SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2007; **17**:3–39
Published online 11 May 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.348

Integrating techniques and tools for testing automation

Macario Polo*,†, Sergio Tendero and Mario Piattini

University of Castilla-La Mancha, Paseo de la Universidad, 4, E-13071 Ciudad Real, Spain

SUMMARY

This article presents two tools to generate test cases, one for Java programs and one for .NET programs, as well as a structured testing process whereby such tools can be used in order to help in process automation. The main innovation in this work is the joint use of diverse techniques and technologies, which have been separately applied to test automation: reflection to extract the class structure; regular expressions to describe test templates and test cases; JUnit and NUnit as test case execution frameworks; mutation and MuJava as test case quality measurers; serialization to deal with the parameters of complex data types; and once more, reflection, to facilitate the test engineer in the task of writing the oracle. Finally, the article presents an experiment carried out to validate the work. Copyright © 2006 John Wiley & Sons, Ltd.

Received 6 July 2005; Accepted 1 March 2006

KEY WORDS: testing process; testing automation; JUnit; mutation

1. INTRODUCTION

Recent surveys have highlighted the lack of automation of software testing tasks in most software organizations [1–6]. According to Meudec [5], three main categories of automation in software testing can be distinguished: (1) administrative tasks (recording of test specifications, test report generation); (2) mechanical tasks (running and monitoring, capture/replay facilities); and (3) test generation tasks.

Regarding test generation, Ball *et al*. [7] list three research approaches: (1) given some source code, automatically generate test inputs to achieve a given level of statement, branch or path coverage; (2) given the tester's knowledge of some source code and its intended behaviour, develop algorithms that automatically generate inputs and check outputs; (3) given a formal specification, automatically generate test cases for an implementation of that specification.

*Correspondence to: Macario Polo, University of Castilla-La Mancha, Paseo de la Universidad, 4, E-13071 Ciudad Real, Spain.
†E-mail: macario.polo@uclm.es

Contract/grant sponsor: Ministerio de Educación y Ciencia/FEDER (MÁS Project); contract/grant number: TIC2003-02737-C02-02

In addition to other factors, the presence or absence of formal specifications of the program under test has a strong influence on the automation of the testing process. Thus, the more formal the specification of a program (in the sense of *Z* or algebraic specifications), the more capability there is to automate the testing process [7]. However, formal specifications are expensive to write and maintain [8], difficult to apply in practice and usually unavailable in industrial software [7]. Thus, in conventional testing practice, the programmer writes the tests for a new system feature after the corresponding production code for that feature [9]. As the program code is a common source for tests [10], the program under test is used to generate test cases, which are executed on a computer. The correctness of the program is checked by comparing the actual output of every test case with the expected output.

Over the last few years, the agile development community has implemented various frameworks to automate the software testing process, commonly known as X-Unit, that are based on this principle of comparing the obtained with the expected output, and that have quickly reached high popularity: in fact, many development environments have 'plug-ins' and 'wizards' to facilitate X-Unit testing (e.g. Oracle JDeveloper, Eclipse, Microsoft Visual Studio .NET, etc.). Given K, the class under test (CUT), the basic idea of X-Unit is to have a separate test class, *TestK*, containing test methods that exercise the services offered by K: thus, one of the first instructions of each test method will probably build a new instance o of K; between the remaining instructions, the test method will exercise on o some of the services offered by K. Somewhere in the test method, the test engineer adds code for checking that the state of o (this is, the *obtained* object) is consistent with the expected state (see Figure 1).

One important benefit of X-Unit environments is that test cases, that are saved in separate files, can be re-executed when the CUT is changed, thus being useful as regression test cases. A drawback is the effort required for writing test cases and that, moreover, even with a huge number of them, the test engineer will not have any guarantee of the coverage they achieve on the CUT.

In accordance with this discussion, Table I could be the possible representation of the level of automation achieved by X-Unit tools. In fact, related to the three categories of automation pointed out by Meudec [5], X-Unit environments make an important contribution to the automation of the mechanical tasks (running and monitoring, capture/replay facilities), partially automate administrative tasks (test specifications are appropriately recorded in separate files and reports about the number of faults detected can be extracted, setting aside important conclusions related to other testing metrics) and yet provide almost no facilities for test generation.

Thus, the positive impact that X-Unit has had on the testing practices of software development organizations can be increased with additional effort that leads to the automation of the manual tasks, especially test generation and better testing metrics. As a matter of fact, from the surveys referenced at the beginning of this section, the work by Ng *et al*. [6] is the one which shows the best results on test automation: 79.5% of surveyed organizations automate test execution and 75% regression testing. However, only 38 of the 65 organizations (58.5%) use test metrics, defect count being the most popular (31 organizations). Although the work does not present any data about the testing tools used, these results lead one to believe that most organizations are probably automating their testing processes with X-Unit environments.

This article describes a testing process for object-oriented software and a set of tools for supporting its automation. Neither the testing process nor the techniques implemented in the tools are completely new, although the proposal to combine all of them and to adapt the testing process to the tools is. After discussing related work (Section 2), this article is organized around the presentation of the different techniques and technologies adopted (Section 3), their integration into

DOI: 10.1002/stvr

```
package samples.results;

import junit.framework.*;
import samples.Account;
import samples.InsufficientBalanceException;

public class TestAccount extends TestCase {

  public void test1() {
    Account o=new Account();
    o.deposit(1000);
    assertTrue(o.getBalance()==1000);
  }

  public void test2() {
    try {
        Account o=new Account();
        o.deposit(300);
        o.withdraw(1000);
        fail("InsufficientBalanceException expected");
    }
    catch (InsufficientBalanceException e) {
    }
  }

  public void test3() {
    try {
        Account o=new Account();
        o.deposit(1000);
        assertTrue(o.getBalance()==1000);
        o.withdraw(300);
        assertTrue(o.getBalance()==700);
    }
    catch (Exception e) {
        fail("Unexpected exception");
    }
  }

  public static void main (String [] args) {
    junit.swingui.TestRunner.run(TestAccount.class);
  }
}
```

Figure 1. Three JUnit test cases.

DOI: 10.1002/stvr

Table I. Automation of unit testing obtained by X-Unit environments.

Test generation tasks		5%
Mechanical tasks	Running and monitoring	100%
	Replay facilities	100%
Administrative tasks	Record of test specifications	80%
	Generation of reports	10%

a structured testing process (Section 4), the automation of the process by means of a supporting tool (Section 5) and the first results of the application of the testing tool in an experiment (Section 6). The article finishes with the presentation of some conclusions in Section 7.

2. RELATED WORK

This section reviews some techniques and tools related to the goal of automating the unit testing process, and finishes with a discussion about how these techniques and tools can be used to define and automate a testing process. Section 2.1 discusses the usefulness of the 'method sequence specification' technique [11] for automating test generation tasks. Section 2.2 presents some basic concepts regarding X-Unit testing and discusses the characteristics of some X-Unit tools. Since the only testing metric provided by X-Unit tools is the number of faults found in the CUT, Section 2.3 makes a short presentation of mutation and discusses how this technique can be used to evaluate the quality of the test case suite. Section 2.4 explains how these techniques can be integrated into a structured testing process.

2.1. Method sequence specification, state machines and regular expressions

As noted in the previous section, formal specifications are a powerful mechanism for providing mathematical descriptions of software and for automating some costly tasks in the testing process. Intermediate situations between code and formal specifications are diagrammatic representations of the software, such as state machines, which have been widely applied in software test automation [7,10,12–14]. Several of these studies just cited propose coverage criteria for state machines, and then define algorithms to generate tests satisfying them. Offutt *et al.* [10], for example, process state machines described in software cost reduction (SCR) format and expand their predicates to generate test cases for covering transitions, full predicates, transition pairs and complete sequences. Hong *et al.* [13] translate state machines into inputs to a symbolic model checker and generate test cases for five coverage criteria.

In 1994, Kirani and Tsai introduced the idea of 'method sequence specification' as a way to document 'the correct order in which the methods of a class can be invoked by the methods in other client classes' [11]. A method sequence specification can be expressed by means of a regular expression, whose alphabet is composed of the set of public operations in the class. Since a state machine can be understood as an extended finite automata, these authors propose the generation of test cases for classes by extracting regular expressions from the state machine representing its behaviour.

DOI: 10.1002/stvr

In code-based test generation, regular expressions can be written to represent method sequences, independently of the previous existence of state machines. In fact, sequences of methods can be used to write test cases in testing frameworks such as JUnit or NUnit. Beck and Gamma [15] describe the basic steps for writing a test as follows. The test engineer must write: (1) code which creates the objects to interact with during the test; (2) code which exercises the objects; and (3) code which verifies the result.

2.2. X-Unit frameworks and tools

There exist several Web pages related to X-Unit technologies. From http://www.xunit.org, one can navigate for example to JUnit (for Java testing) and NUnit (for .NET testing). All these frameworks share the same testing philosophy: as was pointed out in Section 1, X-Unit test cases are written in separate classes containing test methods that exercise the services offered by the CUT: in its most simple form, an object o of the CUT is built in each testing method; then, a set of messages are sent to o; finally, the test engineer makes use of the framework libraries to check that the obtained state of o is consistent with the expected state.

Figure 1 shows three JUnit test cases for testing the behaviour of a bank account. The first test case checks that, after depositing 1000 monetary units into an account, its balance is 1000. In the second test case, 300 monetary units are deposited into the account, and then there is an attempt to withdraw 1000. In this case, the CUT will have correct behaviour if it throws an 'InsufficientBalanceException': thus, if the exception is thrown, the control of the program jumps into the 'catch' block and the test case is passed; if the exception is not launched, the 'fail' instruction is reached, which, in X-Unit, means that the test case has failed. The third test case checks that the balance of the account is 700 after depositing 1000 and withdrawing 300 monetary units. No exception is expected from this test case: thus, if an exception is thrown, the control of the program jumps into the 'catch' block and the framework executes the 'fail' instruction, which means that the test case has found a fault.

X-Unit was initially conceived for its application in 'test-driven development' [16], where test cases are written before code (also called 'test-first') and, together with the 'user stories', are the only documentation of the software project. However, the conventional practice is to use X-Unit in a 'test-last' dynamic, where tests are written and executed after code production [9]. For this approach, and taking into account the structure of X-Unit test methods, regular expressions based on the set of public operations of the CUT would be a suitable basis for generating test cases.

In fact, commercial tools related to X-Unit are designed to be applied after having written the CUT: setting aside the plug-ins and development environments that already consider X-Unit as an essential component, there are also stand-alone tools to automate test case generation: JTest is a tool by Parasoft to 'automate and improve Java Unit Testing' using JUnit [17]. In addition to other capabilities not directly related to JUnit, JTest loads the class and automatically builds a testing class containing blank test methods, one for each method in the CUT (if the CUT has n methods, the testing class will also have n testing methods). It is the responsibility of the test engineer to fill in the body of the testing methods, which can be partially done using a graphical interface (although in any case by hand). In order to achieve high coverage, it is common to exercise the CUT not only with calls to a single method of the CUT, but with a sequence of messages. For this same purpose of increasing the coverage, it is also usual for a JUnit (or NUnit) testing class to contain several versions of the same testing method, each one with different values passed as parameters. Thus, with JTest, the test engineer must still write a substantial amount of code to obtain a sufficient number of test cases.

DOI: 10.1002/stvr

Enhanced JUnit is a tool by SilverMark [18] that, in addition to other capabilities, generates a JUnit testing method for each public method in the CUT. Each testing method is composed of the creation of an instance of the CUT, a call to only the one method under test and a final piece of validation code. The tool can also trace object interactions, which makes a more powerful generation of test cases possible.

In general, all of these tools extract the structure of the CUT using reflection and show it on graphical interfaces that facilitate the selection of operations and the manual construction of test cases; however, although they offer powerful navigation capabilities, in the authors' opinion they fail in the automation of the test case generation step. In this way, existing tools only partially automate the process of writing test cases and the task still remains too expensive and time consuming.

The research community has also produced some relevant results in this area. Oriat [19] has built Jartege, a tool to generate random test cases for Java classes. The main drawback of this approach is the use of the Java Modelling Language (JML) [20], an academic proposal that has not transferred into the industrial world. JML is a specification language inspired by Eiffel, VDM and Larch. In JML, the operations of the CUT are annotated with assertions written in a particular notation, that are translated by a specific compiler into standard Java assertions, and later inserted into the code of the corresponding operation. The algorithm presented by Oriat generates testing methods composed of random sequences of calls to the constructors and public methods of the CUT, that are placed into a *try* block. Each testing method finishes with a *catch(Throwable)* block that captures the possible violations of the test case. Unfortunately, the algorithm generates many 'inconclusive' test cases, for which Jartege is not capable of detecting whether the program behaviour is correct or not. Moreover, the tool has problems dealing with parameters of complex types, since these are also randomly created.

