Compile-Time Checked Reflection API

Gabriele Kahlout
Free University of Bozen
g@dp4j.com

ABSTRACT

When using the Reflection API to access private members, there is enough information at compile-time to eliminate common runtime errors resulting from the incorrect use of the API. This paper presents dp4j as a tool for injecting the Reflection API code at compile-time while hiding it from the developer and thereby limiting runtime errors to an eventual subset of dp4j bugs.

Keywords

Reflection API, testing, refactoring, annotations processing, Java

1. INTRODUCTION

When testing the correct functioning of a computer program, testers sometimes need to access private members of other classes from their test methods[1]. Scenarios include:

- Testing private methods executed only at start-up or setup time with parameters received from the main method; testers might want to simulate the start of the program with particular parameters verifying that each of those methods indeed initializes variables and behaves as expected in certain test-simulated circumstances. Because those methods should not be called from other classes nor should the initialized fields be accessible to other classes, it is desirable to keep restrictive access modifiers;
- Setting the value of fields that obtain sensible values through the normal execution that are, however, not suitable in a testing context; such fields might be time interval constants that are too slow to tolerate during test simulations or references to external resources. Because those fields need not be variable in the production system, it is desirable not to alter their access restrictions or re-design the program such that they might be readily modified.

Those testers must, however, deal with access restrictions built into object-oriented programming languages that inhibit access to non-public members of the classes[2] that they intend to test.

Existing solutions include:

- Identifying and calling public methods that access the non-public members as required in the test, and yet do not have side effects on it[3][4] (idempotent with respect to the test);

This solution is not always viable because there might not be any such public method. It might also not always be obvious when a method is idempotent, and could indeed become a source or errors and frustration for the tester.

- Changing the access modifier of the member such that it could be accessed from the test; `@VisibleForTesting` is an annotation from Google used to indicate that the visibility of a member has been relaxed for testing[5];

This solution is the easiest to implement because it eliminates the problem but possibly introduces many others that jeopardize the stability of the program; it affects production code, instead of being limited in scope to the tests. If the member visibility were restricted to inhibit access from client classes, then disabling the language built-in access restrictions would undermine the program stability and could lead to malfunctioning. If the member visibility were restricted just as a safe practice of encapsulation, the risk of problems would be mitigated to those of breaking encapsulation[6].

- Re-design the program or test code to eliminate the problem** (also referred to as making the code more testable) by testing at broader levels[7];

This solution also eliminates the problem, but, unlike relaxing access modifiers, it should not lead to instable programs. It does also require changing production code, which might not be possible for the testers (as clients) or it may just be infeasible.

- Put the test class in the same package as the class tested, or even put the tests in the same class[8].

This solution does not separate the program code from the source code and might require adding test-only dependencies and other test classes to the package if the test class depends on them.

- Using the Reflection API[9] to temporarily set the member as accessible during the test execution[10].

---

1 Stéphane Barbey and Alfred Strohmeier: The problematics of testing object-oriented software. In Proceedings of the Second Conference on Software Quality Management (SQM '94), Edinburgh, UK

2 Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)

3 Yoonsik Cheon and Gary T. Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the 16th European Conference on Object-Oriented Programming* (ECOOP '02), Boris Magnusson (Ed.). Springer-Verlag, London, UK, 244

4 **Stefan Wappler and Ina Schieferdecker. 2007. Improving evolutionary class testing in the presence of non-public methods. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (ASE '07). ACM, New York, NY, USA, 381-384.

5 http://www.webcitation.org/5xvpvpBj6

6 Alan Snyder. 1986. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications* (OOPLSA '86), Norman Meyrowitz (Ed.). ACM, New York, NY, USA, 38-45.

7 Matt Stephens and Doug Rosenberg (2010). *Design Driven Testing: Test Smarter, Not Harder*. Apress. 247.

8 Dwight Deugo (2000). *More Java gems*. Cambridge University Press. 359.

9 *http://www.webcitation.org/5z4l7SllG

10 Liangliang Kong; Zhaolin Yin; , "The Extension of the Unit Testing Tool Junit for Special Testings," *Computer and*

DOI: 10.5176_2010-2283_1.3.90

This solution has the advantage of not changing production code and being viable (unless running under a security manager*) when access through public methods is not possible. Yet the Reflection API is not straightforward and requires bloating the testing method with boilerplate lines of code to access the member and handle runtime exceptions that result from incorrect use of the API*. This could be attributed to the API's ability to access members not yet known at compile-time for which the API needs to receive enough parameters to identify them in the Bytecode at runtime.
Mocking frameworks and wrapper libraries[11] only helps to reduce the complexity of using the Reflection API, but the developer must still, for example, pass the member name as a String.

