Using Aspect-Oriented Programming for mutation testing of third-party components

Macario Polo Usaola

Institute of Information Systems and Technologies
University of Castilla-La Mancha
Ciudad Real, Spain
macario.polo@uclm.es

Abstract. This paper presents a new approach for testing third-party components using aspect-oriented programming. The idea consists in developing mutation operators as pointcuts that trap the code under test and manipulate its behavior.

Keywords. Testing, mutation, component testing, aspect-oriented programming

1 Introduction

Mutation is a testing technique traditionally applied to evaluate the quality of test suites. A mutant is a copy of the program under test that holds a small syntactic change that can be considered as a simple fault. The goal of test case execution is to discover all these "artificial" faults that have been introduced in the mutants.

Mutants are usually produced by automated tools that systematically apply well-defined rules (called *mutation operators)* to the program under test. Each operator is specialized in modifying a certain type of statement (such as substituting an arithmetic operator by another) and tries to imitate usual faults committed by programmers [1]. Thus, discovering these faults is almost as discovering the faults that a *competent programmer* could have introduced. Once a set of mutants is available, the tester must write test cases that discover the faults inserted in the mutants (i.e., that *kill* the mutants): if non-equivalent mutants remain *alive*, then new test cases must be added to the suite until it becomes *mutation-adequate*, what means that the test suite has killed all the non-equivalent mutants. In fact, mutation testing is mainly used to evaluate the quality of test suites: the more artificial faults are found, the higher the quality of the test suite. This quality is measured with the *mutation score* (**Fig. 1**), which is the quotient of the *killed mutants* (mutants whose fault has been found) divided by the number of non-equivalent mutants.

$$MS(P,T) = \frac{K}{M - E}$$

,where:
P = program under tests
T = test suite
K = Number of killed mutants
M = Total number of mutants
E = Number of equivalent mutants

Fig. 1. Mutation score

The results of the execution of test cases against the mutants can be represented in a *killing matrix*, which shows the mutants killed by test cases: a cell with an X in **Fig. 2** evidences that the test case in the top row has killed the mutant in the left column. An empty cell represents that the corresponding test case has not found the fault inserted in the corresponding mutant. Thus, *tc4* is an ineffective test case, since it does not find any fault; *tc6* is a very good test case because it finds 5 of the 7 faults inserted; supposing *m7* is not equivalent, it represents a very hard-to-kill mutant, since no test case in the suite finds it; on the contrary, *m2* is a "bad" change, since it is found by almost all the test cases.

An assumption that can be made is that, if a mutation-adequate test suite discovers all the artificial faults inserted in the mutants and does not discover any fault in the original system, then it is very likely that the program under test is almost correct [2].

Thanks to the contributions of many researchers, mutation has evolved since 1978, when DeMillo et al. described the technique in a seminal paper [1]. Besides many other important results and advances, **Fig. 3** shows how mutation has evolved with respect to the testing level: in its first years, it was applied to the discovering of faults in single program functions or procedures, being Mothra [3] one of the first tools. Later, Ma and Offutt [4] presented the MuJava tool, which enabled mutation testing for classes and, after its publication and deployment, was applied in many research studies. MuJava implemented a wide set of mutation operators both traditional (i.e., defined to be applicable to statements present in almost any programming language) and others specific for object-oriented programs. In parallel, Delamaro et al. [5] presented Proteum, a tool for mutation testing of C programs that had some operators for integration testing. More recently, the Bacterio tool [6] implements, besides traditional and integration operators, a specific method of compiling and linking to allow the testing of big Java applications at the integration level.

		Test cases					
		tc1	**tc2**	**tc3**	**tc4**	**tc5**	**tc6**
Mutants	***m1***	X	X				
	m2	X	X	X		X	X
	m3		X				X
	m4			X			X
	m5			X			X
	m6			X			X
	m7						