2.3. Mutation as a coverage criterion

Even with a huge number of test cases (manually or automatically generated), the quality of the test case suite (i.e. its capability for detecting faults) is not guaranteed if the test engineer does not know the coverage achieved in the CUT. In fact, in the controlled experiment presented by Erdogmus *et al*. [9], there is no statistically significant correlation between the number of test cases and the program quality. Thus, a method to assess the quality of test cases is required.

For source code testing, Cornett [21] lists and explains some common criteria such as number of sentences, conditions, decisions, conditions/decisions, paths, data flow, etc. Mutation is also often used as a way of measuring the coverage of the CUT achieved by a suite of test cases and, in this way, the quality of test cases and of test case generation strategies [22–25]. Moreover, several studies have discussed its adequacy for validating such strategies with positive results [26,27].

A mutant is a copy of the program under test, but with a small change in its code. Mutants are usually generated by automated tools that apply a set of mutation operators to the sentences of the original program, thus producing a high number of mutants. Understanding a mutant to be a faulty version of the original program, a test case will be 'good' if it discovers the fault introduced, by 'killing' the mutant (i.e. the outputs of the original program and of the mutant are different for that test case). Thus, the higher the percentage of killed mutants one obtains, the higher the quality of the test cases. A test case suite is acceptable when the percentage of killed mutants is higher than a predefined threshold.

Although powerful, mutation is computationally a very expensive technique [28–32]. In fact, its three main stages (mutant generation, mutant execution and result analysis) require a lot of time and resources and, thus, researchers have devoted significant effort to reducing its costs.

DOI: 10.1002/stvr

(a) Regarding mutant generation, the main problem that arises is that almost each executable instruction of a program can be mutated with several mutation operators and, thus, the number of mutants generated for a normal program is huge. The cost of compilation of all mutants is also significant. For example, a simple program containing just one instruction such as *return* $a + b$ (where a, b are integers) may be mutated in at least 20 different ways ($a - b$, $a * b$, a/b, $0 + b$, $a + 0$, etc.). Offutt *et al*. [33] reported an experiment that, from a suite of 10 Fortran-77 programs ranging from 10 to 48 executable statements, produced from 183 to 3010 mutants. Mresa and Bottaci [30] utilized a set of 11 programs with a mean of 43.7 lines of code that produced 3211 mutants. To deal with this, in 1991 Mathur [34] proposed 'selective mutation', a line of study that has been continued by other authors: experiments have been conducted by Mresa and Bottaci [30], Offutt *et al*. [33] and Wong and Mathur [35] in order to find a set of sufficient mutant operators to decrease the number of mutants generated without information loss. Mresa and Bottaci [30] and Wong and Mathur [35] also investigated the power of randomly-selected mutants and compared this with selective mutation.

(b) Regarding mutant execution, the problem is the huge number of programs that must be executed. Both the original program and the mutants are executed with each test case. Later, the outputs of all of the mutants are compared with the original program output. According to Ma *et al*. [36], research in this area has proposed the use of non-standard computer architectures [37] and weak mutation. In weak mutation, the state of the mutant is examined immediately after the execution of the modified statement, considering the mutant to be killed even if the incorrect state is not propagated to the end of the program. Weak mutation was initially introduced by Howden [38] in 1982, and has received further attention from Offutt and Lee [39]. Another research line was proposed by Hirayama *et al*. [40], who proposed the reduction of the number of test cases to be executed by means of a prioritization of the program functions. Kim *et al*. [24] analysed the effectiveness of several strategies for test case generation with the goal of finding which one manages to kill more mutants.

(c) Regarding result analysis, the major difficulties appear to be with the detection of functionally equivalent mutants (mutants that cannot be killed by any test case and that will remain forever live). Manual detection is very costly, although Offutt and Pan [41] have demonstrated that it is possible to detect almost 50% of functionally equivalent mutants automatically.

The MuJava tool [36] is an efficient mutation system for Java programs: it uses bytecode translation and a technique called Mutant Schemata Generation (MSG) to produce the mutants, which reduces the cost of mutant generation; for mutant execution, MuJava executes all test cases against the CUT and the mutants using MSG and also a technique based on BCEL (the Byte Code Engineering Library); for result analysis, MuJava automatically compares the output of test cases for the CUT and for the mutants, reporting on the number of killed and live mutants. The tool facilitates the detection of functionally equivalent mutants by showing the original class and the mutant in separate text areas, highlighting the mutated instruction.

MuJava test cases are very similar to X-Unit test cases: they are also implemented as test methods that are grouped into separate classes. Each test method also consists of instructions to create and manipulate an instance of the CUT. However, whereas JUnit methods return a *void* type, MuJava methods return a *String* type, which corresponds to the string representation of the instance manipulated in the test method. Figure 2 shows the same test cases of Figure 1, but now rewritten in the MuJava format: MuJava executes the class with the test methods against the original program and its mutants,

DOI: 10.1002/stvr

```
package samples.results;

import samples.Account;
import samples.InsufficientBalanceException;

public class MujavaTestAccount extends TestCase {

  public String test1() {
    Account o=new Account();
    o.deposit(1000);
    return o.toString();
  }

  public String test2() {
    try {
        Account o=new Account();
        o.deposit(300);
        o.withdraw(1000);
        return "InsufficientBalanceException expected";
    }
    catch (InsufficientBalanceException e) {
       return e.toString();
    }
  }

  public String test3() {
    try {
        Account o=new Account();
        o.deposit(1000);
        o.withdraw(300);
        return o.toString();
    }
    catch (Exception e) {
       return "Unexpected exception";
    }
  }
}
```

Figure 2. The test cases of Figure 1, now in MuJava format.

compares the strings provided as results and registers the number of live and killed mutants. The goal of MuJava is not the finding of faults in the CUT: it is a mutation tool to generate and execute mutants, and for obtaining mutation results. Thus, its test methods do not contain the assertion instructions that are included in the JUnit test methods.

DOI: 10.1002/stvr

Table II. A test template, some parameter values and the test cases produced.

Test template	Test values		Test cases	
	Method	Parameter	JUnit format	MuJava format

`Account()`
`deposit(double)`
`withdraw(double)`
	`deposit`
	
	`withdraw`
		`double {0, 100}`
		
		`double {-100, 300}`
			`public void test1() {`
			` Account o=new Account();`
			` o.deposit(0);`
			` o.withdraw(-100);`
			` assertTrue(...);`
			`}`
			`public void test2() {`
			` Account o=new Account();`
			` o.deposit(100);`
			` o.withdraw(-100);`
			` assertTrue(...);`
			`}`
			`...`
				`public String test1() {`
				` Account o=new Account();`
				` o.deposit(0);`
				` o.withdraw(-100);`
				` return o.toString();`
				`}`
				`public String test2() {`
				` Account o=new Account();`
				` o.deposit(100);`
				` o.withdraw(-100);`
				` return o.toString();`
				`}`
				`...`

2.4. Integration of techniques

The intention of the three previous sections was to provide the reader with an overview of the different possibilities that current techniques and tools have for testing automation. Given the focus on the Java programming language and on the two testing tools reviewed, and taking into account the similarities between JUnit and MuJava test cases, the same regular expression can be used to generate the same test cases for both tools: thus, MuJava will be used to assess the quality of the JUnit test cases.

More generally, the idea is also valid for other programming languages. For the Microsoft .NET languages, NUnit is a tool equivalent to JUnit, sharing its principles of test case structure and execution. Unfortunately, to the authors' best knowledge, there is no mutation system for .NET, although the authors are developing a mutation system compatible with NUnit.

The proposal described in subsequent sections integrates the previously reviewed techniques, with the goal of improving X-Unit test case generation and the measurement of its quality through mutation. Two tools (for Java and for the .NET languages) have been developed that, using reflection, extract the structure of the CUT and facilitate the writing of regular expressions, with an important contribution to the automation of the subsequent test case generation. Given a class to be tested, the user provides a 'method sequence specification' [11] of the CUT by means of a regular expression; then, using a standard Java or .NET library, the spanning tree of the finite automata corresponding to the regular expression is analysed and used to generate test templates. A test template is a sequence of operations of the CUT with no values that must be later combined with actual test values to generate testing files. Table II illustrates this concept: the template is a list of signatures of operations in the CUT; each parameter of each operation in the template has assigned a set of test values. Test cases are obtained by generating all possible combinations of the methods in the test template with the corresponding test values. Let $ts = (op_1, op_2, \ldots, op_n)$ be a test template composed of calls to the operations $op_1, op_2, \ldots, op_n$, and let $p_{m1}, p_{m2}, \ldots, p_{mn}$ be the number of parameters in $op_1, op_2, \ldots, op_n$; then the number of test cases generated from ts is $p_{m1} \times p_{m2} \times \cdots \times p_{mn}$.

DOI: 10.1002/stvr

For Java programs, test cases will be written in JUnit or MuJava formats (containing the same test cases, but each with a different structure, appropriate for its corresponding tool). For .NET, test cases are written in NUnit format [42] and in any of the following languages of Visual Studio: Visual Basic, C# and J#. For Java, JUnit and MuJava are complementary files, in the sense that MuJava test cases measure the quality of the JUnit test cases. In the .NET case, it is not yet possible to measure the quality of the generated tests although, as previously stated, a compatible mutation tool is being developed.

Primitive values can be given by hand during the testing session or be read from a file. Values of complex types must have been previously serialized by the user, and will be deserialized during test file writing.

In order to reach a higher level of quality in test cases, it is very common to exercise the CUT with several versions of the same testing methods, only differentiated by the set of values passed as parameters. It is also interesting to test the class invoking its services in different orders. Regular expressions are a very good way to specify the messages composing each testing method. This technique, therefore, constitutes a highly appropriate alternative to the test case generation strategy of the commercial and research tools previously discussed, and industry may take advantage of the integration ideas summarized in this article and put them into practice in its tools.

3. SUPPORTING TECHNOLOGIES

This section provides brief descriptions of JUnit and MuJava, two third-party tools, suitable for integration into the proposed testing process.

3.1. A brief introduction to X-Unit

In order to explain the testing methodology using X-Unit, this section presents some characteristics of JUnit, the X-Unit Java framework, which can be used as a basis for understanding the remaining technologies.

To write a class for testing a domain class, the software engineer must write a specialization of *TestCase*, one of the main classes included in the framework. In turn, *TestCase* is a specialization of *Assert*, a class that contains many *assert* operations (Figure 3), that are used to check the state of the instance under test. Using reflection, the JUnit engine executes all of the methods in the testing class whose names start with 'test'. Each testing method exercises some services of the CUT on one instance. Finally, the obtained instance is compared with the expected instance using some of the *assert* methods. If none of the methods fail (i.e. they do not find any fault in the CUT), the JUnit user interface shows a green bar; if at least one testing method finds an error, the user interface shows a red bar.

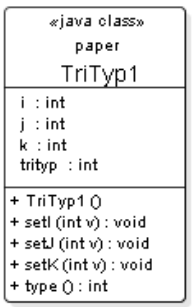
Consider a Java version of the triangle-type problem (TriTyp) [43], commonly used as a benchmark example in testing papers. Figure 4 shows the UML representation of the TriTyp class. It contains three fields i, j, k representing the lengths of the three sides of a triangle, whose values are given by the three *setI*, *setJ*, *setK* methods; the triangle type is decided in the body of the *type* method, which assigns the appropriate value to the *trityp* field.

In JUnit, possible test cases would consist of a call to the class constructor, calls to the three setter methods and, then, a call to some of the *assert* methods inherited from *Assert*. Figure 5 shows two JUnit testing methods: *test1* checks that a triangle with three sides of length 5 is equilateral;

DOI: 10.1002/stvr

```
void assertTrue(String, boolean)
void assertFalse(String, boolean)
void assertEquals(String, Object, Object)
void assertEquals(String, String, String)
void assertEquals(String, float, float, float)
. . .
void assertNotNull(Object)
void assertNull(Object)
void assertSame(String, Object, Object)
void assertNotSame(String, Object, Object)
void assertTrue(boolean)
void assertFalse(boolean)
void assertEquals(Object, Object)
void assertEquals(float, float, float)
void assertEquals(boolean, boolean)
. . .
void assertNotNull(String, Object)
void assertNull(String, Object)
void assertSame(Object, Object)
void assertNotSame(Object, Object)
void fail(String)
void fail()
```

Figure 3. Some operations in the Assert class.



