How to compare and exploit different techniques for unit-test generation

Alberto Bacchelli
bacchelli@cs.unibo.it

Paolo Ciancarini
ciancarini@cs.unibo.it

Davide Rossi
rossi@cs.unibo.it

Department of Computer Science
University of Bologna, Italy

Abstract

The size and complexity of software is continuously growing, and testing is one of the most important strategies for improving software reliability, quality, and design. Unit testing, in particular, forms the foundation of the testing process and it is effectively supported by automated testing frameworks. Manual unit-test creation is difficult, monotonous and time-consuming. In order to reduce the effort spent on this task, several tools have been developed. Many of them can almost automatically produce unit tests for regression avoidance or failure detection.

This paper presents a practical comparison methodology to analyze different unit-testing creation tools and techniques. It validates the effectiveness of tools and spots their weaknesses and strengths. The validity of this methodology is confirmed through a real case experiment, in which both the manual implementation and different automatic test generation tools (based on random testing) are used. In addition, in order to integrate and exploit the benefits of each technique, which result from the comparison process, a testing procedure based on "best practices" is developed.

Keywords: testing; comparison methodology; failure detection; regression testing; automatic test generation tools

1 Introduction

Testing is an important and widely-accepted activity to verify a software at runtime. It can be performed at three different stages with increasing granularity: single class (or module) testing, classes (or modules) group testing, whole system testing [35]; which are usually referred to as: unit testing, integration testing and system testing. Each stage has its own difficulties and strategies, and different techniques are available. In this paper, we decided to concentrate our efforts on unit testing, which constitutes the foundation for other testing levels. In particular, we consider object-oriented software and the Java programming language, but our results could be easily transposed to be useful for different programming languages and paradigms.

The *x*Unit testing [2] framework was created in order to improve object-oriented programmer productivity in developing and running unit-test cases. Through it, it is possible to easily write unit tests that exercise and make assertions about the code under test. Each test is independent of each other and it is usually written in the same language as the code they test. The *x*Unit framework is intentionally simple to learn and to use, and this was a key design criteria: the authors wanted to be sure that it "was so easy to learn and to use that programmers would actually use it" [28]. The developer, with a simple command, could automatically run all the *x*Unit tests he created and then, after the execution, receive the report generated by the framework with the number of successes and the details of any failures.

In this article, we consider JUnit [1], which is the Java flavor of the *x*Unit framework. Although it is the "de-facto" standard tool to automatically execute Java unit tests, there are other interesting alternatives that could be effectively used (e.g., [10]).

In industry, testing is used for quality assurance, because it provides a realistic feedback on system behavior, reliability and effectiveness. At the same time, testing -especially unit testing- can be tedious and difficult, and it often consumes a considerable amount of time [31]. For this reason, research is now highly active in producing testing techniques capable of automatically creating unit tests. The usage of these tools is not already equally standardized as JUnit usage.

In a previous article [7], we compared the effectiveness of unit-test cases which were automatically created with manually written ones. We used various automatic test creation tools, based on random testing, and we proved that they can produce trustworthy and useful test cases, and can improve and speed-up testing engineers' tasks. We also showed the advantages which result from the manual creation of unit tests, that cannot yet be achieved by automatic tools. Finally, we noted that the success rate of this kind of testing depended to a great degree on proper planning and proper use of these testing tools. Thus, we briefly outlined some "best practices" for integrating the manual and automated test processes, in order to effectively produce unit-test cases and obtain benefits from all the techniques.

As scientific research is highly active and motivated in this particular field, the tools we examined and compared in [7] have since been further enhanced [33], studied in more detail [32] or replaced with more effective tools [15, 39]. Also the only commercial software we included [5], which was highly experimental, was replaced with a more stable version [4] in order to address the company's business.

These automatic unit-test generation tools evolve extremely quickly, and thus we consider it significant to outline a comparison methodology to help the test engineer to analyze the capabilities of different tools, which can thus be exploited and fit into the correct testing "best-pratices".

In order to effectively extract this methodology, we use the real-world case study we conducted in [7]. We show, in detail, the comparison method we used and its validity. Then, we see what its advantages and shortcomings are, and how it could be usefully extended. Finally, we outline how the results obtained could be used in "best practices" capable of exploiting and integrating the different techniques which were analyzed.

In Section 2 we introduce unit testing and we show its main objectives. Then, in Section 3, we present the experimental study showing the environment in which we conducted it, describing which unit testing techniques and tools we used, and discussing its validity. Later (Section 4), we point out the comparison we adopted to study this real world case and we explain how we analyzed the different facets of the results which we obtained through the different testing approaches. Consequently, we outline the abstract comparison methodology. Finally in Section 5, we propose the "best practices" to effectively exploit the different techniques used.

2 Unit testing

Ideally tests are implemented in the flow of the software development process, which means at increasing granularity. Only when we have some module/class implemented and their unit test cases ready, can we advance creating integration tests or new features. For example, it is possible to proceed in a cyclical manner by first writing unit tests for a few classes, then developing the necessary integration tests for those classes, then restarting again with new classes and their unit tests, which leads to further integration testing, and restarting the cycle again until the system is completed. Within this process unit test cases fill the foundation of the whole testing system and other testing parts can rely on first checking.

The principal aims of unit testing are the same as functional testing: verify the behavior of the tested component and help in finding implementation defects. The difference consists in their target: unit testing is mainly focused on small source code parts which are separately testable. Unit test cases input small functions or methods with specific values and check the output against the expected results, verifying the correct code behavior.

2.1 Failure detection and regression avoidance

Unit testing is mainly used to achieve two objectives: searching for defects in new -or not previously tested- code and avoiding regression after the evolution of code already under test. In the first instance, through test cases, the test engineer tries to check if the code is correctly implemented and does what it is expected to do; in the second situation failing tests warn that a software functionality that was previously behaving correctly stopped working as intended.

For this reason, in literature and industry we find unit-test generation tools that address those two different issues: failure detection and regression avoidance. In the former case tools try to spot unwanted or unexpected behavior, which could lead to a detection of failure. In the latter case they generate "tests that capture, and help you preserve, the code's behavior" [4]. They will "pass", if the code behaves in the same way that it did at the moment of their creation, but will "fail" whenever a developer changes the code functionalities, highlighting unanticipated consequences.

As we focus essentially on tools which are based on random testing, the main difference between tools that create unit tests for failure detection purposes and tools that generate regression tests is how they deal with the *oracle problem*. "An oracle is any (human or mechanical) agent which decides whether a program behaved correctly in a given test, and accordingly produces a verdict of "pass" or "fail" [...] oracle automation can be very difficult and expensive" [3].

When automatic generation tools produce regression tests, the source code which is provided is supposed to be without defects. For this reason, those tools generally consider the system itself as a sort of oracle: everything it outputs in reply to an input is the expected and correct answer. Thus, they will create only not failing tests, and, in this way, they take a "snapshot" of the tested system state.