Fig. 2. A killing matrix

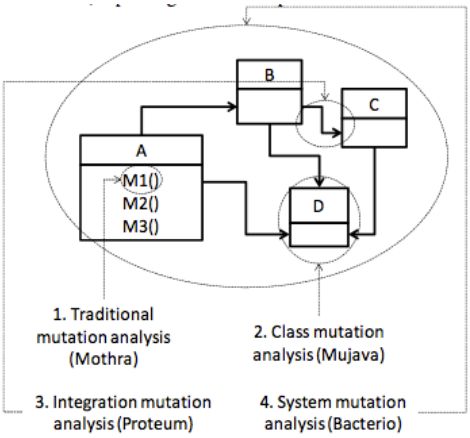


Fig. 3. Different levels of mutation testing

Integration and system testing are respectively concerned with the behavior of modules and systems when they are connected to others. Commonly, these connections and communications are carried out using third-party components (i.e., API's or

library functions) in the middle, in whose behavior the developer relies. These components are directly linked to the system and used "as they are". Due to the usual unavailability of the source code and to its typical deployment in the form of packaged libraries, it is common that users do not test them at all. So, most works related to mutation testing of third-party components or systems take advantage of their interfaces: modification of the parameter values passed, data perturbation, etc.

An unexplored approach, which is applicable for interpreted languages (such as ASP, Java, Perl, PHP or Ruby), is the use of the dynamic weaving of Aspect Oriented Programming (AOP) to modify the behavior of external libraries and components in a controlled way, inserting faults as in traditional mutation testing. Whilst mutation testing of AOP has received attention, the use of Aspects as a tool to generate mutants has not.

In a previous article [7] we described 48 mutation operators for integration and system levels. This work is concerned with the application of AOP to mutation testing, especially at these testing levels. Unless one exception that will be analyzed in next section this is, to our best knowledge, the first work dealing with this topic. The paper explores the possibilities of aspects to be applied for mutation testing.

2 Related work

First of all, this section makes a brief overview of AOP by showing its main concepts and drawing its possibilities for mutation testing. In a second subsection, previous works on aspects and mutation are reviewed. Then, a revision of the most relevant research on the application of mutation to integration testing is presented. Finally, we present some reflections about mutation and system testing.

2.1 Overview of aspect-oriented programming

In aspect-oriented programming, aspects are used to highlight the pieces of the code related to some kind of functionality. An aspect consists of one or more *pointcuts*, and a *pointcut* represents the places in a program whose behavior needs to be logged or modified. **Fig. 4** shows the AspectJ header of a pointcut that catches all the calls to the *write(byte[], int, int)* method included in the *java.io.FileOutputStream* class. The *around* keyword instructs the execution engine to substitute the original method body by the code between brackets, which is called the *advice*.

```
void around(java.io.FileOutputStream caller,
    byte[] array, int off, int len) :
     target(caller) && args(array, off, len) &&
     call(* FileOutputStream.write(byte[], int, int)) {
   byte[] theArray=new byte[array.length-1];
   for (int i=1; i<array.length; i++) {
       theArray[i-1]=array[i];
   }
   array=theArray;
   proceed(caller, array, off, len);
   }
```

Fig. 4. Implementation of a pointcut

In a single aspect there may be several pointcuts: for example, an aspect specialized on the *FileOutputStream* class may contain pointcuts to catch calls to the methods *write(byte[], int, int), write(byte[]),*

write(int), close(), etc. Another aspect specialized in persistence may trap all calls to *insert* and *update* methods in order to perform, for example, a previous checking the grants of the user.

The pointcut body (i.e., its advice) may act on any of the parameters passed (including the *target* object, represented in **Fig. 4** by the *caller* parameter) in order to modify the regular behavior of the method. The boldfaced code, for example, removes the first byte contained in the *array* parameter. Then, the original *write* method is called (*proceed* keyword), but receiving the modified *array*.

In interpreted languages, the code affected by aspects may be executed in two different ways:

1) With *static weaving*, the code is modified at compilation time: the pointcut implementation of **Fig. 4** should generate a new version of *FileOutputStream* with a modified version of *write(byte[], int, int)*. However, for functions packaged in library files, this kind of weaving is not viable.
2) With *dynamic weaving*, the program code remains with no changes. At runtime, an execution engine observes the execution of the program and applying the corresponding pointcuts to those fragments of the code that match the pointcut header.

Besides with *around* (that substitutes the call), AspectJ (the most popular AO framework for Java) pointcuts can be established *before* or *after* method calls. There exist many other modifiers to catch code [8].

2.2 Mutation testing and AOP

Mutation and aspects can be studied from two points of view:

1) Since pointcuts allow code modifications, aspects can be used as a tool to generate and execute mutants.
2) Since AOP introduces some new constructors that complement those of traditional object-oriented software, specific mutation operators for AOP can be developed.