Possible values of the *trityp* field:

- 1 if scalene;
- 2 if isosceles;
- 3 if equilateral;
- 4 if not a triangle.

Figure 4. UML representation of the TriTyp class and possible values.

test2 checks that a triangle with a side of length zero is not a triangle. The right-hand side of the figure shows the graphical interface of JUnit: the two test cases have been executed and the bar shows with the green colour that neither errors nor failures in the CUT have been found by these test cases.

It is possible to change the code in the CUT to throw exceptions if some of the parameters passed to the setter methods are less than or equal to zero. In this case, the testing methods in Figure 5 would not compile because the exception should be either caught or thrown by *test1* and *test2*. In the case of *test2*, the correct behaviour is that the sentence *t.setK(0)* throws the exception; otherwise it is incorrect. To deal with these situations, the *Assert* class includes the *fail* method, which forces the red bar to appear: sentences in the *test1* method of Figure 6 should not throw any exception; so, if one occurs, the exception is captured in the catch block and the bar is forced to appear in red colour by the *fail* instruction. The *test2* method should throw the exception after executing *t.setK(0)*, since this parameter must not be accepted. If it throws the exception, the program control jumps into the catch block that manages the exception as appropriate; if it does not throw the exception, then the program control continues to the *fail* sentence, with the red bar appearing in the JUnit window.

DOI: 10.1002/stvr

```
public void test1()
{
   TriTyp1 t=new TriTyp1();
   t.setI(5);
   t.setJ(5);
   t.setK(5);
   super.assertTrue(t.type()==3);
}

public void test2()
{
   TriTyp1 t=new TriTyp1();
   t.setI(5);
   t.setJ(5);
   t.setK(0);
   super.assertTrue(t.type()==4);
}
```

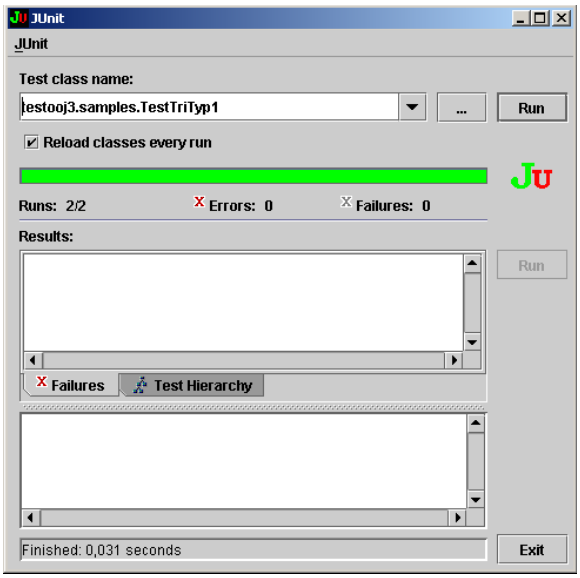


Figure 5. Two JUnit test cases for the TriTyp problem.

```
public void test1()
{
  try {
    TriTyp2 t=new TriTyp2();
    t.setI(5);
    t.setJ(5);
    t.setK(5);
    super.assertTrue(t.type()==3);
  }
  catch (Exception ex)
  {
    fail("This test case shouldn't " +
    "fail: it is equilateral");
  }
}
```

```
public void test2()
{
  try {
    TriTyp2 t=new TriTyp2();
    t.setI(5);
    t.setJ(5);
    t.setK(0);
    fail("This test case should have failed");
  }
  catch (Exception ex)
  {
  }
}
```

Figure 6. The test cases of Figure 5, now adapted to the new implementation of the CUT.

DOI: 10.1002/stvr

```
public String test1()
{
   TriTyp1 t=new TriTyp1();
   t.setI(5);
   t.setJ(5);
   t.setK(5);
   return t.toString();
}
```

```
public String test1()
{
  try {
    TriTyp2 t=new TriTyp2();
    t.setI(5);
    t.setJ(5);
    t.setK(5);
    return t.toString();
  }
  catch (Exception ex)
  {
    return "Error: " + ex.toString();
  }
}
```

Figure 7. Two MuJava test cases.

As is well known, there are many different possibilities for obtaining complete testing of the TriTyp problem. Knowing the interface of the class, it is possible to write regular expressions to generate test cases automatically.

Explanations presented in this section are completely valid for other X-Unit frameworks, such as NUnit, the .NET testing framework. Moreover, the two tools implemented share the same principles and algorithms. The only difference is that the .NET tool does not generate test cases for any mutation tool.

3.2. MuJava

Test cases in MuJava also consist of sequences of calls to services, but they are slightly different from those of JUnit: a test case in MuJava is a method returning a string, which may be the string representation of the object under test.

The two test cases in Figure 7 are the MuJava test cases corresponding to the *test1* methods in Figure 5 and Figure 6. Note that, now, both test methods return a string. The example on the right-hand side catches the possible exception. MuJava executes the class with the test methods against the original program and its mutants, compares the strings provided as results and registers the number of live and killed mutants.

4. DEFINITION OF THE TESTING PROCESS

Mutation is used to determine the adequacy of the test cases for testing the CUT. However, mutation must be used once the test engineer has checked that the test case suite does not find any fault in the CUT. Otherwise (i.e. if a test case finds a fault), the CUT must be corrected and the test cases re-executed. This process model requires less time and effort than others, such as the ‘traditional’ and

DOI: 10.1002/stvr

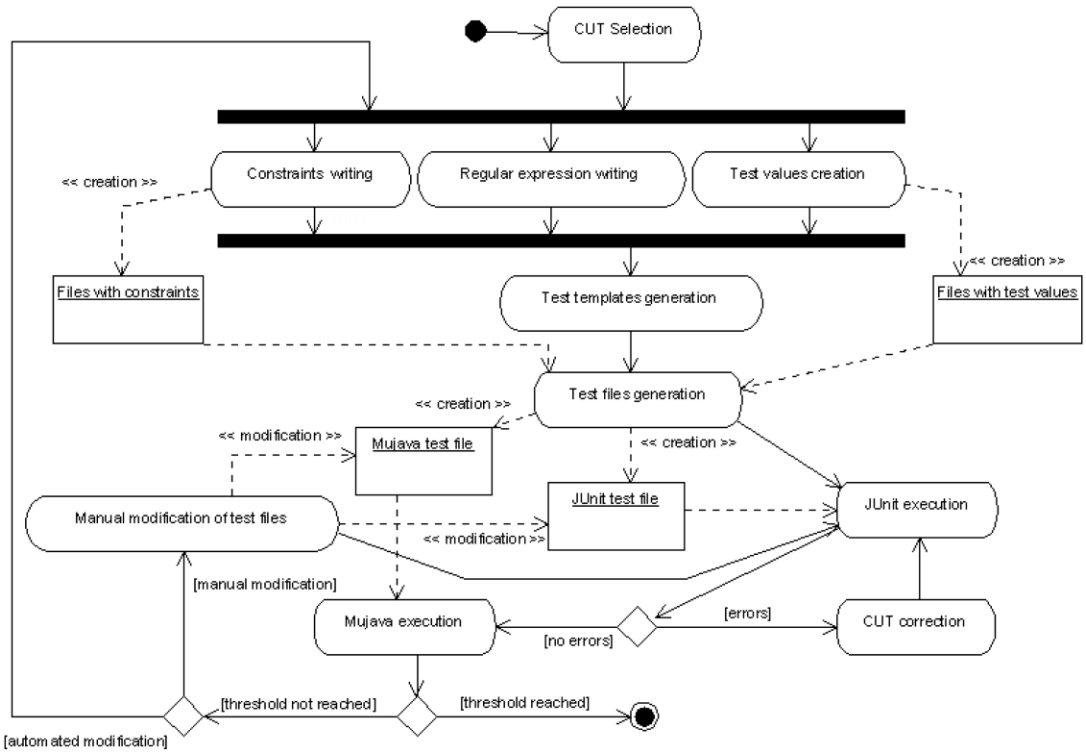


Figure 8. Proposed testing process.

the 'new' mutation testing processes described by Offutt [44]: in these process models, test cases are executed against the mutants even before knowing whether the CUT is correct or not.

In order to automate test generation, the test engineer must write a regular expression representing the structure of the test templates. The combination of these with the test values will generate the test cases. Since X-Unit also requires assert code, the operations in the CUT must be annotated with assertions. Assertions must be valid for all of the possible X-Unit test cases generated: thus, a possible assertion for the *deposit(amount:double)* operation of an *Account* class could be:

$$assertTrue(obtained.getBalance()==balancePre+amount)$$

where *obtained* is the name of the instance under test in all test methods, and *balancePre* saves the result of executing the *getBalance()* operation on *obtained* before the call to *deposit*.

Figure 8 shows the testing process proposed.

(1) In the first step, the CUT is selected and its set of public operations is extracted.
(2) Then, the tester can: (a) write a set of constraints for each operation to be tested (these constraints will be used later to write the assertions in the X-Unit testing methods); (b) write a regular

DOI: 10.1002/stvr

expression for the further generation of test cases; (c) create test values that will be passed as parameters in the calls to operations of the CUT.

(3) The regular expression written in the previous step is used to generate test templates, which will be later combined with the testing values to generate test cases.
(4) X-Unit test files are generated combining test templates, test values and operation constraints, elements created in the previous steps. MuJava files do not require constraints to be generated.
(5) The X-Unit file generated in the previous step is executed using the appropriate testing framework (JUnit, NUnit, etc.). If test cases highlight the presence of faults in the class, these must be corrected and the X-Unit file should be re-executed until no further faults are found.
(6) Once the CUT presents no faults, it is time to check the coverage achieved by the test cases, which is done by executing the same set of test cases in a different environment. In the case of Java programs, mutants must be generated and executed against test cases within the appropriate environment (in this case, MuJava). Obviously, the test engineer must detect and remove functionally equivalent mutants.
(7) If the threshold of killed mutants is reached, then the testing process can be finished; otherwise, new test cases must be written for both the X-Unit and the mutation system, which can be done manually or automatically. The X-Unit file should be executed first, meaning that there is a return to step (5). The stipulated threshold determines the minimal percentage of mutants that must be killed in order to consider that the testing phase is satisfactory.

5. DESIGN OF THE TOOLS

Two tools have been implemented to generate test cases: one deals with Java code and produces JUnit and MuJava files; the other deals with .NET compiled code and produces NUnit files.

Java and .NET include the possibility of extracting and knowing many details of the structure of classes by means of reflection. In Java, the *java.lang.Class* class is the starting point to access the set of operations and fields in the CUT, which is later used to write the regular expression to generate the tests. In .NET, the equivalent class is *System.Type*, which also has a wide set of operations for the same goal.

Figure 9 shows some of the members of some reflective Java classes. As can be seen, *Class* includes operations to recover its set of constructors, methods and fields. When, for example, a method is recovered, it is possible to know its return type (note the *getReturnType* operation in the *Method* specification of Figure 9), the types of its parameters (using the *getParameterTypes* operation), the exceptions it throws or, even, to invoke the method on an instance with a set of parameters (by means of the *invoke* operation).

The equivalent structures in .NET (*Type, ConstructorInfo, MethodInfo* and *FieldInfo*) offer similar functionalities to the programmer.

The following sections explain in detail how to carry out the different steps of the proposed testing process (Figure 8), as well as some implementation techniques.

5.1. CUT selection and extraction of public operations

In the Java tool, the CUT must be accessible through the CLASSPATH environment variable and can be packaged into a JAR or ZIP file; in .NET, the CUT structure is extracted and then processed from

DOI: 10.1002/stvr

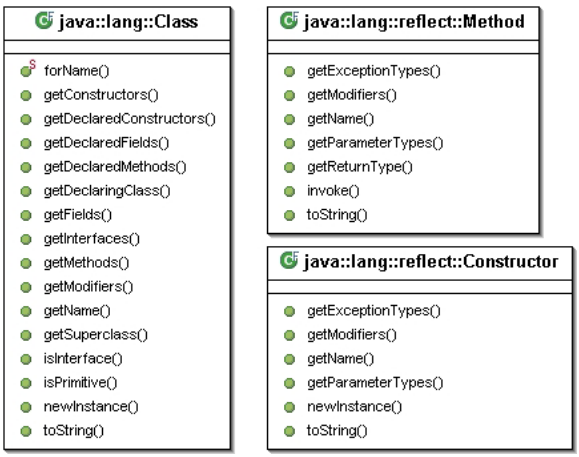


Figure 9. UML representation of the *Class* and *Method* Java classes.