On the other hand, tools that generate unit tests that reveal bugs in code cannot consider the tested system as the oracle, because it could provide both right and wrong answers to inputs, as it is not supposed to be without defects. For this reason, these tools need another way to know if the received output is the expected one or not (i.e. to solve the oracle problem). The tools we examined in [7], use a different approach to tackle this issue. One of them [14] considers unexpected exceptions that are raised as evidence of a possible failure. It does not need the test engineer to provide any additional information before submitting the source code to the tool. Another software [33] requires more information about the tested class, such as class *invariants*. These properties will be used as the oracle: if the tool discovers an instance in which the submitted code does not respect the requested invariants, a failing unit test will be created. If the test engineers want to use this tool, they have to spend time preparing a more detailed input, which actually constitutes the oracle.

Finally, it is important to underline that failure detection tests and regression tests have different targets and for this reason they are mainly created in a different way. However, from the moment unit tests are implemented, they become part of the same unit test suite. A test that is used now to reveal a defect, can be incorporated later in the test suite and used as a regression test. A regression test that is not failing now, could be an effective failure detection test in the future or in a different implementation.

This already gives a hint of "best practices" which explain how to exploit and integrate both tests which are automatically generated by tools with different targets and tests which are manually implemented.

3 The experimental study

In [7] we wanted to show which approach was the best one when the test engineer had to decide whether to use tools which automatically generate unit tests and how. We chose to apply both manual and automatic techniques to a real case study, to have the chance of showing authentic data.

3.1 The Freenet Project

The Freenet Project [13] peer-to-peer software (henceforth Freenet) was chosen as subject for experiment, because it had all the features which we were interested in and which we considered relevant for our experiment validity.

Freenet is free software which allows the publishing and obtaining of information over the Internet, avoiding censorship. To achieve this, the network it creates is completely decentralized and publishers and consumers of information are anonymous. Communications between network nodes are encrypted and are routed through other peers to make communication tracking extremely difficult.

The software was subject to three drastic changes which caused important modifications of Freenet functionalities and its development team. For this reason, Freenet assumed, over time, many of the characteristics that allows us to equate it with legacy software [36].

Like with legacy software, many of the developers who created Freenet are not working on the project anymore, even though much of the source code they implemented is still in use and form an important part of the application. Then there is a crucial lack of documentation both for old source code and for new pieces of it; furthermore this is an issue for both high and low level documentation. For this reason, to fully understand Freenet functioning, or even only little parts of it, it is necessary either to read the corresponding source code or to interact with the developers community. As with many legacy software, the few active developers are constantly busy and they put most of their effort into developing new functionalities rather than revising the existing codebase.

Tests are so important that "the main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of comprehensive tests" [18]. Also in this sense, Freenet is a legacy software: when we started the experiment it had only 14 test cases for the whole source code, which included more than 800 classes.

The main difference between legacy software and Freenet lies in the fact that the latter is still under heavy development and evolution. This aspect, however, does not have any relevance to our study, so we have chosen Freenet to represent not only software without a pre-existing significant test suite, but also legacy software in general.

Positively, Freenet is open source code and was developed in Java, which is a language that is fully supported by the majority of automatic unit test generation tools. In this way, we had the possibility of comparing the most and latest relevant examples of industry and academic research in this field.

Furthermore, even though the software had many aspects related to legacy software, the developing structure was modern and efficient. It had effective mailing lists and an active IRC channel populated by the most important project developers. The code was maintained in a functional Subversion repository and we were granted full privileges to it. Finally the infrastructure to insert JUnit tests was already prepared and there was also support for continuous integration and test.

The changes we described earlier correspond to release 0.3, release 0.5 and release 0.7. During our experiment we used the last release which was the official one.

3.2 Development environment

The development environment for the experiment has been the GNU/Linux operating system provided with SUN Java 5 SE, Eclipse Europa IDE, Ant build tool version 1.7, JUnit version 3.8. During manual tests implementation hardware performance has not been an issue, whereas we got serious benefits in computational time using a dedicated workstation (dual Intel Xeon 2.8Ghz processor, with 2GB of Ram) when generating tests with the automatic tools and calculating the value of some of the metrics.

The developer who implemented unit tests manually was a graduate student with a reasonable (about three years) industrial and academic experience in Java programming. He also had a basic knowledge of *x*Unit, previously developed using SUnit (the SmallTalk *x*Unit dialect) [2].

3.3 Procedure

In order to gain a better knowledge of the Freenet source code, environment and community, we began the experiment dedicating three full-time working months (which corresponded to almost 500 hours) to the manual implementation of unit tests. Then we started studying automatic test generation tools and we created tests using them. At the end we performed the comparison between each test suite that was generated using different techniques.

The decision to start with the manual implementation of tests made it easier to decide which and how many classes to put in our study domain. We decided to concentrate our efforts on support "leaf" classes (i.e. without outgoing references), because they offer two important advantages: they do not have significant references to other classes in the project -which means they are easier and faster to study and comprehend- and they are support classes, very similar to library classes, so they are often referenced through the whole code, and this allowed us to see the testing effects spread as broadly as possible across the system.

At the end of the creation of manual unit tests, we had two different "snapshots" of the Freenet system, that we used as a basis for further testing. First we had the source code which was present at the beginning (i.e. without unit tests), in which we found certain bugs through manual testing; and we had the source code after manual testing, which was modified to remove bugs and included the regression tests which we had manually implemented.

In order to successfully compare manual unit test implementation and automatic unit test generation tools, we used these two source code snapshots in a different manner. The first "snapshot" (i.e. without any manual modification) was useful for comparing failure detection capabilities. Using the source code as it was before the manual work, we were able to compare the bugs which had been found throughout manual implementation with the bug set that was detected by automatic tools. This comparison was possible only by working on the same source code basis, and allowed us to effectively spot similarities, advantages or faults of different the techniques.

On the other hand, we used the second "snapshot" to compare regression tests, manually and automatically generated. We already had manually implemented regression tests for this snapshot (which result from the manual testing), so it was more appropriate to use tools to generate regression tests over the same code, to see if their regression avoidance capability was less effective, similar or even better than the manual implementation.

It is still possible to checkout the source code we used as a basis for our tests, using the official Freenet Subversion repository.

3.4 Manual unit test implementation

To manually create unit tests, we first had to understand what was expected from the code we wanted to test. A reasonable approach to achieve this understanding is to read the documentation that was accompanying the code. Unfortunately, as already mentioned, very often it was not present or very poor and outdated. For this reason, we usually had to directly study the source code under test. It is not a good practice to understand what the code is supposed to do by reading it, because if the code contains defects or the developer had a wrong understanding of the requirement, studying only the code does not help in finding the problem. In addition, often we were unable to understand the functionalities supported by the analyzed code. So we had to search external documentation to clarify well-known problems that were addressed by the code (e.g., the DSA algorithm implementation), or -even worse- we had to contact the Freenet developers for further explanations. For this reason, the lack of documentation we encountered created important problems in the speed of manual testing, which we could not correctly estimate at the beginning of our work.

When we finally achieved full and correct knowledge of the classes that we wanted to test, we started writing test cases. Even though there is some research addressing the issue [41], there is not any widely accepted method of conducting the manual implementation of unit test cases. For this reason we decided to follow well-known and broadly used non-formal testing principles [24, 36].