With respect to the first line, we have found only a research group working on it: Bogacki and Walter [9][10] describe an approach to generate and execute mutants using AspectJ, that they have included in a prototype tool. They get mutation operators for primitive and *String* data types by means of pointcut implementations. In these papers, the technique is not actually explained with detail enough to know how it works but, according to the authors, their mutation operators substitute the results returned by the methods in the class under test. For methods returning *int*, for example, they define six operators, all of them consisting in the substitution of the method body by a *return* statement that returns either: *-result, result+n, result-n, Integer.MIN_VALUE, Integer.MAX_VALUE, 0*. It looks like that all mutation operators consist in the whole substitution of typed-methods by single *return* statements. Clearly, mutants produced by these operators are almost useless, since they can be killed very easily: the coverage reached by killing all these mutants corresponds, more or less, to reaching all-methods coverage, which is a quite poor testing criterion. On the other side, the mutant executor has the choice of replacing methods executions with own code (i.e., that included in the advice) or with the original code, but no details are

provided about how this choice is effectively carried out. The authors show a mutant operator example with no statements related to this control. If the choice is made based on probability and this is relatively high, it is easy that a given test case execution reaches more than one mutant, so being more or less equivalent to a high-order mutant that, as it is well known, are much more easy to kill [11][12]. In fact, they compare their prototype with the Jester tool using some test suites of the *apache.jakarta.math* project: whilst with Jester 189 mutants remains alive, their prototype generate 1978 mutants (actually 1978 "mutated behaviors") and only 3 remain alive.

On the second line there are several works for the mutation testing of aspect-oriented programs, which contribute with identification of common faults in this programming paradigm [13] and with the definition and implementation of mutation operators for them [14][15]. The object of study of these works is however completely different than ours, since they focus on mutation testing of aspect-oriented programs, and ours is the use of aspects for performing mutation testing.

2.3 Integration testing with mutation

Integration testing tries to find faults when two or more software units, which have been previously tested at unit level, are put to work together.

Delamaro et al. [5] are likely the first in applying mutation to integration testing. Being *F* a unit that calls to another one *G,* and being G_I and G_O the corresponding tuples of values that *G* receives and returns, they found three possible types of errors depending on the values exchanged between *G* and *F*. The authors define two groups of mutation operators that act inside the code of the functions in charge of the interaction of units: 9 operators for the caller function (*F*) and 24 for the called (*G*). Most of them modify the values of the parameters passed and the variables shared between units. For example, they increment or decrement values, swap parameters of compatible types, etc. Besides the definition of specific operators for reproducing the three types of errors mentioned, an important difference with respect to traditional mutation is that the scope of application of these operators is limited to the areas of the program in charge of interactions.

Ghosh and Mathur [16] also propose mutation operators for integration testing, but now from a more "black-box point of view". These authors define five types of mutation operators for CORBA interfaces: *replace* the *in, out* and *inout* parameter modifiers with another in the list; *swap* parameters of compatible types; *twiddle* a parameter value *x* with *pred(x)* or *succ(x)*; *set* to a fixed value (a boundary value, for example); and *nullify* an object reference. To perform this kind of mutation testing, "the tester runs the test inputs on the client using the original interface and then on the client using the mutated interface". The goal is to obtain a response different in the mutated interface. The authors define three coverage criteria too: *MET* (coverage of all the methods in the interface), *EX* (raising all exceptions thrown) and *ME* (coverage of both *MET* and *EX).*

In the context of black box testing, Edwards [17] also lays elements between the client and the server, what he calls *BIT (Built-in-test) wrappers*, which offer the client

the same interface than the original component. The author suggests that BIT wrappers can "provide a number of self- checking and self-testing features that can then be used to encase the underlying component." Although he proposes other uses, BIT wrappers could also be used to mutate the interfaces, in the same line than Ghosh and Mathur do.

The approach of modifying the data sent to the component is also used by Offutt and Xu [18], but now in the context of web services, which offer their interface through WSDL document and play a similar role to a component. This technique is called *data perturbation* and is based on the substitution of the values exchanged in the SOAP messages.

2.4 System testing with mutation

The main concern of testing at system level is to find faults when the system (previously tested at integration level) is almost ready to be deployed at the production environment. Thus, this level requires different types of testing techniques, such as usability, performance, scalability, security, compatibility, etc.