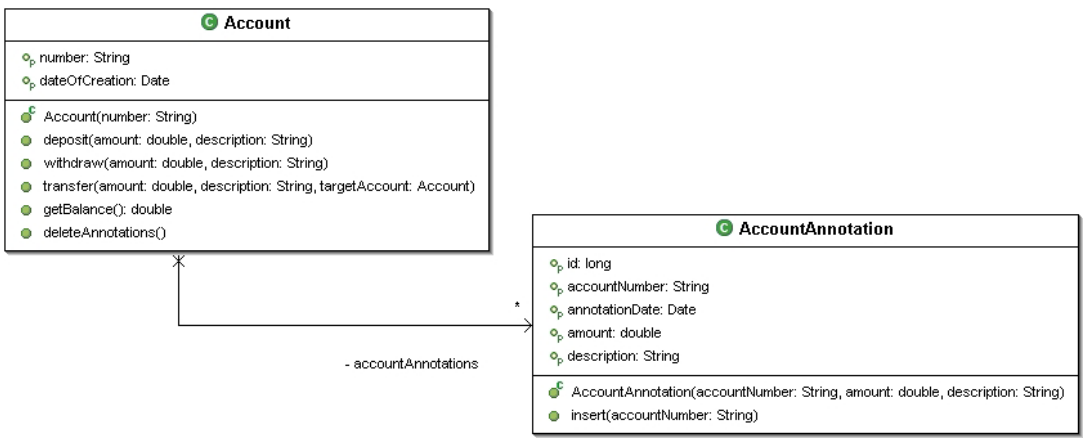


Figure 10. Structure of a banking system.

the compiled file where it has been deployed: either dynamic link libraries (DLL) or executable files (EXE). Using an instance of *Assembly* (a class defined in the *System.Reflection* .NET namespace), the tool shows the list of classes contained in the selected file, one of which will be the CUT.

In either case, the CUT is instantiated (by means of a Java *Class* or a .NET *Type*) and, using the capabilities of reflection, its structure is extracted and shown. In the remaining sections, the Java testing tool is presented, applying it to the example of Figure 10, that represents a fragment of the structure of a banking system, where each account may have several annotations.

DOI: 10.1002/stvr

In addition to others, *Account* has the operations whose code appears in Figure 11.

- *Account(String number)* looks in the database for the account whose number is the value passed as parameter and, if it is found, assigns the instance fields the corresponding values; otherwise, it throws a *NonExistingAccount* exception. Since the connection with the database can fail, it may throw an *SQLException*.
- *deposit(double amount, String description)* creates a new annotation, that is associated with the corresponding bank account, and inserts it into the database. If *amount* is not greater than zero, then the method throws an *IllegalArgumentException*.
- *withdraw(double amount, String description)* creates a new annotation associated with the account and saves it in the database. It can throw: (1) an *IllegalArgumentException* if *amount* is less than or equal to zero; (2) an *InsufficientBalanceException* if the account balance is less than *amount*; and (3) by means of *getBalance*, a *NonExistingAccountException*. If there are problems with the database connection, it can also throw an *SQLException*.
- *transfer(double amount, String description, Account targetAccount)* produces three annotations: two on the account that executes the method (a withdrawal for the *amount* and a *commission* for the transfer) and one on the *targetAccount*. This method can throw the same exceptions as *deposit* and *withdraw*, as well as *InvalidAccountNumber*.
- *deleteAnnotations()* deletes all the annotations in the corresponding *Account*. This operation is required to start the execution of each test case with the account instance in the same state: since accounts are persistent objects (annotations are saved in the database), the execution of the *deposit* operation on an account will change its state in the database; later, when the same test case is executed on a mutant affecting the same account, the account state (i.e. its balance) will be different. Thus, for testing purposes, the annotations of each account must be deleted after calling the aforementioned *Account(String number)* constructor.

5.2. Regular expression writing, processing and test template generation

Figure 12 shows the main screen of the Java tool. The CUT has been selected navigating the class path after pressing the button labelled 'Select class'. Then, its public constructors and methods are loaded and shown in the tree appearing on the left-hand side. Regarding the regular expression, the operations in the CUT are referenced with letters: the first operation is 'A', the second is 'B', etc. If there are more than 'Z' operations, then the successive ones are mapped to combinations of letters starting with '(AA)', '(AB)', etc. The tool user can write the corresponding letters, or select a node and press the button under the tree, or double click on the corresponding node. If the user does a double click on the regular expression, then the sequence of actual calls is shown in the messages area (in the bottom part of the screen).

In the Java tool, regular expressions are analysed using *java.util.regex.Pattern* class (and the *System.Text.RegularExpressions* namespace in the .NET tool); thus, they must be written using its notation[‡].

[‡]A summary of Java regular-expression notation is found at http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html [10 May 2005].

DOI: 10.1002/stvr

```
public Account(String number) throws SQLException, NonExistingAccount {
    String SQL="Select Number, CustomerId, DateOfCreation from Account where Number=?";
    Connection bd=null;
    PreparedStatement p=null;
    try {
       ...
       ResultSet r=p.executeQuery();
       if (r.next()) {
          ... // Assignment of columns to fields of "this"
       } else throw new NonExistingAccount();
    }
    catch (SQLException ex) {
       throw ex;
    }
    finally {
       bd.close();
    }
}

public void deposit(double amount, String description)
          throws IllegalArgumentException, SQLException {
    if (amount<=0) throw new IllegalArgumentException();
    AccountAnnotation a=new AccountAnnotation(this.number, amount, description);
    a.insert(this.number);
}

public void withdraw(double amount, String description) throws IllegalArgumentException,
       SQLException, NonExistingAccount, InsufficientBalanceException {
    if (amount<=0) throw new IllegalArgumentException();
    if (amount>getBalance()) throw new InsufficientBalanceException();
    AccountAnnotation a=new AccountAnnotation(this.number, -amount, description);
    a.insert(this.number);
}

public void transfer(double amount, String description, Account targetAccount) throws
                    IllegalArgumentException, SQLException, NonExistingAccount,
                    InsufficientBalanceException, InvalidAccountNumber {
    if (targetAccount.getNumber().equals(this.getNumber()))
       throw new InvalidAccountNumber();
    if (amount<=0) throw new IllegalArgumentException();
    double commission=0.02*amount;
    if (commission<3) commission=3;
    if (amount+commission>getBalance()) throw new InsufficientBalanceException();
    this.withdraw(amount, "Transference");
    this.withdraw(commission, "Commission");
    targetAccount.deposit(amount, "Transference");
}

public void deleteAnnotations() throws SQLException {
    String SQL="Delete from AccountAnnotation where AccountNumber=?";
    ...
}
```

Figure 11. Some code of some operations of *Account*.

DOI: 10.1002/stvr

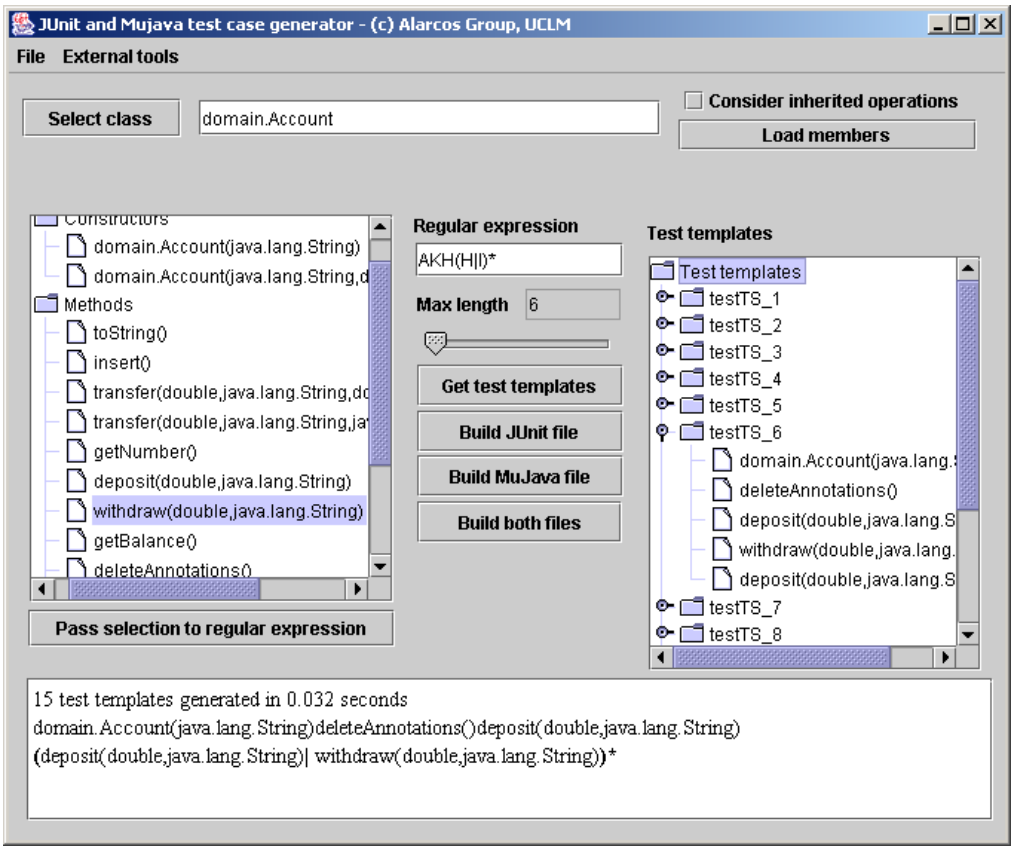


Figure 12. Main window of the Java tool.

The regular expression in the figure is:

$$AKH(H|I)^*$$

This denotes (see the class structure in the left-hand tree of Figure 12) that test cases must start with a call to the constructor *domain.Account(String)*, which must be followed by a call to *deleteAnnotations()* (operation 'K'), another one to *deposit(double, String)* and combinations of calls to the methods 'H' and 'I' (respectively *deposit* and *withdraw*). The maximum length of test cases (i.e. the maximum number of calls) can be limited (up to six, in this example).

When the user presses the 'Get test templates' button, the corresponding set of 'test templates' is obtained and shown in the tree on the right-hand side. Test templates contain the operations that will later be combined with the values and assertions given by the user to generate the test file. Adapting the work of Chow [45], templates are generated calculating the spanning tree of the regular expression:

DOI: 10.1002/stvr

```
Set buildTemplates(Class cut, String regularExpression, int maxLength)
        Set result=∅
        int previousPosition=1
        for I=1 to maxLength
               buildTemplates(cut, previousPosition, regularExpression, result)
        end_foreach
        foreach template in result.templates
               if regularExpression does not accept template.regularExpression()
                       result.remove(template)
               end_if
        end_foreach
        return result
end

void buildTemplates(Class cut, byref int previousPosition,
                                    String regularExpression, byref Set result)
        foreach template in result.templates
               foreach operation in cut.operations
                       if regularExpression contains operation
                              currentTemplate.addOperation(operation)
                              result = result ∪ { template }
                       end_if
               end_foreach
        end_foreach
        previousPosition=i
end
```

Figure 13. Algorithm for generating test templates.

the function in charge is the first version of *buildTemplates* shown in Figure 13: *result* saves the set of templates. In the first loop, and by calls to the second version, the algorithm progressively fills *result* with test templates of lengths 1, 2, . . . , *maxLength*. When templates are generated (and by means of the aforementioned regular expression libraries), the second loop filters the templates that do not fulfil the *regularExpression*. When the second function is called with 1 as *currentLength*, templates of length 1 composed of the signature of operations included in *regularExpression* are added to *result*; each new time the second function is called, it adds a new operation signature to the previous set of templates. For brevity and clarity, details of reference controlling and object cloning have been removed.

5.3. Test values and assertions writing

Test templates must be combined with actual values and additional code to produce the testing methods in the generated testing class. Test values must be manually assigned to parameters by the user, for which he/she can apply his/her best knowledge about the system, as well as limit values of assertions, an approach that has been suggested in many studies [10, 46–49].

The additional code includes calls to the *assert* operations inherited from the *Assert* class (see Figure 3), but also other type of code that may be useful for testing purposes, such as variable declarations, conditional instructions, etc.

