UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Automatic generation of test cases to improve code coverage in Java

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Carlos Santiago Galindo Jiménez

Tutor: Thomas Gross
Jose Miguel Benedi

Course 2017-2018

Contents

CHAPTER 1

Introduction

Testing is an essential part of writing software. It is used to verify the integrity of the program, detect bugs and other errors. Writing good test suites will therefore increase the general quality of any software project.

A common measure employed when writing software and tests for such software is code coverage, or the amount of lines, functions, branches, etc. of a program that are executed for a given test suite. This improves the test suite generation and expansion, specially as the program is updated and grows.

1.1 Motivation

While teaching programming subjects, it is common to test projects written by students with test suites that cover most legal/illegal cases. Such task can range in difficulty, depending on the scope of the project and its input complexity. Ideally, all paths of a properly programmed solution will be executed by such test suite, but obviously the submitted solutions may have dead paths or simply paths that no normal (legal or illegal) input will trigger.

These paths are normally the result of multiple code revisions, or bad coding practices by the students. The program could function properly for most inputs, though, so an advance coverage tool is needed to help the evaluator generate further tests that cover those missing paths.

Developers would also benefit from an automated or simplified way to generate and improve tests, specially with increasingly complex software solutions for our everyday problems.

1.2 Objectives

- Obtain coverage information for a Java program, optionally only for a specific class, method or even line in the source code.

- Produce a history overview of the evaluation of a conditional jump for a given input.

- Develop a suite of programs to demonstrate the tool.

The tool will be developed in Java and will deal with the compiled classes with debug information, as well as the original source files, to further help the evaluator by providing relevant snippets of code.

The target language will be Java, although due to employing the Java byte-code contained in class files, it could easily made compatible with other JVM-based languages, such as Scala or Kotlin, as well as other languages that do not run primarily in a JVM, but can be compiled to byte-code.

CHAPTER 2
Solution design

A tool that generates new test cases will be either severely limited in scope and restrictive of the language features it supports or it will unavoidably need to “understand” what happens inside each method call (not only user-written methods but also external libraries and internal classes and methods from Java). Therefore, the process of designing a solution starts by obtaining the relevant information and designing systems that can, if not understand the information, present it to the developer in order to ease the process of understanding which variable or variables must change in order to execute previously not run sections of code.

Once the basic information is obtained, the analysis performed on it can be simple, such as just displaying that information, or more complex, showing the developer the exact modification that must be made in the test suite to increase coverage.

2.1 Technologies employed

As detailed previously, the tool has been developed in Java, targeting programs written in Java, but being adaptable to any language that can be run in a Java Virtual Machine (JVM). Some helper libraries have been employed to ease the general development; such as `commons-cli` from Apache Commons[1] for parsing command-line options easily.

2.1.1. ASM: byte-code engineering

Most of the work is done using the APIs exposed by the ASM[2] library, which exposes the byte-code of a class. It provides two different APIs:

- Core API: a event-based API, by which a chain of `ClassVisitor` (and `MethodVisitor`) objects can be chained, reading and optionally modifying the class. All chains begin with a `ClassReader`, that reads each element of the class, triggering the appropriate events to the chain. Chains can optionally end in a `ClassWriter`, that receives the events called by the last visitor instance in the chain and converts them into a byte array, allowing the programmer to either load the class or save it to disk for later usage.
- Tree API: an object based API, which represents each instruction, label, method and class as an object, representing classes as a tree of nodes. This allows for modifications or analyses that are not possible in a event-based environment, as in that case,

[1] https://commons.apache.org/proper/commons-cli/

[2] https://asm.ow2.io

the whole method/class is only complete when its last event is received. Additionally, as part of this API, there exists a framework for generating control flow graphs from the byte-code instruction list, which can then be used for static analysis of the code.

Both APIs are used in the software project, with the Core API being predominant in it, and the Tree API is only used where the Core API is lacking, such as the generation of control flow graphs (CFG). A class or method can be obtained in the tree API from a series of events in the core API, allowing the connection between both. Performance-wise, the tree API is 20% slower than the event based API, so it is used as sparsely as possible, even though the design is not specifically focused on performance.

