing framework that uses both compile-time and load-time metaprograms to process an application transparently, regardless of input form (e.g., bytecode or source code). Figure 1 illustrates the architecture of the testing framework based on metaprograms. This framework is comprised of a Parser, Test Case Designer, Transformer, and Test Manager. In the framework, the Parser and Transformer exploit metaprogramming techniques to analyze inputs and generate instrumented test outputs.

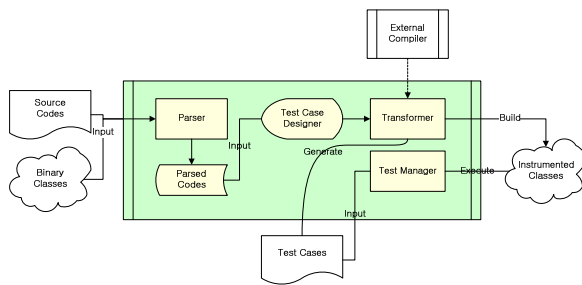


Figure 1. Architecture of Test Framework

The Parser (input processor) implementation relies on metaprogramming APIs available in OpenJava and JMangler, which are compatible to Java Reflection APIs [10], to extract information such as classes, methods and attributes from the input source. The type of Parser is automatically determined by looking at the extension of inputs (e.g., .java for source code and .class for bytecode). The Parser can selectively extract information using metaprogramming so that it helps engineers to understand the system under test by limiting or expanding information per their needs. The resulting Parsed Code is passed to the Test Case Designer, which is used to design test cases and specify target classes and the methods to be instrumented. Figure 2 illustrates how the Test Case Designer is used to enter values for arguments and the expected results of a test execution of a specific method.

Each row in the Test Case Designer represents a test case and the number of test cases can be governed after analyzing a test report. After the test case is designed, the Transformer loads each class file, retrieves its bytecode, and attaches metaobjects to produce run-time information. Before attaching metaobjects, the Transformer checks the potential conflicts among the selected test options. For example, in Figure 2, the Sub method has two test options: C/T for coverage and trace, and P for performance. Because the C/T option could affect measuring performance of the method, the Transformer generates two different instrumented codes for coverage and performance. If the input is source code, the Transformer launches external compilers for attaching metaobjects to perform compile-time adaptation. Finally, the Test

Manager processes the test cases and controls the whole test execution by referring to test options specified in the test cases. The Test Manager is integrated with the Test Case Designer.

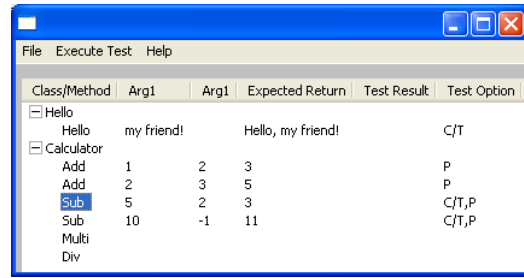


Figure 2. Test Case Designer

4. Conclusion

This poster describes how metaprogramming can assist in the construction of a test framework (e.g., unit test and system test). The Parser and Transformer play key roles in this framework as a potential solution to the challenges addressed in the introduction by using both compile-time and load-time metaprograms to service their functionality transparently. The Transformer generates instrumented test code by attaching metaobjects so that instrumentation takes less time and helps to produce quality instrumented code.

The implementation of the test framework is a prototype that needs additional experimental evaluation. Although test cases have been created and effectively used on small applications, additional evaluation is needed. The future plans include applying the test framework to a large open source application, such as JBoss or other system software written in Java. An additional future work will explore ways to reduce the amount of manual effort involved in specifying parameters in the Test Case Designer.

5. REFERENCES

- [1] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2007.
- [2] <http://www.junit.org/>
- [3] <http://cunit.sourceforge.net/>
- [4] <http://apps.sourceforge.net/mediawiki/cppunit/>
- [5] Ribeiro, J. and Relá, M. Z., "Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing," *Workshop on Testing and Fault Tolerance*, held at the *Brazilian Symposium on Computer Networks and Distributed Systems*, Belém, Brazil, May 2007, pp. 143-156.
- [6] <http://jfunc.sourceforge.net/>
- [7] Tatsubori, M., Chiba, S., Killijian, M., and Itano, K., "OpenJava: A Class-Based Macro System for Java," *Reflection and Software Engineering*, Denver, CO, November 1999, pp. 117-133.
- [8] Chiba, S., "Load-time Structural Reflection in Java," *European Conference on Object-Oriented Programming*, Cannes, France, June 2000, pp. 313-336.
- [9] Kniessel, G., Costanza, P., and Austermann, M., "JMangler: A Framework for Load-Time Transformation of Java Class Files," *International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001, pp. 100-110.
- [10] Forman, I. and Forman, N., *Java Reflection in Action*, Manning Publication, 2004.