DOI: 10.1002/stvr

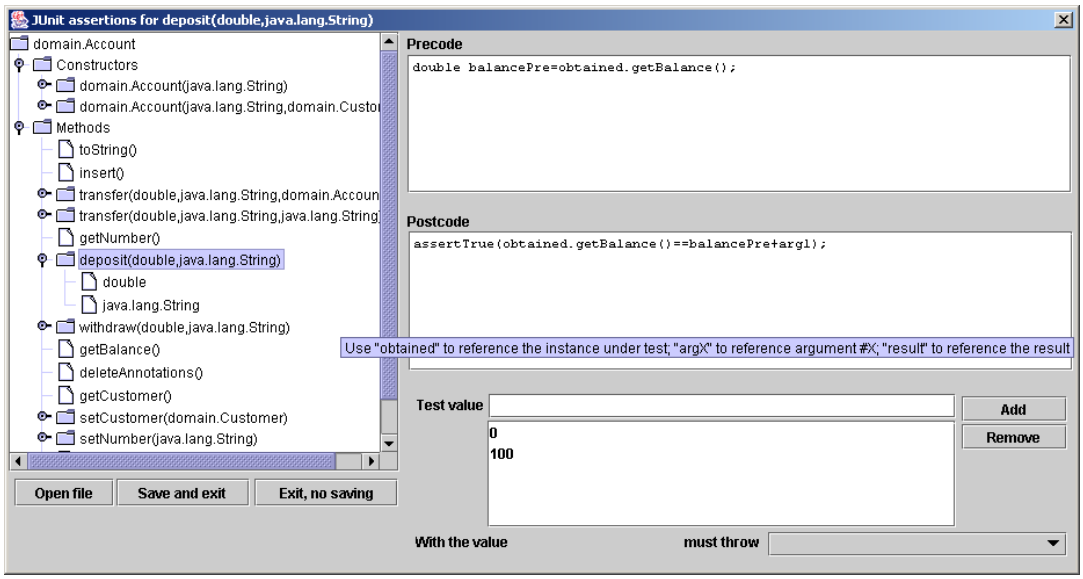


Figure 14. Giving values and assertions.

Both the testing values and the additional code are written by the tool user by hand using the window shown in Figure 14. In the 'precode' and 'postcode' areas, the user may write code that, in each testing method, will be put before and after the call to the business method that is selected in the left-hand tree. Note that the word *obtained* is used in the precode and postcode, since this is the name the tool assigns to the instance of the CUT in each testing method. In the same way, references to arguments are made using *argX*, where *X* represents the argument number in the method signature. When the operation returns a result, this is referenced with *result*.

Values for parameters of primitive data types are directly assigned in the tool interface shown in Figure 14; values of non-primitive data types must be previously serialized and saved in a specific location. Later, the tool 'deserializes' these objects and passes them as parameters to the testing methods. The use of serialized objects is explained in Section 5.5.

As has been illustrated in Figure 6, the correct behaviour of the CUT sometimes requires the throwing of an exception. By reflection, the tool extracts the exceptions that each method can throw and shows them to the test engineer (Figure 14). Thus, in this screen he/she may assign a given exception to a specific value of a method. For example, since the correct behaviour of *deposit* is the launching of an *IllegalArgumentException* with the zero value, this association of an exception with a test value can be established at the bottom right-hand side of the screen (Figure 15).

With these 'reserved words' (*obtained*, *result*, *arg1*, *arg2*, . . .) and this way of operating, the tool makes an important contribution to the description of a generic test oracle for the CUT, and partially supports the automation of testing result analysis, which are currently active areas of research [10].

DOI: 10.1002/stvr

Figure 15. Associating test values with exceptions.

5.4. Test case generation

When the testing values and the additional code have been assigned, the tool has all the information required to generate the test file. Testing methods are named using the name of the test template they proceed from, followed by the order of the testing method inside the template. The number of testing methods obtained from a test template depends on the number of testing values given to its parameters (Section 2.4).

The tool manages a metamodel representing all of the elements involved in the process. Figure 16 is a partial view of the metamodel, extracted after having reverse-engineered the main classes of the tool and having hidden some details: *TClass* represents the CUT, from which a set of *TestTemplate* elements are generated with the algorithm of Figure 13; once the class knows its templates, it builds the set of test cases (*TestCase* class); these are later translated into JUnit or MuJava test methods (respectively, *TJUnitMethod* and *TMujavaMethod*). The main difference between the two specializations of *TestMethod* resides in the *toString* operation, since each one must write the method into the format of the target tool. Finally, the test methods are added to the corresponding testing files and saved in the location pointed out in the tool setup. Since the number of test methods can be huge, they are not saved in main memory during the generation process, but each time a method is built, it is serialized and saved on disk. When all methods have been generated, the tool goes into a loop that reads each test method and adds it to the corresponding JUnit or MuJava test file.

Figure 17(a) shows the algorithm to generate the test cases corresponding to the test template passed as parameter. The first loop builds a tree whose nodes save the test values of the parameters of all operations, ordered by operation and, inside each operation, by parameter order (Figure 17(b) shows the tree corresponding to the *test_6* template). The second loop considers the branch corresponding to each leaf node in the tree (Figure 17(c) represents the branch corresponding to the left leaf, from top to bottom), and builds a test case for each branch, adding the parameters to the operations in the same order (Figure 17(d)).

Figure 18 shows the window containing all of the testing methods generated for this testing scenario. The code in the example corresponds to the *test_6_31* method, one of the test cases generated

DOI: 10.1002/stvr

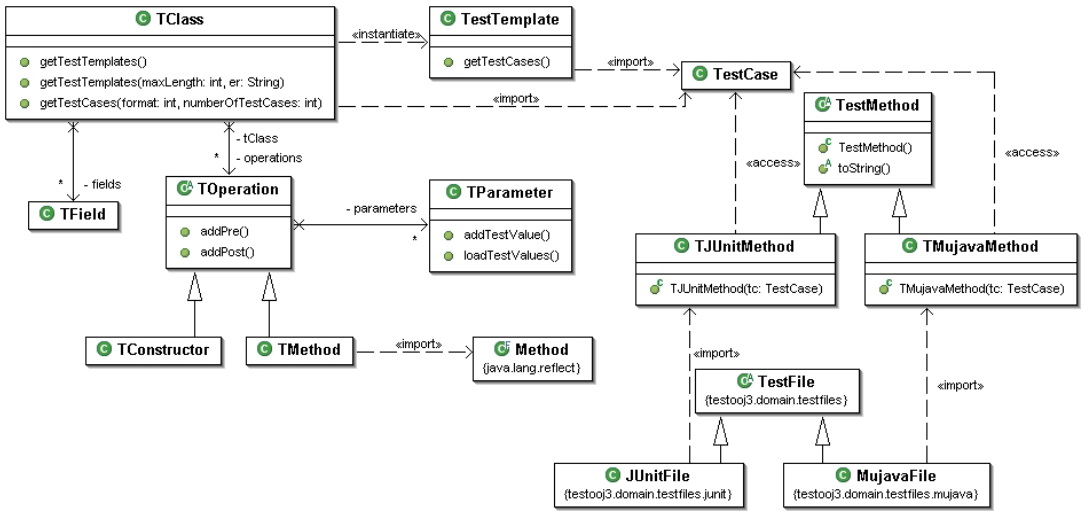


Figure 16. Partial view of the tool metamodel.

from the *TS_6* template (selected and expanded in the tree on the right-hand side of Figure 12). Note that, since the last value passed to *deposit* is zero, and this value had been marked to throw an *IllegalArgumentException* in Figure 15, the tool has structured the test method into three blocks as follows. (1) The first is a *try* block that attempts the execution of the CUT operations. If no exception is launched, then the *fail* instruction at the end of the *try* is executed and JUnit is instrumented to highlight the presence of a fault in the CUT. (2) If an *IllegalArgumentException* is thrown, the program control jumps into the *catch(IllegalArgumentException)* block and the test case is passed. (3) If another type of exception is thrown, the program control goes into the *catch(Exception)* block and forces JUnit to highlight the presence of a fault.

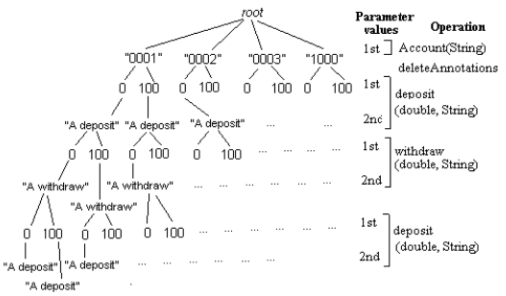
However, if it is executed, the test case will fail because, according to the specification of *withdraw* (Figure 11), it is not possible to withdraw 5000 monetary units from an account in which only 100 units have been deposited. Thus, the correct behaviour of the CUT for this test case is the throwing of an *InsufficientBalanceException*, which can be selected from the list placed at the top in order to change the test method structure. By reflection, that list is loaded with all the possible exceptions the methods in the test case can throw.

Without removing any of the testing methods in the previous figure, 2072 test cases are generated on a standard Pentium II computer in 10.8 seconds for the *Account* class using the regular expression of Figure 12, with four values for the string argument of the constructor, two values for the double parameter of *deposit*, four for the double of *withdraw* and one for their string parameters. The tool splits JUnit test cases into several files depending on the user preferences. Selecting a size of 600 KB per JUnit file, five test files are generated and written in the result location, named *TestAccount1* to *TestAccount5*. The first file contains 540 test cases, one of which is shown in Figure 18.

DOI: 10.1002/stvr

```
Set buildTestCases(TestTemplate template)
   Set result=∅
   Tree mainTree=new Tree()
   foreach operation in template.operations
      foreach testValue in operation.testValues
         Tree child=new Tree(testValue)
         if (mainTree.leafs = ∅)
            mainTree.addChild(child)
         else
            foreach leaf in mainTree.leafs
                 leaf.addChild(child)
            end_foreach
         end_if
      end_foreach
   end_foreach
   foreach leaf in mainTree.leafs
      NodeList branch=leaf.getBranchToRoot()
      TestCase tc=new TestCase()
      tc.name=template.name + "_" + i
      foreach operation in template.operations
         int counter=0
         foreach parameter in
                        operations.parameters
            node=branch.getNode(counter)
            tc.addParameter(
                 node.getTestValue(), op)
            counter = counter+1
         end_foreach
      end_foreach
      result = result ∪ { tc }
   end_foreach
   return result
end
```

(a)



(b)

"0001", 0, "A deposit", 0, "A withdraw", 0, "A deposit"

(c)

domain.Account("0001")
deposit(0, "A deposit")
withdraw(0, "A withdraw")
deposit(0, "A deposit")

(d)

Figure 17. Algorithm to generate test cases from a test template.

As can be seen in Figure 19, JUnit executes the test file in 27 seconds (which, in this example, includes the establishment of 2300 connections to the banking database). One of the test methods that has found a fault is *testTS_6_31*, since, as pointed out in the previous discussion, it has thrown an *InsufficientBalanceException* instead of *IllegalArgumentException*.

The tool saves all test methods on disk: thus, the test engineer may rebuild the test files and correct the test methods in the window of Figure 18 or, if there is a problem in the CUT, go to this to fix it and, then, re-execute the test cases.

The same test cases can be also generated in MuJava format (Figure 20). In order to avoid problems with huge files that could force the computer to throw an *OutOfMemoryException*, the tool builds more than one MuJava test file depending on the number of test cases, and puts them into the appropriate MuJava directory.

MuJava builds 82 traditional and 14 class mutants for *Account*. In the example, each MuJava test file has 300 test cases. This means that $(82 + 14) \times 300 = 28\,800$ executions of the test cases are made.