First of all, when implementing tests, we tried to adopt a different point of view from the one which was used by the original code developers. We created a series of test cases that were intended to "break" rather than confirm the software under test. We also deeply analyzed all data structures that were created or used in each tested class. We created test cases to verify that data did not lose its integrity when using classes' methods or algorithms.

We also performed boundary-value analysis, because it has been statistically proven to be capable of detecting the highest number of defects [24]. We checked the code using the highest and lowest possible values, and we also tried to use values slightly outside the boundaries. Confirming the statistics, we found relevant bugs using this technique.

Our tests also performed control-flow path execution, which consists of trying to execute the code in every possible path to check the behavior of functions when dealing with inputs from different subsets. We usually created one test case for each possible code branch, to improve tests readability and facilitate future source code debugging.

We also did exception handling checking to verify if source code correctly deals with wrong inputs and methods usage.

Finally we created mock objects [42] to replace the objects used by the methods under test. This offered a layer of isolation and allowed us to check the code when dealing with specific results from external objects which we created the mock object for.

The failures we gradually found during the manual creation of unit tests formed a basis of knowledge that was useful when trying to find bugs in new tested classes.

At the end of this manual activities, we had defined the Freenet classes that formed the domain of our experiment. As mentioned earlier, their number was below our and Freenet developers' expectations. We supposed that it was possible to test at least all the classes in the freenet.support package, however the documentation problems considerably slowed the manual testing process.

As mentioned before (Section 3.3), after the manual testing process we obtained two different "snapshots" of the source code. The former, which was the source code basis without any testing, is used to compare the bugs that were manually found with the ones found through automatic tools, in order to see if they are complementary or overlapping. The latter, which was the source code after the manual testing phase (that is after the removal of defects and the creation of the manual unit-test suite), is used to compare the regression avoidance quality of the different techniques.

3.5 Automatic generation tools selection

We based automatic unit test generation tools selection on two criteria. First of all, we wanted to analyze the results of automatic tools both when dealing with regression avoidance and failure detection, because we supposed that their effectiveness would be different. Then we chose tools that were simple for developers to learn or use, as they would be more easily adopted in an industrial context.

For these reasons, we focused our attention on JUnit Factory [5] which addresses regression tests creation, JCrasher [14] which deals with failure detection and Randoop [31] that can be used for both problems.

JCrasher and Randoop, and presumably also JUnit Factory, are based on random testing, which consists of providing well-formed but random data as arguments for methods or functions and checking whether the results obtained are correct. This random approach has a few advantages [14]: it requires low or no user interaction (except when an error is found), and it is cost-effective. However, one of the major flaw of random testing is that very low probability sub-domains are likely to be disregarded by it [25]. To deal with this issue, the chosen tools are designed to easily cover shallow boundary cases, "like integer functions failing on negative numbers or zero, data structure code failing on empty collections, etc." [14]. Finally, we decided not to use tools that rely on formal specification of the system under testing (e.g., model-based testing [34, 43]), because such systems are not common in practice [9], especially for legacy systems. In addition, in our real case study even an external model (based on requirements, documentation or other sources) was not available.

3.6 JUnit Factory

JUnit Factory was developed at AgitarLabs, the research division of Agitar Software, and it was freely usable upon registration.

JUnit Factory was different from the other tools we used because it was proprietary and was only offered as a service. This means that the user can neither see the software source code nor download and use the binaries on his computer.

To use the JUnit Factory service, we had to download a specific Eclipse [17] plugin (the web interface version was not suitable for a large project, as it was only useful for quickly seeing JUnit Factory results on very small independent classes). This plugin allowed us to choose the classes which we wanted to automatically generate unit tests for. When the selection was completed, the whole Eclipse project was uploaded to Agitar server and put in a queue. Some minutes after (the waiting time was influenced by tested class characteristics, such as lines of code and complexity) the resulting unit tests were ready and downloaded into our Eclipse project and became part of it. JUnit Factory creates unit-test cases with a clear variable and method naming, in order to increase their readability . However, those tests still remain terse, as depicted in Listing 1.

```
public void
testBytesToBitsThrowsNullPointerException1()
   throws Throwable {
 byte[] b = new byte[2];
 b[0] = (byte)75;
 try {
  HexUtil.bytesToBits(b, null, 100);
  fail(''Expected
   NullPointerException to be thrown'');
  } catch (NullPointerException ex) {
   assertNull(
    "ex.getMessage()",ex.getMessage());
   assertThrownBy(
    HexUtil.class, ex);
  }
}
```

Listing 1. JUnit Factory-generated test case

From that moment, JUnit Factory tests could be easily run using the Agitar's executor. Even though they were very similar to JUnit tests, they could not be executed using the standard JUnit framework, because they made use of Agitar proprietary libraries.

This usage description shows two major shortcomings of this tool: the user, through a non-secure communication channel, must upload the code to remote server, where it could be stored and used for unspecified purposes; and the service does not generate fully compatible JUnit tests, and for this reason proprietary libraries and a binary program must be included in the tested software in order to run them.

In our experiment we used an open source software, so we had no security issues, but when dealing with the automatic creation of unit tests for a proprietary software, these are problems that could mean that it is may impede the use of JUnit Factory.

Finally, due to its proprietary nature, it was not possible to study JUnit Factory internal functioning, but we had to rely on Agitar public articles [11]: which stated that it was based on the same research which formed the basis for Randoop (see Section 3.8).

3.7 JCrasher

JCrasher generates tests which target failures in the tested code. It produces type-correct inputs in a random fashion and attempts to detect bugs by "causing the program under test to crash, that is, to throw an undeclared runtime exception" [14]. It is almost completely automatic: no human supervision is required except for the inspection of the test cases that have caused an error.

JCrasher is based on C. Csallner and Y. Smaragdakis academic research and its source code is completely available under the MIT license. The binary executable and its results could be used freely by a test engineer without any restriction.

The usage of this tool is straightforward, even though it requires to be familiar with Apache Ant [6], the popular Java-based software build system. The user must prepare an Ant makefile with a specific target to invoke JCrasher [14] and a simple text file containing the list of classes for which it must create failure detection tests. Then the tool will create JUnit test cases, in a directory specified in a property of the Ant makefile. Even though the JCrasher target is failure detection only, it produces both passing and failing unit tests.

The test creation is a heavy task from a computational point of view, and this caused memory issues (i.e. java.OutOfMemory exception raising) leading us to move our development to the dedicated workstation we described previously. In addition, resulting JUnit test cases are extremely terse (e.g., Listing 2) and they are generated in the same directory as the tested source code.

```
public void test121() throws Throwable {
 try{
  java.lang.String s1 = "";
  java.lang.String s2 =
   "\"\n\\.‘’@#$%^&/({<[|\\n:.,;";
  HTMLNode h3 =
   new HTMLNode("", "", "");
  h3.addAttribute(s1, s2);
 } catch (Throwable throwable) {
  throwIf(throwable);}
}
```

Listing 2. JCrasher-generated test case

For these reasons we decided to produce tests for one class at a time and this also allowed us to separate tests in different meaningful subdirectories.