ASM also includes multiple tools to generate instructions more easily, such as:

AdviceAdapter A method visitor that includes instructions to manage local variables without having to analyze which are used where, and eases instruction generation by providing distinct methods for each type of instruction (push constant, call method, etc.).

CheckClassAdapter A class visitor that checks for errors such as mismatching stack and the validity of types, among others.

TraceClassVisitor A class visitor that outputs the class structure in readable format, and can be used to visualize changes or even just convert byte-code opcodes (operation codes) to readable text, e.g. 0x9a is printed as `IFNULL`.

ASMifier Utility that can show the programmer the necessary Java instructions to generate a snippet of Java code using the ASM library, easing the transition into byte-code for programmers that only know Java.

Note that the class visitors described have a method version, that is both used internally and accessible to the programmer.

2.1.2. Java Code Coverage Library (JaCoCo)

As a helper tool to determine the coverage easily and avoid re-implementing a probe system, JaCoCo[3] has been used. It runs a program and collects information on multiple levels: byte-code instruction, branch, line, method and class coverage. It also displays method complexity (a measure of the complexity of the control flow graph associated with a method).

On top of that, JaCoCo can accumulate results from running the same program multiple times (with different inputs). It also provides multiple options to gather execution data, and we opted for the safest one regarding the possible side-effects, albeit the slowest performance-wise: using the JaCoCo Java agent to run the code in a different JVM, which saves the result to a file, and then reading the file containing the coverage report.

JaCoCo helps generate an initial coverage report before any instrumentation begins, which allows the program to optionally instrument and analyze only the first branch with partial coverage (meaning that it has been executed but only once or always with the same result). It is only employed when the user wants to find information regarding only one branch without specifying the location, and it plays a supporting role in the general structure of the program.

[3] https://www.jacoco.org/jacoco/

2.1.3. Java backwards-compatibility

At first, there wasn't any obvious need to develop the program with backwards compatibility, because Java is in general forwards-compatible source wise. The most common reason to target previous versions of Java is to run the software in devices that cannot obtain a newer version of Java for any reason. ASM, which is the only truly indispensable library for this project has backwards-compatibility all the way back to Java 5 (and even previous versions for parts of the library), so even though there is no compatibility, the port to previous versions should be a simple change.

Regarding Java Development Kit (JDK) and Java Runtime Environment (JRE) compatibility, the design tries to be the most generic and compatible it can be, but it makes some assumptions, for example that `javac` does not perform complex analyses and simplifies loops or branches in the program. Small optimizations are expected; such as computing arithmetic operations where all operands are numbers or constant propagation.
Tests to the current design have been performed exclusively using the OpenJDK version 1.8.0_181-b13 (64 bit); which is not the best practice: the Java SE and EE environments should have also been tested.

2.2 Detailed design

In its current form, the resulting software consists of two distinct jar files: the profiler, containing most of the logic and classes of the program, to model program execution and obtain static information from the class files and a Java agent, that can be attached to a JVM in order to obtain the execution data (which contains, among others, a method call stack trace, and a log of variable assignments and conditions with the respective runtime values).
The process performed in the profiler jar can be described as:

1. Initialization
   (a) Process command-line arguments.
   (b) Obtain list of files containing the inputs for each test.
   (c) If necessary, run each test with JaCoCo attached to obtain the first location that is partially covered.
2. Run tests with agent (section 2.2.2)
   (a) For each input, run the program to be tested with the agent attached.
   (b) Read the logs generated by each execution and convert them to objects.
3. Process results
   (a) Generate a control flow graph for each method and obtain the chain of instructions that lead to the values consumed by the condition in a branch. Section 2.2.3.
   (b) Read the tested program's classes to determine variable names, line numbers and other debugging information.
   (c) Obtain the last value logged for relevant local values and arguments.
4. Display results

2.2.1. Byte-code format and execution model

Byte-code, as previously mentioned, is the intermediate representation into which multiple languages are compiled, and it is executed by a JVM. Its instruction format and execution model is dictated by Oracle's specification of the Java Virtual Machine[4]. A complete understanding of Java's byte-code is not necessary to use it, but grasping its execution model is key to understanding how Java is compiled and how byte-code works.