DOI: 10.1002/stvr

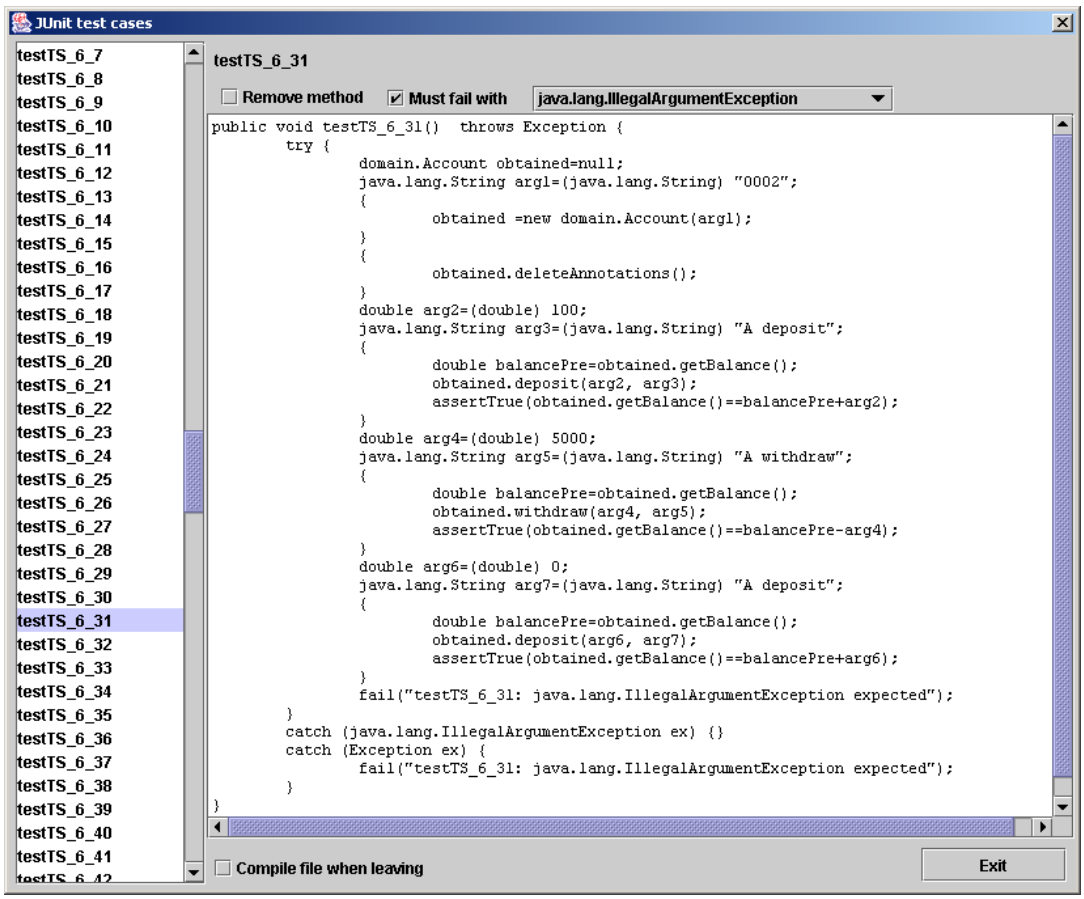


Figure 18. Test cases generated for JUnit.

In spite of these numbers, the percentage of killed mutants is small, since these test cases exercise only a subset of the methods in the CUT. Therefore, the removal of the mutants affecting non-tested operations is required. After removing these, 31 traditional and four class mutants still remain, and the testing file is executed against them. Figure 21 shows the results of the process: 90% and 50% of traditional and class mutants, respectively, are killed by the test suite. After examining with MuJava the live mutants, all of them are functionally equivalent: as an example, the code of the AOIS_5 mutant has added the ++ operator in the construction of the *AccountAnnotation* object (line 68), that increases the value of *amount*, but *after* the operation, therefore having no effect on the amount assigned to the annotation:

DOI: 10.1002/stvr

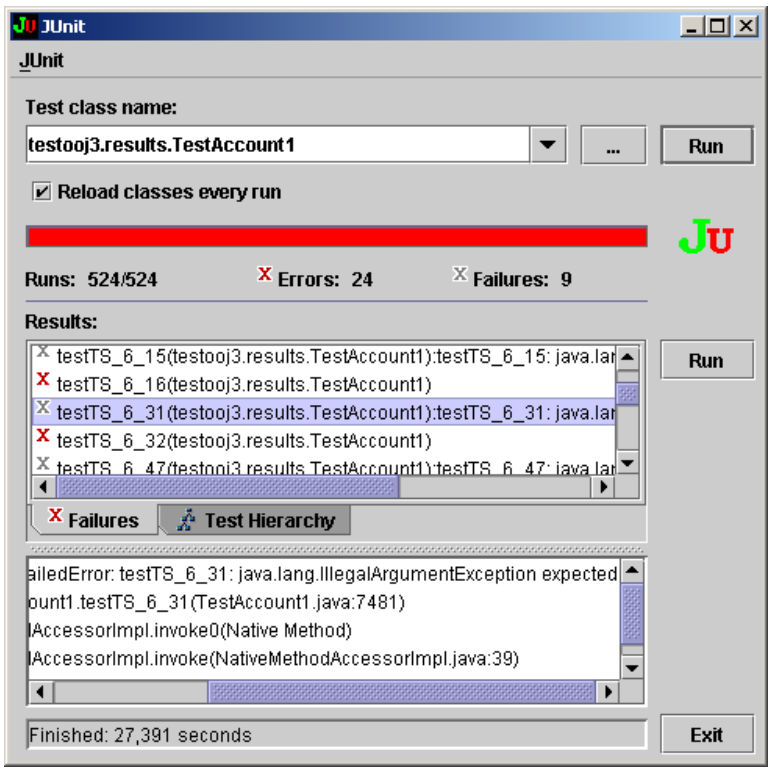


Figure 19. Results of executing in JUnit the generated testing file.

```
61 public void deposit( double amount, java.lang.String description )
62    throws java.lang.IllegalArgumentException, java.sql.SQLException
63 {
64    if (amount <= 0) {
65       throw new java.lang.IllegalArgumentException();
66    }
67    domain.AccountAnnotation a = new domain.AccountAnnotation(
68         this.number, amount++, description );
69    a.insert(this.number);
70 }
```

5.5. Dealing with non-primitive data types

Account has a *transfer(double amount, String description, Account targetAccount)* operation that deposits in *targetAccount* the *amount* passed as first parameter, withdrawing the same amount from the account executing the method plus a commission.

DOI: 10.1002/stvr

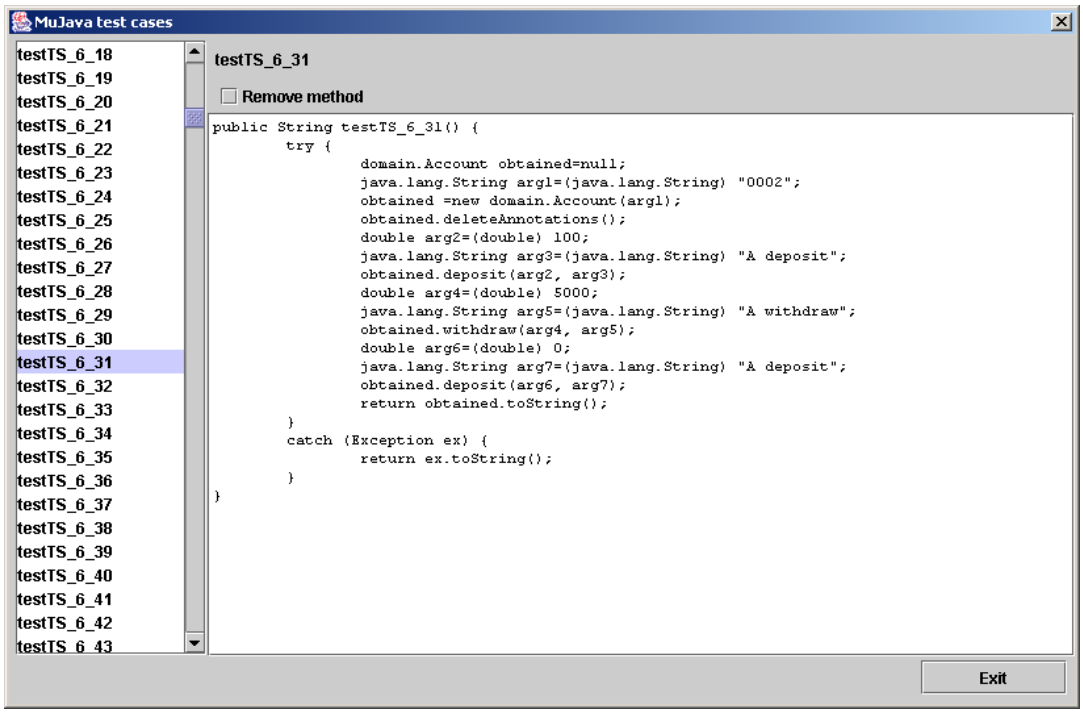


Figure 20. MuJava test cases.

Values for the first and second parameters can be given by hand by the user using the window in Figure 14; however, instances of *Account* must be taken from a repository of serialized objects which must have been previously built and saved.

The tool includes a small code generator that helps write a file to create and serialize instances of any class that, as in this example, can be required as parameters in test methods. Figure 22 shows the corresponding window: the class structure is loaded and shown in the tree on the left; the tool adds some code to the editor on the right-hand side for creating an instance of the required class. The tool user may replace the '//TODO' line by calls to class methods for putting the instance in the required state, such as is shown in the highlighted lines of Figure 22.

The objects created in this way are serialized and saved in the serialized objects folder given in the tool configuration. The contents of this folder can be inspected with the window of Figure 23: when the user selects a file with *.ser* extension, the tool recovers the corresponding instance and, by means of reflection, shows its set of fields and the public methods with no parameters (among these, there are the *get* methods that read the instance state). The user can select a field and see its value (if it is public), as well as the result of non-void, public methods with no parameters (since this is the usual signature of *observer* methods).

DOI: 10.1002/stvr

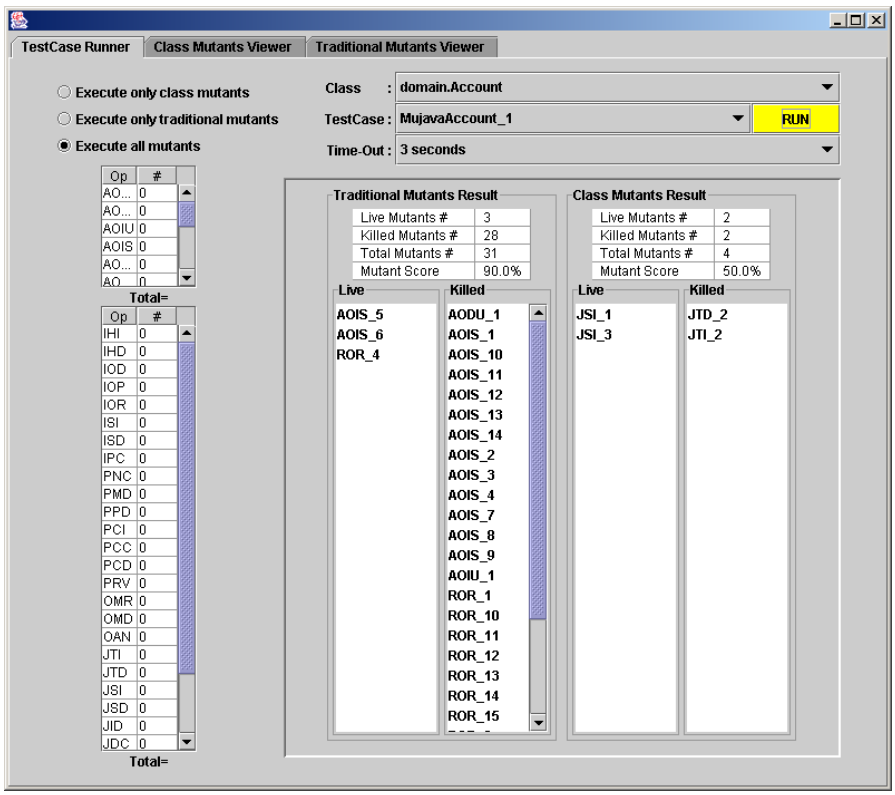


Figure 21. Results of the execution of one of the MuJava files.

Test methods that require serialized instances include code for recovering them from disk. Figure 24 shows a sample of one testing method that makes use of the aforementioned *transfer* method (note the highlighted lines): the third parameter of *transfer* is an account, that is read from a file. The tool uses as testing values those files in the folder of serialized values whose name starts with the name of the class of the required instance and whose extension is '.ser' (thus, it would use *domain.Account.ser*, but also *domain.Account100.ser*).

6. EXPERIMENT DESCRIPTION

The research question can be stated as:

Does the use of the automatic testing tool affect the efficiency of the testing process?