The number of test cases which were created varied from class to class, but it was usually extremely high. For example it outputs 100.000 test cases for a single class, many of which were almost identical. The execution time for such a vast quantity of tests was not acceptable even for a very fast system (it took about twelve minutes to check only one class). For this reason we reduced the number of tests before integrating them into our Eclipse project, deleting tests that used inputs from the same subset.

Finally we decided to keep the JCrasher tests which past in order to verify whether passing tests could be effectively used as regression tests.

3.8 Randoop

Randoop (*Rand*om *T*ester for *O*bject-*O*riented *P*rograms) is the practical result of research by Dr. M.D. Ernst and C. Pacheco and, like JCrasher, it is distributed under the MIT License. Randoop can generate unit tests to do both regression avoidance and failure detection.

In the first case the generation requires only a very low human interaction: in order to obtain the regression test suite, it is sufficient to specify which classes have to be tested, and their *helper classes*. Helper classes are needed because the tool will only generate tests using the specified classes. For example, in order to effectively test Collections, it is necessary to input a class (e.g., java.utils.TreeSet) that allows the tool to instantiate concrete collections. This class is defined as *helper class*.

Randoop then accepts a time limit which is an upper bound for the time it uses for its computations. We found that ten seconds was long enough to generate meaningful regression tests. We noted that longer time limits did not result in enhanced tests, but only lead to a higher number of tests, which are more expensive when carried out. Finally, as Randoop is based on random input generation, its developer also offers the possibility of feeding Randoop with different random seed in order to obtain different random inputs. We note, however, that this possibility did not significantly change the produced tests, even though we tried a lot of different seeds. It is an issue that was also later noted by Randoop creator in [32], when he was trying to use it in another real world case. Randoop produces succinct test cases, where could even be difficult to spot the tested class (e.g., Listing 3).

```
public void testclasses15()
 throws Throwable {
  boolean v0 = false;
  freenet.support.SimpleFieldSet v1 =
    new SimpleFieldSet((boolean)v0);
  String v2 = "";
  double v3 = (double)1.0;
  double v4 =
   v1.getDouble((String)v2,(double)v3);
  String v5 = "hi!";
  long v6 = (long)100L;
  v1.put((String)v5,(long)v6);
  assertEquals(
   (double)1.0,
   (double)(Double)v4);
}
```

Listing 3. Randoop-generated regression test

On the other hand, Randoop failure detection functionality requires more time to prepare its input, because the user must write *contracts*. A contract is the way in which the programmer defines which are the properties that should remain the same in the class under examination. In practice, a contract is the implementation of a Java interface in which the developer makes some assertions about the tested object state. These assertions have to be always true. If during class checking, Randoop finds a sequence of method calls and inputs that make a check fail, then the test case which caused the failure is displayed.

We chose Randoop because we believe that a tool is more useful for developers, if they can express themselves using constructs they already know. In fact with this tool, they can write contracts in Java language. Other solutions [29], on the contrary, require the learning of a new formal language to express tested class properties and invariants. Randoop authors admit that this approach could be less expressive than using a specific formalism [31], but in our experience this did not turn out as a limiting factor.

Randoop performances were increased using the dedicated workstation, but its impact was less evident than when creating tests using JCrasher. Randoop did not suffer the same memory problem as JCrasher and it was reasonable to use it on the computer which we also used for standard programming. This implies that a single developer could benefit from using it, without the need of a powerful hardware.

As with JCrasher, we keep the tests that found failures to use them as regression tests. However, there were only a few of them and their influence was irrelevant.

3.9 Validity

The procedure we used to conduct the experiment led to relevant results [7], however since we are trying to outline an abstract method for unit tests comparison, we would like to suggest some enhancement that could be adopted to increase both validity of results and efficacy of comparisons.

In our experiment the same individual implemented manual unit tests and produced tests throughout automatic tools usage. To reduce the possibility of any influence between these two experiment phases, we first conducted the manual implementation then the automatic tools usage. In this way, we avoided bugs that were found by automatic test generation tools suggesting which tests the engineer could manually implement. What is more, the fact that we first conducted the manual part did not influence the automatic generation (and the experiment results confirmed this), because the human interaction was extremely low in the automatic phase. The only exception was when writing Randoop contracts. However, in this case the programmer did not have to directly write unit tests or give suggestions to the tool about how to write them. He only had to express class properties from a higher level of abstraction. For this reason, the fact that the manual part was done first, influenced only the time necessary to write contracts. The developer already knew the intended behavior of classes, so it was sufficient to express this in a more formal way.

In addition, by assigning the same individual to do both the tasks, we were able to see to what extent his normal abilities could be overcome or helped by the usage of automatic test generation tools. If we had assigned two different people, one for the manual task and one for the automatic tools usage, we would not have been able to be sure whether their personal programming abilities had influenced their results.

For this reason, the only way we could suggest improving the procedure of comparing different testing technique results, is to use a double-blind trial where professional developers are asked to implement manual and automatic tests to a set of representative software components. In this case a high number of developers is suggested, in order to lower differences caused by their respective testing abilities. It would be even better to ask each developer to test different parts of the software, using a different technique for each piece of code. In this way it would be possible to see the results of different testing techniques (without any influence from previous testing of the same code with other techniques) and not to be concerned about different developers abilities.

4 Comparison

After we obtained the unit tests through manual implementation and automatic tools usage, we proceeded to compare the effectiveness of each technique. Here we present the criteria we used to conduct the comparison and the results. In this way we attempt to show how these criteria can be enhanced and extended in order to outline an abstract methodology to perform a practical unit test effectiveness comparison.

We needed universal metrics capable of showing differences in a deterministic manner, in order to perform a correct empirical comparison and link each testing technique with its costs. In addition, such metrics are not only useful for comparisons, but also to help the test engineer to define the quality standard that the tests must achieve.

The quality of tests depends on many facets, for this reason we present the different metrics we used to capture as many characteristics as possible.

4.1 Time metrics

The first metric that we used to compare unit test creation techniques was **generation time**. As testing can consume more than fifty percent of software development time [9], this metric is crucial, especially considering the fact that the first target of automatic test generation techniques is often to dramatically reduce tests production time.

In this generation time metric we combine the time which is necessary to perform various aspects of tests generation. When considering the generation time of manually implemented tests, we also included the hours that were needed to correctly understand the classes we wanted to test and the amount of time we spent on them in order to perform source code refactoring or to improve the existing documentation. We decided to consider the time which was required to understand classes under test only for the manual implementation, even though it was also useful for Randoop contracts implementation. In the latter case only a high level class knowledge was necessary, which was faster and easier to achieve than the deep knowledge required to implement manual tests.

The generation time for automatic tools also includes the hours of effort required to learn their correct and most effective usage. JUnit Factory learning time was really short: it had an efficient documentation and it was straightforward to use. JCrasher had some useful examples which illustrated its usage in a easy way, for this reason it is quite simple to understand how to generate tests through it. Finally, Randoop required the longest learning time, because its documentation was not complete and the user had to learn what contracts are and how to write them. Fortunately Randoop's author replied to our questions in a very short time and has since improved the documentation.