In their complete revision of mutation literature, Jia and Harman [19] mention two lines of work in this context: mutation testing for security policies and for networks. With respect to the former, the works propose and/or apply operators that inject common flaws into different types of security policies: Martin and Xie [20], for example, modify policies described in XACML; Ghosh et al. [21] implement a set of "fault injection functions" (that play the role of mutation operators) to "simulate a variety of anomalous program behaviors": their fault injection engine forces overflows, is capable of corrupting booleans, characters, strings, integer and doubles, and also supports composition of these simple fault injection functions. Regarding the latter, the works focus on the mutation of protocols, often changing their formal specifications, or changing the environment in which the target application is executed (changing external libraries, network behavior, environment variables, contents of accessed files, and, in general, all the input channels of the application [22]).

3 Implementation of operators for system testing

The strategy behind system level mutation operators is to identify all system level interactions and induce changes that could lead to erroneous behavior. Mutation operators for this testing level were grouped into five categories [7]:

a) *Internal interaction* mutation operators affect connections between packages and methods, including the common getters and setters.
b) *User interaction* mutation operators target the user interface, testing how the user interacts with the software.
c) *Configuration interaction* mutation operators modify configuration settings of the software.
d) *Version interaction* mutation operators modify which versions of software components are being used.
e) *Environment interaction* mutation operators target connections between the software and its external environment.

When designing these mutation operators, our intent for completeness is to modify all interaction points. As we are dealing with different systems already implemented, some kind of "intrusion" mechanism in the code is needed to introduce the desired modifications in the required places. It is almost tautological to say that introducing faults in the system statements in charge of interacting with other systems requires the detection of the places where such interactions are made.

In general, all the statements that send or receive data to or from outside finally rely on library functions: for example, executing a SQL statement against a database server typically requires the use of a library provided by the database vendor; moreover, for the final sending of the statement to the server, the vendor library will rely in some kind of socket, probably developed by an additional developer. The use of aspects is an excellent way to alter the code of these external elements.

3.1 Aspect-oriented mutation

Considering the pointcut code of **Fig. 4**, two consecutive calls to the write method will save two incomplete byte arrays. Suppose (for clarity) that, besides byte arrays, we could pass strings as first parameter: the pair of calls to write included in the code of **Fig. 7** would be catched by the pointcut. The text saved in the disk will be *ello orld* instead of *Hello World*, since the first byte is removed in both calls.

Mutation operators should not introduce the alteration in the program behavior whenever the statement is executed: this is, if the pointcut is executed the two times that *write* is called, it will be quite easy to detect the change, what has no usefulness for the testing goal.

On the contrary, the execution engine should execute each test case so many times as calls are captured by the existing pointcuts: suppose there are 5 pointcuts that alter the behavior of *write(byte[], int, int)* and that test cases contain 15 calls to this method: then, the engine must carry out 15*5=75 executions of the test cases, each one affecting a different call. Going back to the example of the pointcut in **Fig. 4**, since the test case of **Fig. 7** calls *write* twice, the first execution of the test case would modify the first call to *write*; in a second execution, the change will be applied to the second call (**Fig. 5**).

Call	Original Code	Executed code
1	`f.write("Hello", 0, 5);` `f.write("World", 0, 5);`	**`f.write("ello", 0, 5);`** `f.write("World", 0, 5);`
2	`f.write("Hello", 0, 5);` `f.write("World", 0, 5);`	`f.write("Hello", 0, 5);` **`f.write("orld", 0, 5);`**

Fig. 5. Execution of test cases

Actually, these changes are not mutants in its traditional sense (the program code is not altered), but modifications in the regular program execution, what can be considered as a "controlled mutated behavior".

3.2 Aspect-oriented mutation process

The mutation process starts with the execution of a special set of aspects containing pointcuts in charge of counting the number of calls to the operations whose behavior requires to be modified. The names of the files implementing these aspects end with the *CountCalls* token. These pointcuts just compute this count and do not modify the regular execution of the program. For the code in **Fig. 7**, a *CountCalls* pointcut would save 2 for the *write* method. The code shown in **Fig. 6** counts the number of calls that a program makes to the *getInt(int):int* method of *java.sql.ResultSet*.

```
int around(java.sql.ResultSet caller, int col) :
  target(caller) && args(col) &&
  call(int java.sql.ResultSet.getInt(int)) {
    Store.updateNumberOfCalls(
      thisJoinPointStaticPart.getSignature().toString());
    return proceed(caller, col);
}
```

Fig. 6. Counting calls for the *ResultSet.getInt(int)* method

```
public void testWrite() throws Exception {
    FileOutputStream f=
          new FileOutputStream("A");
    f.write("Hello", 0, 5);
    f.write("World", 0, 5);
    f.close();
          ...
}
```

Fig. 7. A simple test case

Once the number of calls to each pointcut is known, the implementation goes over a different set of pointcuts that share the *CountCalls's* headers but that now, in fact, mutate the program behavior: if there are *n* calls to the captured method, the execution engine makes *n* executions of the test case. In each one, it modifies the behavior only in the call in turn.