DOI: 10.1002/stvr

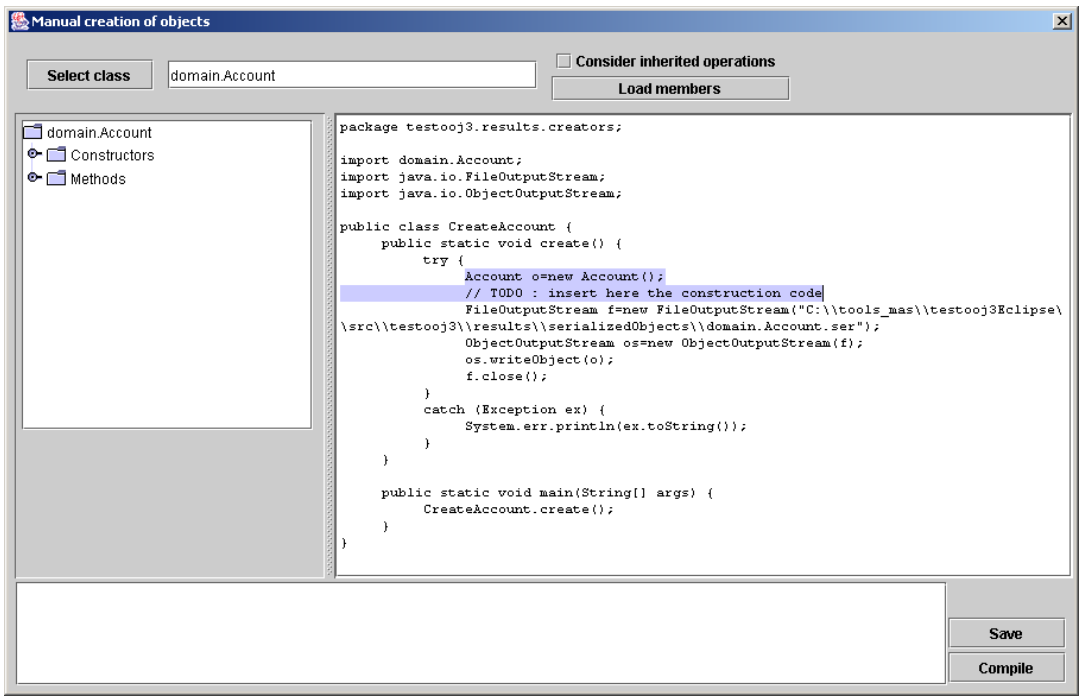


Figure 22. A small code generator to create and serialize instances.

The formulated experimental hypotheses are the following:

Null Hypothesis, H0. The efficiency of the testing process with and without the automatic tool is not significantly different.

Alternative Hypothesis, H1. The efficiency of the testing process with and without the automatic tool is significantly different.

The *efficiency of the testing process*, which is the dependent variable, is measured as the percentage of mutants killed in the given time. The independent variable is the use (or non-use) of the tool.

6.1. Subjects

Twenty eight subjects from the University of Castilla-La Mancha participated in this experiment. All of them were in their fifth year of Computer Science. Table III summarizes their ages and years of experience with Java. One of them had experience with JUnit during the previous year.

DOI: 10.1002/stvr

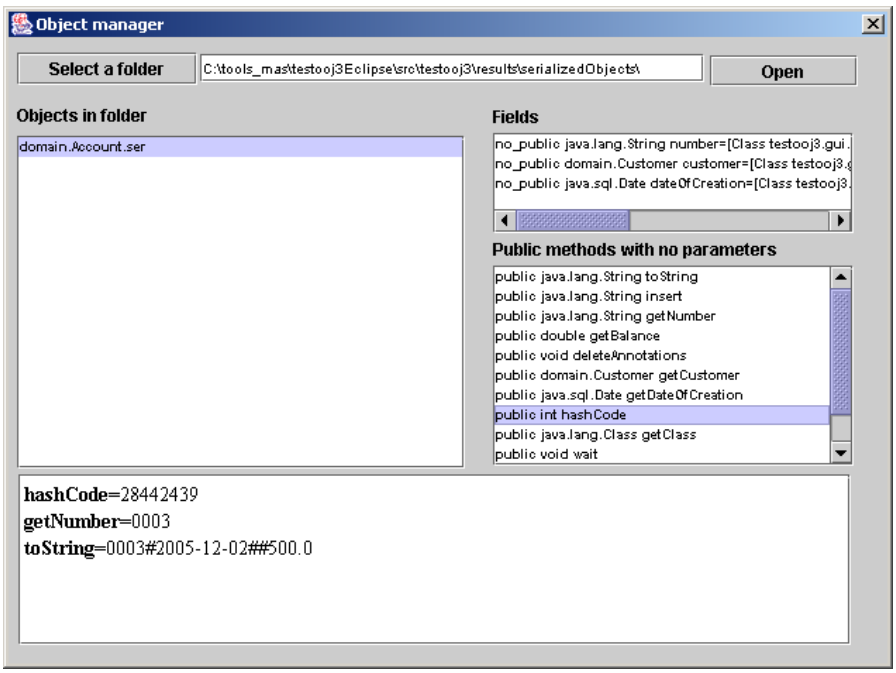


Figure 23. Inspection of serialized objects.

The tasks to be performed did not require high levels of industrial experience, so this experiment with students could be considered as appropriate [50,51]. Moreover, students are the next generation of people entering the profession, so they are close to the population under study [52]. In addition, working with students implies a set of advantages, such as the fact that the prior knowledge of the students is reasonably homogeneous.

Three and two weeks before the experiment, all of the subjects received two training sessions of two hours, in which the main issues of JUnit were commented on and where some examples of the tasks to be performed by them were explained by the conductor of the experiment. So, it can be considered that the level of experience they brought to the experiment was acceptable.

6.2. Experimental task and procedure

The subjects were randomly assigned to two groups of 14 people, and each group performed the experiment in a different laboratory. During their Computer Science studies, all of them had received

DOI: 10.1002/stvr

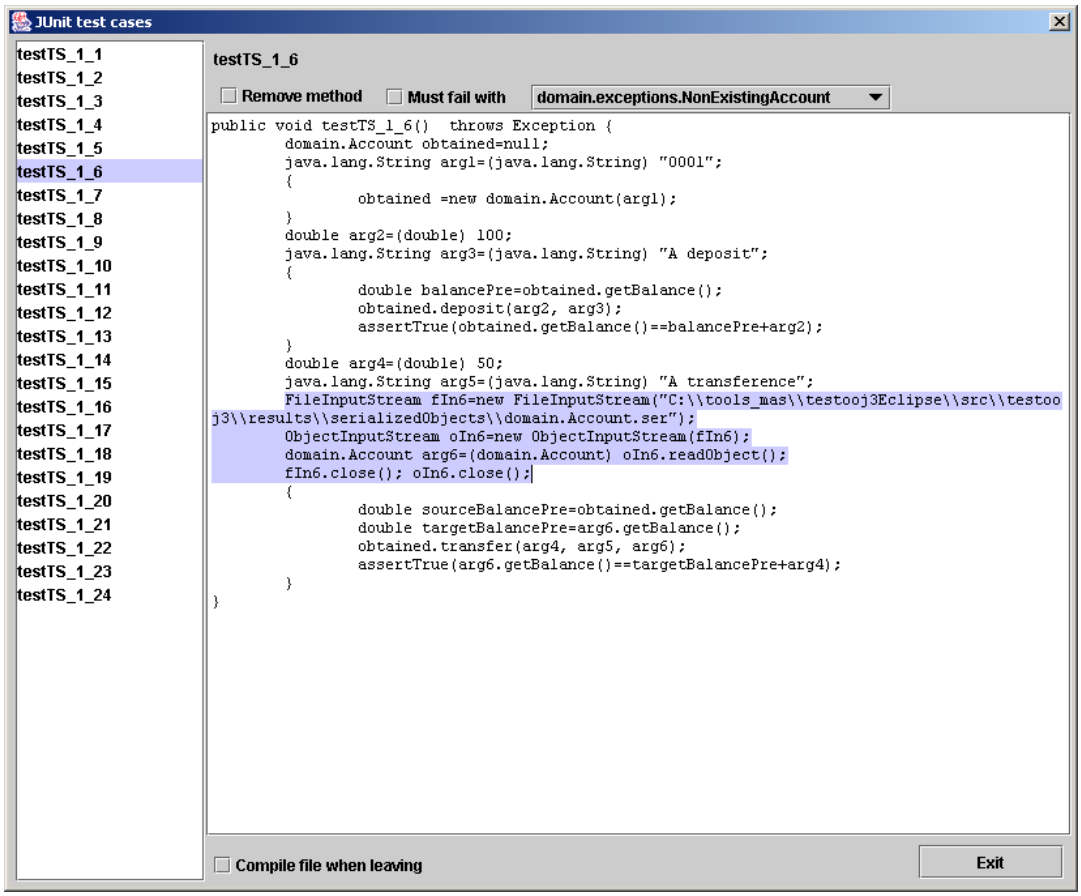


Figure 24. A test case that uses a serialized object.

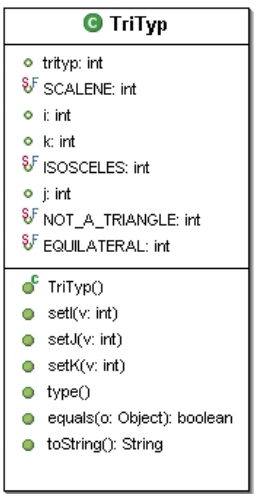
the same training in programming, design, testing, strategies for testing (boundary values, equivalence classes, etc.) and in the use of JUnit.

The Universe of Discourse (UoD) of the testing process was quite usual and not exceptional at all (the already discussed 'classical' triangle problem [43]), so there was no need for extra effort in understanding it. Both groups had to test a class implementing the problem: one group using the JUnit plug-in included in Eclipse (the manual group), whereas the other group had to use the tool presented in Section 5 (the tool group). In the manual group, the first 50 minutes of the experiment were dedicated to reviewing JUnit, and applying it to the testing of a class that implements the bubble sort algorithm. In the tool group, the first 50 minutes were dedicated to presenting the testing tool, and also applying it to test the class implementing the bubble sort algorithm. Thus, the students in both groups would

DOI: 10.1002/stvr

Table III. Main characteristics of the subjects.

		Subject													
Tool group	Years of experience	5	5	4	6	5	4	4	7	6	4	5	5	3	7
	Age	24	23	22	23	23	22	22	25	23	22	16	23	25	26
Manual group	Years of experience	6	5	5	5	5	1	7	1	5	5	6	6	6	6
	Age	23	23	23	28	23	27	26	35	22	23	25	24	24	25



The *TriTyp* class, whose structure is shown in the next figure, implements functionality to determine the type of a triangle.

For this, it includes the operations *setI(int), setJ(int)* and *setK(int)*, that give values to the three sides of the triangle (respectively, to the fields i, j, k). The operation *type()* determines the type of the triangle and assigns the *trityp* field one of the following values: EQUILATERAL, ISOSCELES, SCALENE or NOT_A_TRIANGLE.

A triangle is equilateral when its three sides are equal, isosceles when two are equal and one different, and scalene when the three sides are different. A triangle is not valid when any of the sides is zero or less than zero, or when the sum of two sides is less than the third.

Using JUnit, you must build a class *TestTriTyp* to test the behaviour of the *TriTyp* class, i.e. that the type of the triangle is correctly determined. You must use the operations *TriTyp()*, *setI(int)*, *setJ(int)*, *setK(int)* and *type()*.

When you have finished, the conductor of the experiment will collect the testing files you have produced.

Figure 25. Statement of the experiment.

start the testing of the triangle classification situation during the remaining 50 minutes with the same fatigue level.

Each subject received a correct implementation of the *TriTyp* class and the statement shown in Figure 25. As the figure states, the conductors of the experiment collected at the end all the testing files from the computers.

In order to increase the motivation and interest of the subjects, it was explained to the students of both groups that the exercises that they were going to perform could be similar to those that they would find in their examination at the end of the term, and that it was very important to understand testing. Furthermore, 0.5 points would be added to their marks if they participated in the experiment.

Once the conductors of the experiment had collected the JUnit test cases, they translated them into MuJava test cases according to the equivalence rules discussed in Section 5.4.

DOI: 10.1002/stvr

Table IV. Summary of results.

	Tool group			Manual group		
	Number of test cases	Killed mutants (%)		Number of test cases	Killed mutants (%)	
		Traditional	Class		Traditional	Class
	48	90	100	16	93	100
	6280	100	100	6	77	100
	14 925	100	100	24	82	100
	100	95	100	4	70	100
	91	97	100	14	83	100
	48	79	100	11	97	100
	252	96	100	22	93	100
	216	100	100	14	84	100
	150	98	100	15	96	100
	60	98	100	20	90	100
	100	98	100	9	76	100
	80	98	100	48	90	100
	36	96	100	14	83	100
	32	98	100	15	69	100
Min	32	79		4	69	
Max	14 925	100		48	97	
Sum	22 418	1343		232	1183	
Mean	1601.29	95.93		16.57	84.5	
Standard deviation	4173.86	5.51		9.12	9.12	

6.3. Discussion of results

All the data analysis presented in this section was carried out by means of the SPSS package. Table IV shows the results collected in both groups (number of test cases written by each subject and percentages of killed mutants, distinguishing traditional and class mutants). In the class mutants, there is not any difference between the tool and manual groups, which might suggest that these operators are less efficient than the traditional operators. Thus, this study will only be concerned with the traditional mutants. This table also shows the descriptive statistics of both groups. At first glance, data from the tool group are better than those from the manual group.