Each byte-code designates an operation to the stack, consuming a number of values and pushing a result if available. Other instructions act on local variables or perform unconditional jumps, therefore not altering the stack. Finally, there are operations whose only function is to alter the state of the stack, such as pushing constant values, popping or duplicating the top of the stack, and swapping the two top values.

User-defined variables are placed in a memory section specifically for local variables and method arguments (including the reference to the method receiver[5] for static methods). Values can be copied from this memory segment to the stack and vice versa. The execution of a method will begin with an empty stack and with its arguments placed as the first local variables. Local variables are indexed and treated very similarly as registers: a local variable will be located at a specific location (or index) in the locals segment while it is being used, and no two variables will share the same index if they are both used in a matching section of code.

With the stack and local variable on hand, each byte-code performs either a stack/local editing task or a Java action (field access/assignment, method call, arithmetic, logic, comparisons, etc.). Lastly, if compiled with debug information and not intentionally obfuscated, line number, local variable names and source file path are provided in the byte-code. This is indispensable for our tool to work and provide meaningful feedback to the user, who is viewing the source, not the class files. The analysis tools still work, but the display of results may not be as useful.

2.2.2. Byte-code instrumentation

The modifications inserted into the original code are short and simple. In general, they call a static method to log the runtime value, adding other arguments to identify the location in the code and the current method call stack. Modifications that log a value follow this general structure:

1. Duplicate the value to be logged.
2. If the value is a primitive, call a the corresponding static method to convert it to an object, generally of the form `Type valueOf(primitiveType)` from the class `Type`.
3. Push to the stack any necessary identifiers to help match the logged value with an assignment, comparison, conditional jump, etc.
4. Call a static method from the `Logger` class, which prints an easily parsed message. This consumes all the values generated by the instrumentation code.

In order to position each logged action chronologically in its specific method execution, a stack trace of the different methods called is also logged. The structure is much simpler, as it just consists of pushing the qualified class name and method signature to the stack and calling the `enterMethod` or `exitMethod`, respectively. When a method is exited, the stack is checked for consistency. All methods are wrapped in a `try-finally` structure to avoid a mismatch of the stack due to an uncaught exception.

[4] https://docs.oracle.com/javase/specs/index.html
[5] The object that the method has been called upon, referenced in Java with the keyword `this`.

2.2.3. Control flow graph dependency analysis

The stack-based execution of byte-code instructions can be modeled as a directed graph $G = (V, E)$. In a method with arguments[6] numbered 1 through n and m instructions, the set of nodes is defined as $V = \{v_i \mid \forall i \in [-n, m)\}$ and the set of edges as $E = \{v_i v_j \mid v_i, v_j \in V \wedge i \neq j \wedge j \in [0, m)\}$.

A node (v_i), if $i \geq 0$ represents the result of the ith instruction of the program; otherwise it represents the ith argument of the method. The result may be `null` if the instruction has no result pushed to the stack, therefore being a sink node. An edge ($v_i v_j$) implies that v_i is one of the values consumed by instruction j, which results in the value v_j. v_i can either be an argument or the result of another instruction. Instructions that just modify the stack can be source nodes (push a constant, new object), sink nodes (pop a value), modify the order in which the nodes are placed in the stack (swap) or copy the same node so that it can be used as argument for multiple instructions (duplicate). These last two have no practival effect on the graph, as they just affect the position and amount in the stack. Jump instructions will be ignored for now.

A simple static analysis is perfectly capable of generating such graph without much effort for individual expressions or assignments. In turn, the graph can be used to identify the source from which any value present in the stack has originated, and the instruction(s) in which it will be consumed. Identifying the sources in a comparison expression for a conditional branch is now trivial, but this is not enough, as the values could be traced further to the beginning of the method. The greatest limitation is the exclusion of jumps (conditional and unconditional): the intra-statement analysis will only be valid when jumps modify the graph accordingly.

The most obvious solution is a control-flow graph (CFG): a directed graph which describes the different paths that a program execution may take. Typically, each node of the graph represents what is called a "basic block": a sequence of instructions that are always executed uninterrupted; instructions can only jump to the beginning of a block and from the end of a block. The only way for the control-flow to exit a basic block through an instruction other than the last one would be an error that terminates the program. In the tool's implementation, we are going to represent each instruction as a vertex in the CFG.