Automatic tools generation time also includes the amount of time we spent interacting with them, both before they generated tests, and afterwards when we were inspecting them. Randoop generation time includes the time, which we spent before creating tests, to find helper classes and to write contracts, and the time -after the test case generation- to verify if error revealing tests reported real failures. JCrasher generation time includes the little effort required to prepare tests and to check error-revealing tests after their generation. Finally JUnit Factory generation time does not include preparation or inspection time, because the Eclipse plugin did all the necessary preparation and the resulting tests were not failing.

Another metric which is time related is test **execution time**. Tests are run frequently, unit tests in particular should be executed every time the tested code is modified, in order to verify that new errors are not introduced. This is one of the core extreme programming practice [8] and is also used in continuos integration and testing techniques. The majority of unit tests should be fast enough to be executed very often on a common development computer. For this reason execution time is an important metric when dealing with unit tests.

Although this metric could be improved simply by using a faster hardware, if the number of test cases is exponential the resulting benefits are less important. In order to deal with this issue, various strategies -mainly based on running only a subset of the regression tests- are studied and suggested [37, 30].

In the case study, even though some test cases could take longer to execute, we noted that the average execution time of a single test case was almost the same. This also reflects the fact that each unit test case should only check a small portion of the tested class [24], and this is usually a fast task. For this reason, we decided not to report execution time, but the **number of unit test cases** created using the different techniques. This gives a more correct suggestion of actual test execution time in general.

Table 1 shows these two metrics (generation time and number of unit tests) by different approaches. We split the Randoop metrics in two in order to reflect its different usages. The number of unit tests generated with the regression avoidance feature is much higher than the number unit tests generated when performing failure detection. In addition, the latter phase required more work to prepare the test creation.

Approach	time (hours)	test case
Manual	490	160
JUnit Factory	15	1,076
Randoop (failure)	115	12
Randoop (regression)	50	5,928
JCrasher	35	10,000+

Table 1. Tests # and generation time, by approach

4.2 Regression tests metrics

One of the target of automatic test generation tools is the creation of reliable regression tests, capable of creating a sort of safety net which could warn of possible errors introduced while modifying the tested code.

The most popular metric used to measure regression tests "quality" is **code coverage**, which is also the easiest metric to use. The main idea is to measure how much of the source code is exercised by unit tests during their execution. This is a white-box metric because it examines the internal coverage of the source code, rather than considering it as a black box.

There are different code coverage types [10], and the main difference between them is the basic unit they use for coverage. Class coverage measures the number of classes that are visited by the test suite; method coverage is the percentage of methods executed, without considering the method size; block coverage considers code blocks as the basic unit; statement coverage (also known as line coverage) tracks the invocation of single code statements. Branch coverage, which can also be found as decision coverage, has a slightly different functioning because it performs its calculatations measuring which code branches are executed. That is, it shows whether the boolean value of a control structure is set to both true and false.

Even though these code coverage flavors show different and complementary information about the test suite.

```
public int foo(int a, int b) {
 if (a > b) {
  //many lines of code
  ...
  return 1; }
 else {
  //only a few lines of code
  return 2; }
}
```

Listing 4. Line code coverage

freenet.support class	Code Coverage (%)			
	Manual	JU.F	Rand	JCrash
Base64	79.5	90.7	93.2	64.6
BitArray	71	97.3	87.7	39.8
HTMLDecoder	55.9	94.6	26.7	91.4
HTMLEncoder	71.1	100	85.8	100
HTMLNode	96.6	100	80.9	29.9
HexUtil	73.9	91.4	68.3	35.7
LRUHashtable	83	100	74.2	55.8
LRUQueue	83	100	88.9	74.2
MultiValueTable	84.8	100	73.9	10.5
SimpleFieldSet	53.8	99.8	64.6	10.1
SizeUtil	82.6	96.9	53.4	53.4
TimeUtil	94.8	100	55.2	45.9
URIPreEncoder	78.7	100	46.1	0.0
URLDecoder	67.5	100	46.5	62.4
URLEncoder	85.7	100	88.8	93.9
Average	*77.5*	*98.0*	*68.9*	*51.2*

Table 2. Line code coverage

For example if we prepare a test that exercises only the first branch of Listing 4, the corresponding line coverage will be quite high, because of the relative length of that part. However, in the same situation block and branch coverage will report a mediocre result. When they calculate it, they do not consider branch or block length.

At the beginning of this subsection we put the word quality in quotation marks, because there is much research [10, 40] explaining that code coverage cannot be considered as a serious metric to measure unit tests quality. Through code coverage we can only see which part of the source code is used during test execution, but not how it is actually exercised (i.e. meaningfully or not). In [7] we empirically proved that code coverage is not a correct quality metric for regression tests. In addition, the results of the analysis on "code coverage and defect density (defect per kilo-lines of code) show that using coverage measures alone as predictors of defect density (software quality/reliability) is not accurate" [40].

However, code coverage can be used as a "negative metric": through it we can correctly see which parts of the source code are not executed by tests. For this reason, it can be used to help selecting and prioritizing tests, especially when dealing with a big software, where the test engineer must be selective about what to test.

We showed that code coverage is not a reliable quality metric, thus we decided not to put much effort into finding and using exotic code coverage flavors. We decided to use the most simple one (i.e. line coverage) to actually see what we could expect, in terms of code coverage, from different testing techniques.

Table 2 shows the code coverage that was achieved by each technique. The highest coverage is obtained by JUnit Factory, with an average value of 98% and a standard deviation of 3%. The other techniques results have a decreasing code coverage average value, but also an increasing standard deviation value. It is relevant to note that the average code coverage reached by JCrasher is more than 50%, even though the tool only produced failure detection tests.

The target of regression tests is to inform the software developer when a change in the tested code also produced unexpected side-effects. For this reason, the best way to verify regression test accuracy is to demonstrate whether they can spot a change in source code functioning. And this is what **mutation analysis** does. The idea of this metric is to introduce mutations in the tested code, for example changing operators (e.g., substitute "- -" with "++"), variables (e.g., reset them, invert boolean values) and in other parts that could be changed without affecting the code execution. A first practical implementation of this analysis was proposed in [12] and [26].

When a mutation is introduced and it changes the expected code behavior, we define the resulting source code as a mutant. If this behavior change is detected by a test case, it is said that the test killed the mutant.

A change in the code could produce a functioning that is not different from the original one, in this case the mutation creates an equivalent mutant.

```
public class Mutable() {
 public static int aMethod() {
  int counter, returnValue;
  counter = 0;
  while (true) {
   //some operations on returnValue
   counter++;
   if (counter==12)
    return returnValue; }
 }
}
```

Listing 5. A not mutated class

For example in Listing 5 we show the original class and in Listing 6 one of its equivalent mutant.

```
public class Mutable() {
 public static int aMethod() {
  int counter, returnValue;
  counter = 0;
  while (true) {
   //some operations on returnValue
   counter++;
   if (counter>=12)
    return returnValue; }
 }
}
```

Listing 6. An equivalent mutant

The effort needed to check if mutants are equivalent or not, can be very big even for small programs [19]. However, we only want to compare the results of different techniques applied on the same source code basis. For this, the same equivalent mutants will appear in the mutation analysis for each technique, influencing the results in the same manner for each technique, without introducing any bias towards our comparison validity. For this reason, we did not consider necessary to find equivalent mutants produced during the mutation analysis.