Since the data do not follow a normal distribution, a non-parametric test (Mann–Whitney) was carried out for corroborating the hypotheses. The results obtained allow one to reject the Null Hypothesis (p-value $= 0.000$), so concluding that the use of the tool does have an influence on the test efficiency.

6.4. Threats to validity

This section discusses several issues that could threaten the validity of the experiment and how they have been alleviated.

DOI: 10.1002/stvr

6.4.1. *Threats to conclusion validity*

The conclusion validity is the 'degree to which conclusions can be drawn about the existence of a statistical relationship between treatments and outcomes' [53]. In the experiment, due to participation on a voluntary basis and because of the small population, it was not possible to plan the selection of a population sample by using any of the common sampling techniques. Thus, it was decided to take the whole population of the available classes as target samples. A limited number of data values were collected during the execution of the experiment, due to the limited time duration and the number of subjects.

With regards to the quality of data collecting, 'pencil and paper' were used: hence, data collection could be considered critical. Time was the same for both groups (50 minutes).

Finally, the quantity and the quality of the data collected and the data analysis were enough to support the conclusion, as described in Section 6.3, concerning the existence of a statistical relationship between the independent and dependent variables.

6.4.2. *Threats to construct validity*

The construct validity is the 'degree to which the independent and the dependent variables are accurately measured by the measurement instruments used in the experiment' [53].

The dependent variable, the 'efficiency of the testing process', was measured as the percentage of mutants killed in the given time, and since this is an objective measure, it can be considered as possessing construct validity.

6.4.3. *Threats to internal validity*

The internal validity is the degree of confidence in a cause–effect relationship between factors of interest and the observed results.

The analysis performed here is based on correlations. Of course, such a statistical relationship does not demonstrate *per se* a causal relationship. It only provides empirical evidence of it. On the other hand, it is difficult to imagine what alternative explanations could exist for the results besides a relationship between the use of the tool and the coverage reached by tests.

Finally, the following issues have also been taken into account: differences among subjects, knowledge of the universe of discourse, precision in the time values, learning and fatigue effects, persistence effects, subjects' motivation and plagiarism influence between students.

6.4.4. *Threats to external validity*

The external validity is the 'degree to which the research results can be generalized to the population under study and other research settings' [53]. The greater the external validity is, the more the results of an empirical study can be generalized to actual software engineering practice.

Two main threats to validity have been identified in this experiment that limit the ability to apply any generalization and are outlined below.

DOI: 10.1002/stvr

- *Materials and tasks used*. The experiment has used a 'toy example', but one which is commonly used as a benchmark in testing experiments. However, more empirical studies taking 'real cases' from software companies must be done in the future. However, the authors think that if the automatic testing tool has demonstrated its utility in the small example of the triangle classification situation, it will be even more useful in real, complex cases where automation is more necessary.
- *Subjects*. To solve the difficulty of obtaining professional subjects, students from software engineering courses were used, although more experiments with practitioners must be carried out in order to be able to generalize these results. However, in this case, the tasks to be performed do not require high levels of industrial experience, so experiments with students would seem to be appropriate [50].

7. CONCLUSIONS

This article has presented a testing process whose steps can be almost completely automated using existing tools (JUnit and MuJava for Java, NUnit for .NET) and two new tools to generate compatible test cases.

Both the tools and the process are based on techniques that individually have been in existence for a number of years, but that, to the best knowledge of the authors, have not been put to work together previously. The generation of test cases is based on writing regular expressions, whose alphabet proceeds from the set of public operations of the CUT. The regular expression, combined with the assignment of test values (which, for complex values, are processed using serialization), makes the automatic generation of a huge number of test cases possible, reaching a degree of automation much greater than with other existing tools.

The suitability of the tool for generating high-quality test cases was checked in a controlled experiment with two groups of students with the same academic level, some of whom used the Java tool, while others wrote test cases manually. The students that used the tool obtained much better results than the students who did not. However, more experiments with practitioners and more complex systems must be carried out in order to be able to validate this approach.

ACKNOWLEDGEMENTS

This work was partially supported by the MÁS project, TIC2003-02737-C02-02 (Ministerio de Educación y Ciencia/FEDER).

REFERENCES

1. Runeson P, Andersson C, Höst M. Test processes in software product evolution: A qualitative survey on the state of practice. *Journal of Software Maintenance and Evolution: Research and Practice* 2003; **15**(1):41–59.
2. Rice RW. Surviving the top 10 challenges of software test automation. *CrossTalk: The Journal of Defense Software Engineering* 2002; **15**(5):26–29.
3. Geras AM, Smith MR, Miller J. A survey of software testing practices in Alberta. *Canadian Journal of Electrical and Computer Engineering* 2004; **29**(3):183–191.

DOI: 10.1002/stvr

4. Giraudo G, Tonella P. Designing and conducting an empirical study on test management automation. *Empirical Software Engineering* 2003; **8**(1):59–81.
5. Meudec C. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability* 2001; **11**(2):81–96.
6. Ng SP, Murnane T, Reed K, Grant D, Chen TY. A preliminary survey on software testing practices in Australia. *Proceedings of the Australian Software Engineering Conference (ASWEC 2004)*, Melbourne, Australia, April 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 116–125.
7. Ball T, Hoffman D, Ruskey F, Webber R, White L. State generation and automated class testing. *Software Testing, Verification and Reliability* 2000; **10**(3):149–170.
8. Hoffman D, Strooper P, Wilkin S. Tool support for executable documentation of Java class hierarchies. *Software Testing, Verification and Reliability* 2005; **15**(4):235–256.
9. Erdogmus H, Morisio M, Torchiano M. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering* 2005; **31**(3):226–237.
10. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 2003; **13**(1):25–53.
11. Kirani SH, Tsai W-T. Method sequence specification and verification of classes. *Journal of Object-Oriented Programming* 1994; **7**(6):28–38.
12. Grieskamp W, Gurevich Y, Schulte W, Veanes M. Testing with abstract state machines. *Formal Methods and Tools for Computer Science: Proceedings of the Workshop on Abstract State Machines (ASM 2001)*, Las Palmas de Gran Canaria, Canary Islands, February 2001; 257–261.
13. Hong HS, Lee I, Sokolsky O, Cha SD. Automatic test generation from statecharts using model checking. *Proceedings of the First International Workshop on Formal Approaches to Testing of Software (FATES 2001)*, Aalborg, Denmark, August 2001. BRICS: Basic Research in Computer Science, 2001; 15–30.
14. Burton S, Clark J, McDermid J. Automatic generation of tests from statechart specifications. *Proceedings of the First International Workshop on Formal Approaches to Testing of Software (FATES 2001)*, Aalborg, Denmark, August 2001. BRICS: Basic Research in Computer Science, 2001; 31–46.
15. Beck K, Gamma E. Test infected: Programmers love writing tests. *Java Report* 1998; **3**(7):51–56.
16. Beck K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley: Boston, MA, 2000.
17. Parasoft. Automating and Improving Java Unit Testing: Using JTest with JUnit. http://www.parasoft.com [18 April 2006].
18. SilverMark Inc. *SilverMark's Enhanced JUnit Version 3.7*. SilverMark Inc., 2002.
19. Oriat C. Jartege: A tool for random generation of unit tests for Java classes. *Proceedings of the 2nd International Workshop on Software Quality*, Erfurt, Germany, September 2005 (*Lecture Notes in Computer Science*, vol. 3712). Springer: Berlin, 2005; 242–256.
20. Leavens GT. The Java Modeling Language (JML) home page. http://www.cs.iastate.edu/∼leavens/JML [2 December 2005].
21. Cornett S. Code Coverage Analysis. http://www.bullseye.com/webCoverage.html [15 June 2005].
22. DeMillo RA, Offutt AJ. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(2):109–127.
23. Daniels FJ, Tai KC. Measuring the effectiveness of method test sequences derived from sequencing constraints. *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 1999)*, Santa Barbara, CA, August 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 74–83.
24. Kim S-W, Clark JA, McDermid JA. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability* 2001; **11**(4):207–225.
25. Vincenzi AMR, Maldonado JC, Barbosa EF, Delamaro ME. Unit and integration testing strategies for C programs using mutation. *Software Testing, Verification and Reliability* 2001; **11**(4):249–268.
26. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, MO, May 2005. ACM Press: New York, 2005; 402–411.
27. Juristo N, Moreno AM, Vegas S. A survey on testing technique empirical studies: How limited is our knowledge? *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2002)*, Nara, Japan, October 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 161–172.
28. Choi B, Mathur AP, Pattison B. PMothra: Scheduling mutants for execution on a hypercube. *Proceedings of the ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, Key West, FL, December 1989. ACM Press: New York, 1989; 58–65.
29. Duncan IMM. Strong mutation testing strategies. *PhD Thesis*, Department of Computer Science, University of Durham, U.K., 1993.
30. Mresa ES, Bottaci L. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability* 1999; **9**(4):205–232.

DOI: 10.1002/stvr

31. Weiss SN, Fleyshgakker VN. Improved serial algorithms for mutation analysis. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1993)*, Cambridge, MA, June 1993. ACM Press: New York, 1993; 149–158.
32. Baudry B, Fleurey F, Jézéquel J-M, Le Traon Y. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability* 2005; **15**(2):73–96.
33. Offutt AJ, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(2):99–118.
34. Mathur AP. Performance, effectiveness, and reliability issues in software testing. *Proceedings of the 15th Annual International Computer Software and Applications Conference (COMPSAC 1991)*, Tokyo, Japan, September 1991. IEEE Computer Society Press: Los Alamitos, CA, 1991; 604–605.
35. Wong WE, Mathur AP. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software* 1995; **31**(3):185–196.
36. Ma Y-S, Offutt J, Kwon YR. MuJava: An automated class mutation system. *Software Testing, Verification and Reliability* 2005; **15**(2):97–133.
37. Krauser EW, Mathur AP, Rego VJ. High performance software testing on SIMD machines. *IEEE Transactions on Software Engineering* 1991; **17**(5):403–423.
38. Howden WE. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* 1982; **8**(4):371–379.
39. Offutt AJ, Lee SD. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering* 1994; **20**(5):337–344.
40. Hirayama M, Yamamoto T, Okayasu J, Mizuno O, Kikuno T. Elimination of crucial faults by a new selective testing method. *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2002)*, Nara, Japan, October 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 183–191.
41. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability* 1997; **7**(3):165–192.
42. NUnit. NUnit, testing resources for Extreme Programming. http://www.junit.org [5 January 2003].
43. Myers GJ. *The Art of Software Testing*. Wiley: New York, 1979.
44. Offutt J. A practical system for mutation testing: Help for the common programmer. *Proceedings of the 12th International Conference on Testing Computer Software*, Washington, DC, June 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 99–109.
45. Chow TS. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 1978; **4**(3):178–187.
46. Boyapati C, Khurshid S, Marinov D. Korat: Automated testing based on Java predicates. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, Rome, Italy, July 2002. ACM Press: New York, 2002; 123–133.
47. Tse T, Xu Z. Test case generation for class-level object-oriented testing. *Proceedings of the 9th International Software Quality Week*, San Francisco, CA, May 1996. Software Research Inc.: San Francisco, CA, 1996; 4T4.0–4T4.12.
48. Korel B, Al-Yami AM. Assertion-oriented automated test data generation. *Proceedings of the 18th International Conference on Software Engineering (ICSE 1996)*, Berlin, Germany, March 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 71–80.
49. Tracey N, Clark J, Mander K, McDermid J. Automated test data generation for exception conditions. *Software—Practice and Experience* 2000; **30**(1):61–79.
50. Basili VR, Shull F, Lanubile F. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 1999; **25**(4):456–473.
51. Höst M, Regnell B, Wohlin C. Using students as subjects: A comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering* 2000; **5**(3):201–204.
52. Kitchenham B, Pfleeger SL, Pickard L, Jones P, Hoaglin D, El-Emam K, Rosenberg J. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 2002; **28**(8):721–734.
53. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic: Norwell, MA, 2000.

DOI: 10.1002/stvr