This paper discusses a tool that generates the necessary Reflection API through static code analysis. In this way, it allows testers to access inaccessible members as if they were accessible in their tests without changing production code or coding against any API.

2. INJECTING REFLECTION AT COMPILE-TIME

The idea is to have a CASE tool statically analyze the testing code for 'illegal' access to inaccessible members and replace it with the corresponding reflected legal access code, sparing the tester the burden of writing the reflection code.

Injecting the reflected code only in the AST[12] that is used for compiling (as opposed to modifying the source file) means that the reflected code would be generated at each compilation of the source code. This enables developers to change the production code without having (testers) manually update the reflected code. Moreover, it avoids bloating the test method with boilerplate code and limits the cost of learning the Reflection API to applications that expect run-time parameters.

2.1 Java Implementation

Annotation processing[13] in Java is part of the compilation process, where annotation processors[14] access the AST[15]. Then, the processor can analyze the source code and modify the AST with the Compiler Tree API[16] before bytecode generation.

The diagram below shows how the source code, on the left, is parsed into an AST that annotations processors receive; an annotation processor may either write new source code, restarting the compilation process, or modify the AST before it is passed on to generate the binary code from it.

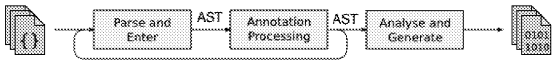


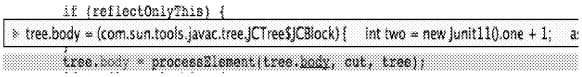
Figure 1. Diagram of the compilation process in the Java Compiler, adapted from the OpenJDK Compilation Overview Diagram

Hence, dp4j declares an annotation processor that processes JUnit[17] and TestNG[18] `@Test` annotation as well as its own `@Reflect` annotation for general-purpose use. With dp4j's processor in the CLASSPATH, Javac passes to it the AST of the program sources[19]; the processor then replaces 'illegal' accesses with the corresponding reflected legal access code, and then passes on the modified AST for further processing by other annotation processors or to bytecode generation.

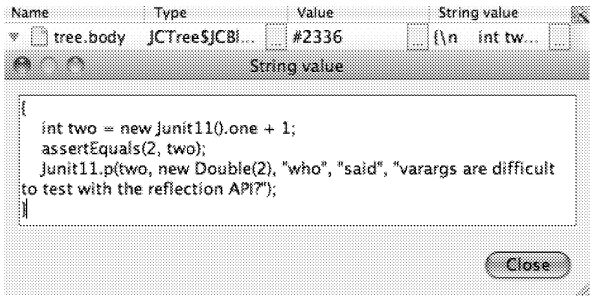
2.2 IDE Integration

Injecting implementation code directly in the AST trades transparency for increased readability and maintainability. In fact, this lack of transparency makes debugging methods with injected code more difficult because injected lines of code will not be visible. Furthermore, IDE editors provide no explicit support for displaying or debugging injected code whereas support for injected code in general is at least a new additional burden on IDE editors, and might frustratingly not always work.

To lessen the lesser transparency, it is here proposed that when hovering over a method in an IDE editor while in debugging mode, the AST code (including the injected code) is displayed. IDE users are used to this IDE interface, as shown below.



Screenshot 1. Shows a flyover popup in NetBeans displaying the toString() return of a hovered-over variable.



Screenshot 2. Shows the same information displayed in the previous screenshot but in a larger window.

Such IDE support would complement Lombok's delombok tool, which creates a copy of the sources replacing all Lombok's annotations with the injected code[20], and dp4j's verbose parameter, which prints a copy of sources with the injected code to the standard output.

Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on , vol.2, no., pp.410-415, 20-24 June 2006

[11] https://sourceforge.net/projects/privaccessor

[12] http://www.webcitation.org/5z4lE8jmh

[13] http://www.webcitation.org/5z4lFuTPm

[14] http://www.webcitation.org/5z4lHLE40

[15] http://foldoc.org/Abstract+Syntax+Tree

[16] http://download.oracle.com/javase/6/docs/jdk/api/javac/tree/index.html

[17] http://junit.sourceforge.net/javadoc/org/junit/Test.html

[18] http://testng.org/javadoc/org/testng/annotations/Test.html

[19] http://download.oracle.com/javase/6/docs/technotes/tools/windows/javac.html#processing

[20] http://projectlombok.org/features/delombok.html

3. MOTIVATING EXAMPLE: TESTING THE SINGLETON

Consider a Singleton implementation[21] that initializes one of its fields with a database-related constant value, as shown in the following code snippet.

```
@Singleton
public class ContentProducer extends
MeaningsDatabase {
...
public class Database extends Observable {
    private static final String defPU = ...
```

Code Snippet 1. Shows the class declaration of ContentProducer Singleton class and its superclass Database which inline-initializes a private constant value defPU. `ContentProducer` is a Singleton class from MemorizEasy source code[22].

A test that sets with Reflection `defPU` with a test value and then sets the Singleton `instance` field to a new instance that will then use the new value of `defPU` follows.

```
@Test
public void testDeleteUser() throws
MultipleMengsException,ClassNotFoundExceptio
n,NoSuchFieldException,IllegalAccessExceptio
n,NoSuchMethodException,InvocationTargetExce
ption,InstantiationException{
    ...
    Field defPUField =
Database.class.getDeclaredField("defPU");
    defPUField.setAccessible(true);
    Field modsField = Field.class.
.getDeclaredField("modifiers");
    modsField.setAccessible(true);
    modsField.setInt(defPUField, 10);
    defPUField.set(null, testVal);
    Constructor cpCon =
ContentProducer.class.getDeclaredConstructor
();
    cpCon.setAccessible(true);
    Field instanceField = null;
ContentProducer.class.getDeclaredField("inst
ance");
    instanceField.setAccessible(true);
    modifiersField.setInt(instanceField,
10);
    instanceField.set(null,
(ContentProducer) cpCon.newInstance());
...
```

Code Snippet 2. Shows a test method using Reflection to change the values of inherited private static final field defPU, and private static final field instance. The highlighted code is the code where the changes actually occur.

The same written with dp4j.jar in the CLASSPATH follows:

```
@Test
public void testDeleteUser() throws
MultipleMengsException{
    ...
    Database.defPU = testVal;
    ContentProducer.instance = new
ContentProducer();
```

Code Snippet 3. Test method accessing a private constructor from ContentProducer as if it were public.

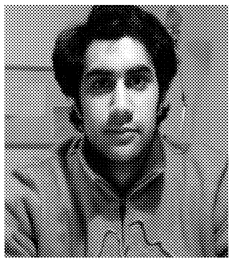
The above code snippets demonstrate how much boilerplate code the tester could be saved from writing using dp4j; it is also worth noting the opportunities to avoid error when not having to pass the field names "`defPU,`" "`modifiers,`" and "`instance`" as Strings, and having to call `setAccessible(true)` on each reflected member before accessing it; as remarked earlier, rename refactoring those members would result in broken the tests whereas, with dp4j, this would not occur because the reflection code would be generated anew at compile-time using the new member names.

4. CONCLUSION

This paper discussed the application of annotation processors to inject boilerplate code, such as the Reflection API, and transparency issues related to it. The contributions include: claiming a novel application of annotation processors to inject the Reflection API, demonstrating usage with sample code that accessed the private fields and constructor of a Singleton, and illustrating an IDE integration proposal to overcome transparency issues of code injection.

5. FUTURE WORK

All of Java's syntax is not yet supported by dp4j. It also only supports the standard Javac but not the Eclipse Compiler that is used by default in the Eclipse IDE[23].



Gabriele Kahlout received his B.S. degree in computer science from the Free University of Bozen in Italy. His research interests span declarative programming, information retrieval, computer-aided learning, and machine translation.
With help from the TIS Innovation Park, he is currently working on a language-learning application that he used to quickly learn both English in Qatar and German in Italy (South Tyrol).

[21] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. AddisonWesley Professional, 1994.)

[22] www.memorizeasy.com

[23] https://bugs.eclipse.org/bugs/show_bug.cgi?id=341842#c4