The ASM library includes, as part of its tree API, a series of classes that can be extended to generate CFGs from byte-code instructions. This CFG represents each instruction connected to others given the program's control flow, with a "frame" associated to each instruction, representing the state of the local variables and stack in that precise moment. Specifically, the state of the program *before* each instruction is described in a Frame, which contains a Value object for every local variable and stack value in use. An Interpreter simulates the effects of each possible byte-code instruction in both the locals and the stack of its corresponding Frame. It receives the instruction and the appropriate amount of Value objects (arguments for the instruction to consume), and optionally generates a resulting Value to be pushed to the stack. Finally, the Analyzer makes use of the Interpreter to populate the Value objects for every instruction's Frame object. It simulates internally stack reordering or duplication instructions, as well as local variable access and storage, but most importantly it performs a forward-flowing data analysis, which allows for multiple checks, for example regarding type consistency or, in our case, value relationships.

[6] Note that when the text makes a reference to arguments, the receiver (see footnote 5) is included by default.

All of these classes have been extended to accomodate both a CFG at the instruction/Frame level and the previously described graph of value generation and consumption at the Value level:

Frame → **Node** Links to the predecessor and successor instructions, representing the edges between the different Nodes/instructions in the CFG.

Value → **StackValue** Links to the Value's "parents", arguments to generate this value, and "children", values that are generated using this one[7]. For source nodes, it indicates the kind of source of the value: method argument, hard-coded constant, new object, static field or method call without arguments (static to avoid having a receiver argument). It also contains utilities to locate sources that have influenced a value, as well as the nodes that act as sinks, consuming such value without providing a result. It is also able to represent the graph going back to the source nodes as a chain of instructions with its arguments.

Interpreter → **StackInterpreter** As mentioned, contains all the logic that consumes and generates StackValues, linking them in the process, according to the arity of each instruction. It also manages merging pairs of Nodes containing two different but mergeable states for the stack and locals, each representing the last state in two paths that converge, e.g. after an `if-else` block. The merge is performed by creating a new Node with new StackValues for each local or stack value, whose parents are the StackValues in the same position. This represents an alternative: either the value comes from one path OR the other. The opposite mechanism (one value that can be used either by one path or another) is implemented in the Analyzer.

Analyzer This last one has been extended as an anonymous class. It specifies the initial Frame/Node for a method (an empty stack and all the arguments as the first locals) and the case where a Node has multiple "children" or "successors", e.g. after a conditional jump.

With all these classes in place, we can create a more complete picture of the structure of the method and the movement of values between stack and locals. Using the complete CFG, we are able to obtain the origin of any stack value or local variable, tracing it back to constants and other source nodes, or even to the start of the method. This allows the tool to display basic information to the developer: which variables influence and therefore modify the result of a conditional jump's comparison, which would increase coverage.

An important remark when modifying or extending the current agent is that the modifications can't alter the control flow, or add values to the execution, but only log values and probe locations, such as method entrance and exits.

2.2.4. Design limitations

In the process of designing the software, some limitations have been encountered and worked around, and some have been added as limitations, due to the high cost or impossibility of an alternative solution. These are the limitations established on Java programs that can be used with our tool and the reason for their exclusion:

Lambdas Lambdas are anonymous functions introduced in Java 7 and are used to simplify the syntax of a single-method anonymous class. Sadly, they are all placed

[7]Values can only be used once and are popped from the stack, but the usage of duplication instructions and storage in local variables can bypass this limitation

inside the `java` package, making it difficult to differentiate them from native Java classes. Furthermore, some libraries used, and Java itself have lambdas that would not be convenient (or sometimes possible) to instrument. Given that lambda functions tend to be short or reference another method entirely, a decision was made to ignore lambdas.

Parallelism Given the cost of taking into account the different facets and options available for parallelism, and the lack of an explicit motivation to implement it has resulted in it being declared a limitation. No check is performed to guard the system about it, but some analyses may return erroneous results.

2.2.5. Security concerns

From a security standpoint, the program to be instrumented is treated in "good faith", meaning that no malicious behavior is expected from it. Regardless, there are always some security concerns and errors that arise from the execution of arbitrary code. To avoid any problems or side-effects when running multiple tests on the same classes, all executions must be completely separated from each other, and the safest way is to run each test in a different JVM. This introduces some additional overhead and maybe unnecessary resource consumption, but it avoids most of the interference and side-effects in static fields between executions that could lead to unforeseen bugs.