Written as an algorithm, a possible pseudocode of the process appears in **Fig. 8**. Note that some additional loops have been added in order to provide a comprehensive description of the process.

3.3 Applications of AOM

During system testing, AOM can be used to assess the quality of the test suite, to find faults in third-party software (i.e., external libraries or systems) or to find faults in the system we have built (when it receives unexpected responses from external systems). Whilst the first application of AOM is obvious, maybe the second and the third are not. For the second, suppose an external system which will be accessed from another one we have developed. To check its suitability to be integrated with ours, its behavior may be modified with AOM in order to test, for example, its robustness when it receives modified data. Supposing it passes the tests, we will be able to check how our system behaves when the responses received from the server are not the expected ones.

Thus, the general goal of AOM is finding faults during system testing by means of a good test suite, whose quality is firstly measured by finding the artificial faults inserted by the aspect advices. The idea is the same of the generic mutation process described in [2]:

1) Build a mutation-adequate test suite *T* that finds all the artificial faults.
2) Execute *T* against the original system: if no faults are found, then is highly probable that it has no faults.

```
testSuite = set of test cases
counters = set of apect files ending with “CountCalls”
For each counterAspect in counters do
    Add counterAspect to the execution path
    For each counterPointCut in counterAspect
        numberOfCallsToPC = 0
        For each tc in testSuite
            Execute tc
            numberOfCallsToPC += calls made by tc
        end
        save numberOfCallsToPC in pcName.txt
    end
    Remove counterAspect from the execution path
end

mutants = set of aspect files not ending with “CountCalls”
For each mutantAspect in mutants
    Add mutantAspect to the execution path
    For each mutantPointCut in mutantAspect
        n = read numberOfCallsToPC from pcName.txt
        For i=1 to n
            For each tc in testSuite
                   Execute tc, applying mutantPointCut to the i-th call
            end
        end
    end
    Remove mutantAspect from the execution path
End
```

Fig. 8. Pseudocode of the AOM process

3.4 Implementation of a tool for AOM

Fig. 9 shows the configurations parameters of the tool. It requires to know the paths for: aspect implementations, boot, class path, junit.jar file (the tool supports both *JUnit* test cases as classes with a *main* method), *AspectJ library* and *AspectJ weaver* (for the dynamic linking).

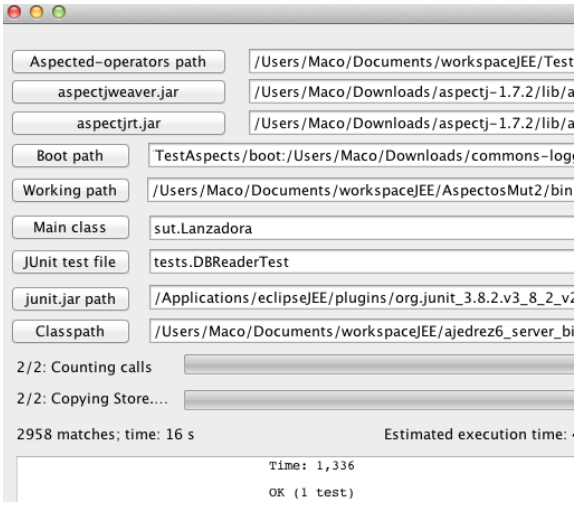


Fig. 9. An excerpt of the tool's main screen

4 Testing the *gmail* SMTP server

gmail is probably the most popular email system, with millions the users along the world. Its SMTP server can be accessed from a Java client using some of the classes in the *javax.mail* package. One example of this (taken from [1]) appears in **Fig. 10**.

```
private String SMTP_HOST_NAME = "smtp.gmail.com";
...
private String emailMsgTxt = "Test Message Contents";
private String emailSubjectTxt = "A test from gmail 2";
private String emailFromAddress = "macario.polo@gmail.com";
private String[] sendTo = { "macario.polo@gmail.com" };
...
public void sendSSLMessage(String recipients[],
  String subject, String message, String from)
      throws MessagingException {
   boolean debug = true;

   Properties props = new Properties();
   props.put("mail.smtp.host", SMTP_HOST_NAME);
   ...
   Session session = Session.getDefaultInstance(props,
           new javax.mail.Authenticator() { ... });
   session.setDebug(debug);
   Message msg = new MimeMessage(session);
    ...
   msg.setRecipients(Message.RecipientType.TO, addressTo);
   msg.setSubject(subject);
   msg.setContent(message, "text/plain");
   Transport.send(msg);
  }
```