The mutation analysis process first considers the source code and the corresponding test suite that is not failing (i.e. it is a suite of regression tests). Later, it creates a single mutation inside the source code, increasing by one the total number of mutations created, then it runs the tests again to verify if the mutant is correctly spotted and killed. If it is, the number of killed mutants is increased by one. Then the process restores the starting source code and restarts the cycle again creating another mutation. It loops until the last possible mutation is applied and checked. At the end we obtain a percentage which is the the number of mutants killed divided by the total number of mutants produced. This value is known as mutation score. The mutation analysis is an extremely long and repetitive task, and it is not reasonable to conduct it manually. It is necessary to have an effective tool to automate the process. In our real case study, we used Jester [23] which can perform the mutation analysis on Java code and creates effective and readable reports. In these reports it not only summarizes the mutation score reached, but it also shows the mutants created and which of them were killed by the chosen test suite.

Using Jester it is also possible to choose which mutations to perform in order to obtain mutants. It is sufficient to setup a configuration file, in which the user can specify how a piece of code could be mutated.

```
...
%==%!=
%++%--
...
```

Listing 7. A part of Jester configuration

For example, in Listing 7 we read that each "==" check will replaced with a "!=" check, and each "++" operator will be replaced with a "- -" operator. Using this possibility, we can avoid turning Jester into "a very expensive way to apply branch testing" [27]. In fact, Offutt explained that "the power of mutation depends on the mutation operators that create mutants of the program [...] Experimental research has found that exchanging 0s and 1s turns out to be almost useless because any input will find them. This is known as an "unstable" operator. Replacing predicates gets branch testing, no more no less" [27]. In our experiment we tried a mix of different mutations to obtain the best results from this analysis.

freenet.support	mutation score (%)			
class	Manual	JU.F.	Rand.	JCrash.
Base64	73	58	49	0
BitArray	46	76	82	7
HTMLDecoder	37	37	14	6
HTMLEncoder	28	59	32	6
HTMLNode	94	98	70	10
HexUtil	59	73	57	0
LRUHashtable	91	91	87	0
LRUQueue	58	100	79	0
MultiValueTable	75	96	59	0
SimpleFieldSet	49	84	48	3
SizeUtil	57	97	24	0
TimeUtil	93	99	17	0
URIPreEncoder	19	82	19	0
URLDecoder	71	86	19	10
URLEncoder	67	87	54	7
Average	*61*	*82*	*47*	*3*

Table 3. Mutation Score

Table 3 shows the mutation score for each technique. JUnit Factory confirms the first position which it also reached in the code coverage comparison, with a significant score of 82. Other techniques follow in the same order as in the code coverage comparison. We want to stress that JCrasher tests have no value as regression tests: even though the code coverage that was reached was decent, those tests were not capable of correctly characterize the system. They did not notice any significant source code change and thus were not capable of killing the mutants introduced. In fact, JCrasher tests are generated to identify only failures, and cannot recognize wrong behavior in the semantic of the tested class. This result still confirms that code coverage must not be used as a quality metric for regression tests. Finally, when working with JCrasher, the test engineer could discard the large amount of unit tests that are not revealing faults, because they are not even useful as regression tests.

4.3 Failure detection tests metrics

In [7] we commented that it is really difficult to determine an objective method to measure the ability of tests to detect defects. The number of failures that were found is not a complete metric to asses test quality, because it is evident that different defects could have a different impact on the system. However, in [38] it is shown that it is impossible to generally classify the severity of a defect without knowing the context in which the application was used.

Usually "many programmers might say that a null dereference is worse than not using braces in an if statement" [38]. On the other hand, a logical error caused by a lack of braces could be more severe, and harder to track down, than a null reference.

Approach	Found bugs
Manual	14
Randoop	7
JCrasher	4

Table 4. Number of bugs found, by approach

For this reason, in [7], we considered the Freenet developers' opinion as a further metric. They evaluated the defects detected by the different techniques and they explained that they had almost the same relevance for the system integrity.

This opinion allowed us to show table 4, in which we directly compared the number of defects found to show how the different techniques handled this task. In the manual approach, we also considered the semantic bugs we found, even though they could not be found through automatic tests. They were errors in the semantic of the tested classes that could only be discovered by accurately reading and understanding the source code that was under test.

As a future work it would be interesting to make use of a static bug finding tool, such as FindBugs [20]. This tool, which uses syntactic bug pattern detection and a dataflow component, statically inspects the code to warn about possible defects. These warnings have different levels of priority, and this classification could be useful to give a suggestion about the importance of bugs that automatic unit test generation tools could find. The process should be this: we run FindBugs to classify the warnings in the source code, then we use tools to generate tests and we remove the bugs they reveal. Then we inspect the code again, using FindBugs, to see whether the tools were capable of spotting the same defects and what their relevance was. In this manner we obtain a sort of classification of the bugs that were found.

However, the most effective way of classifying defects relevance is still to study the context and rely on developers opinion. Fortunately, the number of bugs detected is usually not so high that they cannot be manually inspected and evaluated.

4.4 Side effects

The last aspect we considered in [7] when we performed the comparison, was the presence of testing side effects. During the manual test implementation, the test engineer was forced to completely understand the tested classes and methods. In our case, as the documentation was extremely poor, he had to deeply analyze the source code in order to create effective tests. This necessity was highly time consuming, but also led to important side effects.

During manual tests, the engineer noticed and fixed some class performance issues, he increased class readability by removing code duplication and by using better variable naming, and he even split one class into two classes in order to enhance the code reuse.

Moreover, the documentation was improved, not only because of the comments the engineer added, but also because he tried to keep his unit tests as clear as possible. In this way, each unit test was an efficient and updated example of the piece of code they tested.

Only Randoop forced the developer to obtain at least a higher knowledge of the tested code. Instead, the other tools could be used without knowing anything about the system under test. For this reason, all the important testing side effects were not present when using automatic test generation tools. This means losing an important part of the testing benefits and is an aspect that must be taken into account when comparing different testing methodologies.

4.5 A comparison methodology

After detailing how we did the experiment we conducted in [7], it is now possible to abstract a comparison methodology to effectively compare test suites generated through different techniques, and thus compare the techniques themselves.

The first point is to consider failure detection and regression avoidance as two different tasks that cannot be compared because they have completely different targets. A failure detection test could be later used as a regression test, and vice-versa, but in order to compare them we must use different approaches.

It is even possible to use two different systems to create failure detection tests. However, the approach we suggest is the same we used in our experiment. The idea is to use the same system to compare both failure detection and regression avoidance ability.

We suggest starting by using the techniques that generate tests to find defects, and to use them separately but always on the same source code basis. In this manner, we can see if there are overlapping detected faults and, at the same time, we are not influenced by other technique results. In addition, at the end of this phase, we definitively obtain a more correct source code, that we can use later to check tools which generate regression tests.