The main concern would be the possibility that, for some of the test cases, the program to be tested does not terminate properly, either due to an infinite loop, an uncaught exception or the JVM being killed by the system.

In the first case, the solution is obvious: establish a reasonable time limit (customizable by the user) that would allow the execution to complete normally in the worst case scenario. If it takes longer, the JVM process is terminated and the available results collected. This solution makes handling JVM shutdowns even more relevant. An alternative solution would be to implement communications with the agent attached to the running program in order to signal a timeout, or that the time requirements be handled by the agent itself.

The second case requires that the design either catch the exception by modifying the header of the main method to stop the exception or by printing the information obtain from the execution and ensure that the printer's buffer is flushed before the JVM terminates (either by normal means, an exception or the described timeout). Modifying the main method comes with its set of problems, as it also involves discovering which main method is the true one if more than one matches the requirements to be a main method. On the other hand, the JVM provides a shutdown hook, an option to execute a thread when the JVM is shutting down due to user error, system stop/kill signal, etc. It can be used to flush the remaining information to be printed and close the output stream properly. This option is also more powerful than simply catching every exception in the main method and generally safer, so this is the one we chose for the project. This solution also covers the third case (JVM shutdown by the system or another process).

2.3 Further improvements

Due to the nature of the problem to be solved and the limitations of a Bachelor thesis, there are multiple aspects that are underdeveloped and could be extended to make the tool simpler to use and more powerful.

2.3.1. User-friendly results

Part of the information provided to the user is a chain of operations from arguments, constants and other sources of information to show how each source is combined in order to generate the value that determines the conditional jump. This is done by showing the byte-code instructions' mnemonic with its arguments formatted as "instruction(arg1, arg2, ...)". This puts a burden on the user of the tool to interpret the different byte-code mnemonics. The user may not be familiar enough with byte-code in order to correctly interpret this information. Implementing a printer that converts trees of operations to the equivalent Java statement would be difficult, so the default printout from the ASM library has been used (using `TraceMethodAdapter`).

IDE integration

Expanding on the interaction between this tool and the developer, the output format could be improved into a set of HTML pages, as other coverage tools do. The ideal interface both for executing and presenting the results would be IDE integration, as the programmer can make small changes to their software and test suite on the fly, and re-running the tool as often as necessary. This also allows displaying the results in the context of the source code, which can be navigated normally, but now has extra information attached, such as the dependencies of a condition or the values taken by the various variables in the different assignments or definitions.

2.3.2. Improved branch and loop recognition

In its current state, the lookup of branches and loops is performed by detecting conditional jump instructions, specifically those of the form `IFxx`, `IFNULL` and `IFNONNULL` for comparing integers and objects against 0 or `null` (respectively), `IF_ICMPxx` and `IF_ACMPxx`, which compare pairs of integers or object references. This covers most of the conditional jumps in Java, with the notable exception of 'switch' blocks, but those are required to have constants as the target of the comparison, so there is no complex network of dependencies, and our tool would just point out the obvious.

A small problem arises when some instructions that are apparently not branches appear as such. For example, OpenJDK implements object comparisons with a conditional jump; the comparison `object == null` becomes:

```
aload_0           // Load object
ifnonnull label1
iconst_1          // Write 1 (true)
goto label2
label1:
iconst_0          // Write 0 (false)
label2:
```

The same occurs with comparisons against a second object, but using the `IF_ACMPEQ` or `IF_ACMPNE` for == and != respectively. This generates false positives when locating, instrumenting and displaying branches. The problem is not too great, as it can be useful to cover both options (comparison results in `true` or `false`) when creating tests, but if they are common it can result in too much information about non-branches or loops.

Solving this requires studying not only the JVM specification but each implementation of `javac`, such that structures which make use of conditional jump byte-code instructions can be identified and blacklisted or at least differentiated from the user-written branches and loops.