Fig. 10. Code to send emails with *gmail* (taken from [1])

[1] http://stackoverflow.com/questions/9926125/smtp-client-java-program

With aspects, mutations can be inserted in any of the operations or classes directly imported by the shown code. However, in some moment all these classes end relying in some kind of input or output stream class and, thus, mutation could be inserted in these classes. **Fig. 11** shows the call tree obtained after profiling with the TPTP profiler the *sendSSLMessage* method in **Fig. 10**: the final call to *Transport.send(msg)* consists in a call to *sendMessage* (note the highlighted row in **Fig. 11**) that, in turn, in some moment calls to methods of *java.io.OutputStream*. The use of a profiler is a good method to know what methods should be mutated.

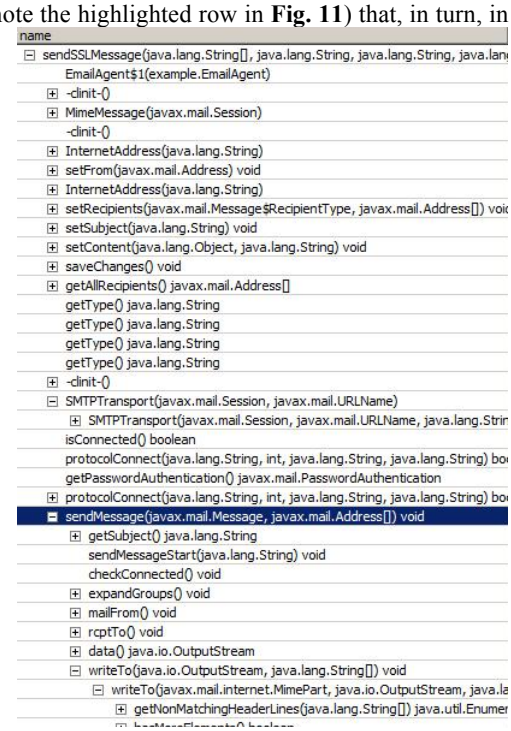


Fig. 11. Call tree obtained with TPTP profiler

Step 1. Counting calls

Once the mutant operators (implemented as pointcut aspects) are written, the first step of the AOM process (**Fig. 8**) is to count how many times each pointcut is called: this is, how many times the interesting operations are catched by the test cases. In our case, we have implemented pointcuts to catch the operations shown in **Table 1**: all of them belong to commonly used libraries that access external systems. When the code in **Fig. 10** is launched, the pointcuts catch the calls shown in **Table 2**.

Implemented operators

Depending on the purpose of the operation, they are mutated in different ways:

- Operations that write or send a byte array or a string (*write, print, println): Deletion* (does not send the message), *Duplicate* (sends twice the message), *Inverter* (reverses the byte array or string), *RemoveFirstByte* and *RemoveLastByte*.
- Operations that take an object as parameter (*writeObject*): nullify the object.
- Operations that read or receive a message (*read, readLine): Deletion* (does not execute the operation, so the message is not receive), *Duplicate* (receives twice the message), *Inverter* (reverses the byte array or string), *RemoveFirstByte* and *RemoveLastByte*.
- Operations returning a boolean (those in *java.sql.Connection* and in *java.sql.ResultSet.next*): return the negation of the true result.
- Void operations (*flush, close):* the operation is not executed.