If the test engineer were able to interact with the developers of the tested code, they might find agreement on a common scale to classify the severity of failures. Then the test engineer should submit the defects he found with different techniques to the developers (in a blind trial manner). They must return the failures classified. This is the best way to classify the relevance of errors, otherwise if it is not possible to interact with developers, the test engineer could use a tool like FindBugs and proceed as we depicted in Section 4.3.

Approach	MS/#(Test cases)	Code Cov.	MS
Manual	0.38	77.5	61
JU.F.	0.08	98.0	82
Randoop	0.01	68.9	47

Table 5. Test case accuracy, by approach

After having classified the detected errors, it is straightforward to compare the different test suites: it is sufficient to use the number and relevance of failures.

Then the regression tests comparison could take place. It is reasonable to use the source code that was used for the preceding phase, but only after having removed all the errors revealed. Usually tools that create regression tests consider the code base as bug-free, and, for this reason, it is useful to try to remove them before with failure revealing techniques.

As for the preceding phase, the different techniques should be used separately but always on the same source code basis. The first metric to calculate is code coverage, which is useful to get a fast overview of created tests. It would also be interesting to use coverages other than line coverage, to see if some techniques are less capable of creating particular scenarios. Finally, mutation analysis has to be performed to obtain the correct mutation score for each test suite. This part is the most important for comparing regression tests, as it is based on a true quality metric.

After these two phases (failure detection and regression test quality comparison), we should continue considering the time metrics. For the generation time it is important to include every aspect that consumed time during test generations -from technique learning to code inspection-. Because there are techniques that greatly improves some of those parts, and this must be taken into account for a valid comparison. For execution time we still suggest not directly calculating it, for example checking the run time, since a faster hardware could dramatically change the values. On the contrary, we suggest considering the number of generated tests, which is a better metric to have a realistic idea of test execution time for every kind of computer. Usually unit tests check a little part of code and for this reason, on average, they take almost the same time to be executed.

Finally, side effects should be considered. Techniques that force the engineer to study the source code he wants to test take a longer time to be used, but they can produce enormous benefits other than the tests themselves. At the end of a comparison which is conducted in this way, the developer has enough information and data to consciously decide which technique is the best for his particular situation. He can also decide to use more than one technique. In the next section we will try to explain what we consider the “best practices” in order to integrate and exploit different techniques with different targets.

5 Best practices

As testing is expensive and time consuming, it is highly desirable to specify successful procedures for doing so [9]. For this reason, after having shown how to compare different testing techniques in order to become aware of their strengths and weaknesses, we here outline a procedure to exploit them so as to improve the testing process.

It is generally agreed that the most effective approach is to combine different testing techniques [9, 15, 16, 39], because each technique could spot different types of faults, and could suffer from a *saturation effect* [22].

When dealing with a legacy system or a modern system which is not yet tested (like the one we used in [7]), we suggest starting with tests for leaf classes or functions (see 3.3). [18] interestingly advises starting from inflection points in order to spread testing effects both to classes that use the tested classes and those that are used by them. An inflection point is a narrow interface to a set of classes. Any change in a class behind an inflection point is either detectable at the inflection point, or inconsequential in the application. However, in [7] we confirmed that finding and understanding inflection points is a hard task when dealing with systems that are not well documented, because the test engineer has to learn the functioning of many parts of the system and might have to read a large amount of the source code. That is the reason why we suggest starting from “leaf classes”, as they are simple to understand and the benefits from their testing are spread across each class that uses them.

When creating unit tests for a not yet tested class, it is reasonable to start from the detection of failure in order to remove all defects and only then create regression tests. The first technique to use should be the easiest one, which does not require the test engineer knowing class functioning internals. For example, considering the techniques we used in [7], JCrasher would be the best choice to start testing with. It requires no test preparation and it can easily find interesting bugs in the code. What is more, the only task that the test engineer needs to do is to check whether failing unit tests that JCrasher generated are correctly reporting real errors. By doing this the test engineer starts to gradually learn what the internal functioning and the meaning of the tested class are.

After this part, the developer should continue using techniques that only need a high level of knowledge of the class under test. For example, from our past experiment, we suggest Randoop, because in order to write contracts it is not necessary to know what the little details of the tested class are, but only to have understood the general meaning of it. In this way, more defects could be revealed and the test engineer further improves his knowledge of the class when he has to verify and remove them.

In [7] we did not use any automatic test generation technique which required a low level knowledge of the tested class, but they could be effective if used in this phase as they could help the engineer in writing test cases. Otherwise, he could directly move to manual implementation of tests. It is evident that this manual work is made much easier by the automatic phases. Here we suggest not only checking for implementation defects but also verifying code documentation and searching for semantic errors. These are testing side-effects that are not produced by automatic tool usage, but they are a fundamental result of testing.

All the unit tests generated in this first “detection of failures” phase could be kept to be used later as regression tests. In the case of automatic tools we suggest keeping only unit tests that revealed errors, because the comparison also proved that not failing tests were useful as regression tests. For example in [7] it was absolutely useless to maintain the thousands of not failing tests that JCrasher produced, because they were completely useless regression tests (as depicted by the very low mutation score they reached).

The second part should be dedicated to “regression avoidance”. During the preceding phase we accurately checked the class we wanted to test, and thus we could be confident enough about its correctness. For this reason, we can move to generate regression tests for it. To integrate the tests we created during the preceding phase, there are two possibilities: integrating them only by manually writing regression tests, or by creating regression tests automatically using appropriate tools and then eventually completing them manually. At the time of writing, and taking into consideration the tools that are available, the choice should be based on a trade-off between test creation time and test execution time.

As depicted in the comparison in [7], the manual creation is much more time consuming, but has the advantage of requiring less unit test cases than automatic tools to reach a good mutation score. This implies a shorter time to execute the whole test suite.

On the other hand, automatic generation tools need a greater amount of unit tests than the manual implementation to reach a high score in the mutation analysis. This means that automatically generated tests will require a longer time to be executed, and this could create problems when using continuous testing and integration. In addition, the number of tests could be so high that it would be a problem to run them in a common development computer. Fortunately the research in this field is extremely active and tools are becoming more and more effective. For example Randoop, which is more modern than JCrasher, is able to automatically remove useless unit tests, and JUnit Factory -which is a commercial software with a bigger team working on it- is capable of reaching a higher mutation score than Randoop with half the tests. Finally, it is always necessary to manually create unit tests to integrate specific scenarios that were not exercised by automatically generated tests.

When describing these "best practices", we assumed the existence of a program to be tested, but they can also be easily adapted for adoption in a *test driven development* (TDD) [21] process. TDD suggests writing automated unit tests before developing the corresponding functional code, in short and rapid iterations. For each small function of the production code, the developer must first implement a test which clearly identify and validates what the code should do. Then the code is only developed to make the test pass, without adding any additional functionality not exercised by the test. The last part of every TDD iteration is the refactoring of both the production code and the test code.