Alternatively, these patterns can be considered as partly covered instructions (if only one branch is taking in the whole execution), as not all of the byte-code instructions that have been compiled from the original Java expression have been executed. This may seem odd to the programmer, but it can make a test suite generated on the basis of full coverage more complete, by providing instances in which these comparisons result in `true` and `false`. Furthermore, coverage tools detail coverage down to the class, method, source line, branch and byte-code instruction level, so the source line (which is the atomic unit as far as a non-expert developer is concerned) would obtain partial coverage; e.g. the JaCoCo library behaves in this way.

2.3.3. Extended analysis of chains

As mentioned in section 2.3.1, the origin of the condition for a branch is shown as a chain of operations performed to the original arguments or the method, constants or objects created. Some operations with "complex" meaning, such as method calls are displayed as another transformation, with its arguments. This means that, because the actions inside other methods are not followed, new dependencies or lack thereof are not obtained. It would be entirely possible for the arguments of a method to not alter the object returned by it, but affect the course of the program in other ways: debug flags, strings for printouts, etc.

A difficult but useful improvement would be in-lining methods to improve the chain's accuracy when reporting dependencies. Short methods are frequently in-lined by the JVM's compiler, but short methods only go so far. The end-goal of this improvement would be to create a control-flow graph that shows the complete picture: all methods in-lined into one giant method. In this new "mono-method" program, the dependency graph can determine with perfect accuracy the values that affect an instruction.

Another step further would include partial dependencies, such as the array partially depending on its size, or arithmetic and boolean operations being simplified to the maximum, in order to display the true effect of each edge in the dependency graph.

2.3.4. Performance improvements

Performance has not been a core objective to strive towards while developing this tool. It tends to err on the side of security rather than performance. However, it doesn't mean that there are no possible simple (and complex) changes to improve it:

- Split the classes into more jar files, in order to load the minimum amount of classes as dependency of the agent, and thus slightly reduce the computational requirement of creating and destroying JVMs for individual tests.
- Parallelization of analysis and tests: the methods are currently analyzed after all the tests have finished, but both could be parallel. The execution of the tests themselves could also be parallelized, but it may not be worth it if the program to be tested is IO-bound.
- Instrumentation could be cached into the disk and not be performed multiple times. It could also be cached between runs, if the classes to be tested haven't changed but the test suite has.
- Lookups of various debug markers and information could be sped-up by caching, at the expense of RAM usage.

CHAPTER 3

Conclusions

The basis of this BSc. Thesis has been code analysis, both static and dynamic. The initial goal of the project was to achieve automatic generation of test cases, by observing how the different inputs of a program modify each element of its execution and therefore affect which branches are executed and the arguments of method calls. As mentioned previously, this tool has erred in the side of a wider compatibility with more expressions, with the penalty of not achieving complete automation in test generation. It provides a tool for the programmer to focus the test-creating effort and shorten the amount of time needed to develop a test suite with complete code coverage.

An alternative version of this tool on the other end of the spectrum would have been an automatic one, which generates test cases for any program written in a limited subset of Java, such as one with only primitive types. In the end, such version would only be useful for the simplest programs, while our tool provides some help and guidance for any kind of program (barring limitations on section 2.2.4).

Another important part of this tool is its extendability, as the current tool could be the base for a more complex framework. This can be done by implementing the multiple improvements mentioned (section 2.3), by simply logging more and more complex values or expressions, performing deeper analyses or even by combining it with machine learning tools.

Knowledge required

Most of the programming done to create this tool is dedicated to generate the control-flow graph and to instrument classes with the ASM library. Most topics utilized are covered in a typical Bachelor on Computer Science; in subjects such as Compiler Design or Programming Languages. Other topics required to understand and extend this program include: the byte-code format and execution model, and by extension the functioning of the JVM, including class-loading mechanisms; instrumentation and reflection tools in Java (specifically in the form of Java Agents); and an understanding of the ASM byte-code manipulation library. The last one is described in detail in their user guide[1]. The guide acts both as an encyclopedic resource to be consulted on specific topics and as an introduction to each of the APIs that are exposed by it.

[1]Published on the ASM project homepage: `https://asm.ow2.io/asm4-guide.pdf`

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

AUTOMATIC GENERATION OF TEST CASES TO IMPROVE CODE COVERAGE IN JAVA

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
GALINDO JIMÉNEZ	CARLOS SANTIAGO

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 05/09/2018	Santiago

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.