Table 1. Some of the mutable operations

	Class	Mutated operations
java.io	BufferedOutputStream	write(byte[], int, int); flush(); close()
	FileOutputStream	write(byte[], int, int); write(byte[])
	OutputStream	
	PrintStream	print(String); println(String)
	ObjectOutputStream	write(byte[], int, int); write(byte[]); writeObject(Object)
	BufferedReader	String readLine(); int read()
java.sql	Connection	getCacheResultSetMetadata(); getUseFastIntParsing() supportsQuotedIdentifiers(); useMaxRows()
	PreparedStatement	setInt(int, int); setString(int, String)
	ResultSet	getInt(int); getString(int); next(); reallyResult()

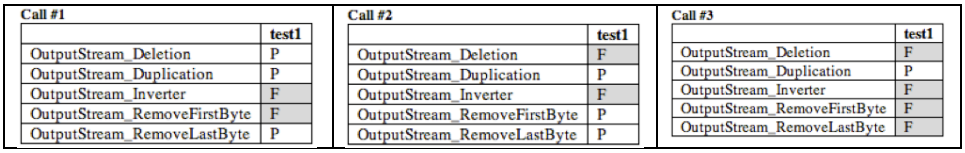
Table 2. Number of times that the interesting operations are catched

Class	Operations	Times used
java.io.OutputStream	write(byte[])	34
	write(byte[], int, int)	22
java.io.BufferedReader	readLine() : String	7
	Total:	63

Step 2: execution

According to the AOM pseudocode, the execution engine will launch the code of **Fig. 10** so many times as calls there are to the catched operations: 63 in this case (**Table 2**). It is expected that, for example, that application of *RemoveFirstByte* will cause an exception in the server when it is applied to the user name or password, but that it works when it affects the email subject or body.

The final result of the execution is a set of killing matrixes, one for each operation call. **Fig.** 12 shows the results of applying the five operators defined for the *write(byte[])* method of *OutputStream*, in the three first calls done by the test case. For example, for *Deletion* the test case returns a pass verdict (*P*) in the first call and a fail verdict *(F)* in the others, *Duplication* always passes and *RemoveLastByte* only fails in the third one.



Call #1

	test1
OutputStream_Deletion	P
OutputStream_Duplication	P
OutputStream_Inverter	F
OutputStream_RemoveFirstByte	F
OutputStream_RemoveLastByte	P

Call #2

	test1
OutputStream_Deletion	F
OutputStream_Duplication	P
OutputStream_Inverter	F
OutputStream_RemoveFirstByte	P
OutputStream_RemoveLastByte	P

Call #3

	test1
OutputStream_Deletion	F
OutputStream_Duplication	P
OutputStream_Inverter	F
OutputStream_RemoveFirstByte	F
OutputStream_RemoveLastByte	F

Fig. 12. Three killing matrixes

To help in the analysis of the results, the advices may be programmed to leave a trace of the actual and the modified data: each row in **Table 3** shows, for the *write(byte[])* operation, the original byte array and the results of applying the *RemoveLastByte* (the non shown rows consist of non visible characters, such as \r or \n).

The first row, for example, consists of the *EHLO* command (*Extended HELLO)*, whose "argument field contains the fully-qualified domain name of the SMTP client if one is available. In situations in which the SMTP client system does not have a

meaningful domain name (e.g., when its address is dynamically allocated and no reverse mapping record is available), the client SHOULD send an address literal" [23]: since it does not matter the value sent, the SMTP server accepts the connection and the test cases passes; in the second row (corresponding to the 3rd call), however, some different situation is detected by the test case that leads to a fail verdict: actually, as seen in **Fig.** 12, the message is not sent with any modification of the "AUTH LOGIN" command.

Table 3. Some results for *RemoveLastByte*

void java.io.OutputStream.write(byte[] a)		
Call	**Values of *a* in the original (left) and with *RemoveLastByte* (right)**	
1	EHLO 192.168.0.2	EHLO 192.168.0.
3	AUTH LOGIN	AUTH LOGI
5	bWFjYXJpby5wb2xvQGdtYWlsLmNvbQ==	bWFjYXJpby5wb2xvQGdtYWlsLmNvbQ=
9	MAIL FROM:<macario.polo@gmail.com>	MAIL FROM:<macario.polo@gmail.com
13	DATA	DAT
15	From: macario.polo@gmail.com	From: macario.polo@gmail.co
33	QUIT	QUI

Step 3. Interpretation of the results

When one is testing an external system, such as the *gmail* SMTP server, the tester should expect that most of test cases fail: pass verdicts should suggest that the server is accepting incorrect commands, although it is possible that some of them, even though they are mutated, fit the SMTP official specification and should pass. **Fig. 13** shows some of the emails received in the account: they correspond to executions that have not shown a fail verdict: some of them have the complete subject ("A test from gmail 2"), others have joined the subject to some header text, etc.

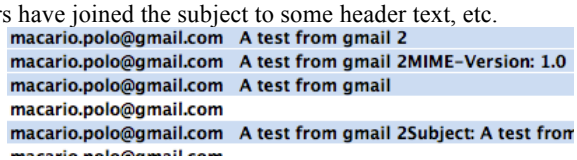


Fig. 13. Some of the "mutated" emails received

5 Conclusions and future work

Aspect oriented programming is a powerful tool to inspect or modify the behavior of programs with no doing direct modifications in their source code. Currently, there are aspect oriented frameworks from many of the most popular programming languages, such as ASP, C++, Java, Perl, PHP or Ruby.

Besides this ability, our state of the art revision evidences that AOP has never been (seriously) used for mutation testing. Thus, to our best knowledge, this is the first work analyzing the possibilities of applying AOP for mutation testing. The paper has discussed some implementation issues and has described the first version of an Aspect Oriented Mutation process that, like any other first research proposal, requires further validation and maturation. The paper has shown how AOM can be applied to the test-

ing of third party software, such as COTS (Commercial-Off-The-Self) components, allowing a much more in depth testing than previous techniques.

6 References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[2] M. Polo and P. Reales, "Mutation Testing Cost Reduction Techniques: A Survey," *IEEE Softw.*, vol. 27, no. 3, pp. 80–86, May 2010.

[3] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing," *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 685–718, Jun. 1991.

[4] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[5] M. E. Delamaro, J. C. Maidonado, and A. P. Mathur, "Interface Mutation: an approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[6] P. Reales and M. Polo, "Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases," 2012, pp. 646–649.

[7] P. Reales, M. Polo, and J. Offutt, "Mutation at the multi-class and system levels," *Science of Computer Programming*.

[8] R. Laddad, *AspectJ in action enterprise AOP with Spring applications*. Greenwich, Conn.: Manning, 2010.

[9] B. Bogacki and B. Walter, "Evaluation of Test Code Quality with Aspect-Oriented Mutations," in *Extreme Programming and Agile Processes in Software Engineering*, P. Abrahamsson, M. Marchesi, and G. Succi, Eds. Springer Berlin Heidelberg, 2006, pp. 202–204.

[10] B. Bogacki and B. Walter, "Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing," in *Software Engineering Techniques: Design for Quality*, K. Sacha, Ed. Springer US, 2007, pp. 273–282.

[11] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Softw. Test. Verif. Reliab.*, vol. 19, no. 2, pp. 111–131, Jun. 2009.

[12] P. Reales, M. Polo, and J. L. Fernandez, "Validating 2nd-Order Mutation at System Level," *IEEE Transactions on Software Engineering*, pp. 1–1, 2012.

[13] J. S. Bsekken and R. T. Alexander, "A Candidate Fault Model for AspectJ Pointcuts," in *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06*, 2006, pp. 169–178.

[14] P. Anbalagan and T. Xie, "Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs," in *Proceedings of the Second Workshop on Mutation Analysis*, Washington, DC, USA, 2006, p. 3–.

[15] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation Testing for Aspect-Oriented Programs," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, Washington, DC, USA, 2008, pp. 52–61.

[16] S. Ghosh and A. P. Mathur, "Interface mutation," *Software Testing, Verification and Reliability*, vol. 11, no. 4, pp. 227–247, 2001.

[17] S. H. Edwards, "A Framework for Practical, Automated Black-Box Testing of Component-Based Software," *SOFTWARE TESTING, VERIFICATION AND RELIABILITY*, vol. 11, p. 2001, 2001.

[18] J. Offutt and W. Xu, "Generating test cases for web services using data perturbation," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–10, Sep. 2004.

[19] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[20] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proceedings of the 16th international conference on World Wide Web*, New York, NY, USA, 2007, pp. 667–676.

[21] A. K. Ghosh, T. O'Connor, and G. McGraw, *An Automated Approach for Identifying Potential Vulnerabilities in Software*. .

[22] W. Du and A. P. Mathur, "Testing for software vulnerability using environment perturbation," in *Proceedings International Conference on Dependable Systems and Networks, 2000. DSN 2000*, 2000, pp. 603–612.

[23] Internet Engineering Task Force, "Simple Mail Transfer Protocol," *http://www.ietf.org/rfc/rfc2821.txt*. .