The automatic unit test generation tools, that are now available, can only be used with a preexisting production source code to test. For this reason, when using a TDD process, the first phase is to manually write unit tests defining the expected functionalities. It is not worth writing many unit tests, but just concentrate on the few cases that can exactly exercise the purpose of the production class that will be later implemented. Then, the class could be created following the TDD procedure correctly. Later, the first few tests that have already been created must be integrated using tools to generate failure detection tests, in order to check whether the class implementation is without defects. In this phase, as the TDD process requires a prior knowledge of the class to be tested, the developer could take advantage of any kind of failure detection tools, even though they require familiarity with the class they must create tests for. After all the detected bugs are removed, the user must create regression tests. This can be performed manually or using appropriate tools, and the decision must be based on the trade-off between execution time and creation time, which we introduced before in this section. This stage makes the final refactoring phase easier and faster to accomplish.

By using the procedures we outlined in this section, the practitioner will receive all the benefits that the different unit test generation techniques supply. It will be easier to take decisions about how to schedule time dedicated to the different techniques.

6 Conclusions and Future Work

This paper outlines a novel comparison methodology that can be used to analyze the effectiveness of different unit-test creation techniques. We first explained what unit testing is and that failure detection and regression avoidance are the two issues that it mainly addresses. Then, we showed a real case study in which we created different unit-test suites whose advantages, shortcomings and effectiveness we wanted to assess . In order to do this comparison, we used a methodology that was able to quantitatively give information about test quality. Consequently, we used this practical example to abstract a comparison methodology that could be used not only in this case, but with unit-test suites produced through any technique.

For example, in the real case we studied, we realized that the automatic unit-test generation tools which we chose are really fast, and that they can produce test cases for a large number of classes in a very short time, and that they scale much better than a manual implementation. We also proved that those tools can create trusthworthy regression tests, which reach a high code coverage and, more importantly, a significant mutation score. Moreover, they can help the test engineer find defects, by the creation of unexpected scenarios or by adding a further abstraction level to test creation.

At the same time, the comparison was capable of also spotting the serious disadvantages that these tools suffer when compared to the manual testing approach. First, they do not force the developer to study the code under test. This means not getting the benefits of an accurate analysis of the source code: which could result in finding semantic defects, performing source code refactoring and improving the documentation. In addition, manually created tests are much more readable and are clear examples of the code they test. Moreover, to characterize classes, automatic unit-test creation tools produce at least ten times more test cases than the manual implementation, and even more when finding defects. This could be a problem when employing continous integration and testing, especially if they are used in common development computers.

It emerged, from the real case study, that the comparison methodology was able to richly characterize all the techniques it analyzed. However, in this paper, we also proposed some additional improvements to this procedure, in order to obtain a further effectiveness enhancement and to make it possible to use it for any kind of system that needs a high-quality unit-test suite.

At the end, we also outlined an efficient procedure based on "best practices", that can be used by test engineers to exploit the benefits of different unit-testing techniques. Using the results from the comparison, they can determine every tools advantage, and they can thus follow the "best practices" which explains how inserting each technique in the testing process can obtain a positive integration and relevant improvements.

A future work can involve the creation of a tool which automatizes the measurement of regression test quality (based on code coverage and mutation score) and helps integrate regression tests from different suites. The tool should report not only the scores, but also all the mutants created, with the unit-test cases -from all the different techniques-that are able to kill them. In this way, the test engineers can adopt the most effective test suite as a basis, and they can integrate it with specific test cases from other techniques, which spot mutants that were not killed by the chosen main suite. Consequently, the effectiveness of the main regression test suites receive a significant improvement, without adding redundant unit-test cases.

Another area of future work is to investigate to what extent the human factor hinders the full potential of automatic unit-test generation tools, especially when the user has to inspect the results of the tool (e.g., in order to verify the failing unit-test cases) or to provide some input to further assist it (e.g., writing the class *contract*).

References

[1] JUnit. http://junit.org/. Accessed May 2009.

[2] SUnit. http://sunit.sourceforge.net. Accessed May 2009.

[3] A. Abran and J. W. Moore. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, USA, 2004.

[4] Agitar Technologies. AgitarOne JUnit Generator. http://www.agitar.com/pdf/ AgitarOneJUnitGenerator-Datasheet.pdf. Accessed May 2009.

[5] Agitar Technologies. JUnit Factory. http://www.agitar.com/news/pr/20071015.html. Accessed May 2009.

[6] Apache Software Foundation. Apache Ant. http://ant.apache.org/. Accessed May 2009.

[7] A. Bacchelli, P. Ciancarini, and D. Rossi. On the effectiveness of manual and automatic unit test generation. In *ICSEA '08: Proc. of The Third Int'l Conf. on Software Engineering Advances*, pages 252–257. IEEE Computer Society, 2008.

[8] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[9] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[10] C. Beust and H. Suleiman. *Next Generation Java Testing*. Addison-Wesley Professional, October 2007.

[11] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proc. of the 2006 int'l Symposium on Software testing and analysis*, pages 169–180. ACM, 2006.

[12] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, New Haven, CT, USA, 1980.

[13] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[14] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software – Practice & Experience*, 34(11):1025–1050, 2004.

[15] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 422–431. ACM, 2005.

[16] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 59–68. IEEE Computer Society, 2006.

[17] Eclipse Foundation. What is Eclipse and the Eclipse Foundation? http://www.eclipse.org/org/#about, May 2009. Accessed May 2009.

[18] M. C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, USA, September 2004.

[19] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *J. Syst. Softw.*, 38(3):235–253, 1997.

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[21] D. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[22] M. R. Lyu, editor. *Handbook of software reliability and system reliability*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[23] I. Moore. Jester – a JUnit test tester. In *Proc. of the 2nd Int'l Conf. on Extreme Programming and Flexible Processes*, pages 84–87, 2001.

[24] G. J. Myers. *The Art of Software Testing*. Wiley & Sons, USA, June 2004.

[25] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.

[26] A. J. Offutt. A practical system for mutation testing: Help for the common programmer. In *Proc. of the IEEE Int'l Test Conf. on TEST: The Next 25 Years*, pages 824–830. IEEE Computer Society, 1994.

[27] A. J. Offutt. Jester analysis. http://cs.gmu.edu/ offutt/jester-anal.html, April 2005. Accessed May 2009.

[28] A. Oram and G. Wilson. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Inc., June 2007.

[29] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *Proc. of 2nd International Workshop of Software Quality - SOQUA'05*, pages 242–256, September 2005.

[30] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. *SIGSOFT Software Engineering Notes*, 29(6):241–251, 2004.

[31] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007.

[32] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *ISSTA '08: Proc. of the Int'l symposium on Software testing and analysis*, pages 87–96. ACM, 2008.

[33] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. of the 29th Int'l Conf. on Software Engineering*, 2007.

[34] M. Pezzé and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008.

[35] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[36] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill Higher Education, USA, June 2001.

[37] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[38] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *ISSRE '04: Proc. of the 15th Int'l Symposium on Software Reliability Engineering*, pages 245–256. IEEE Computer Society, 2004.

[39] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *Proc. Int'l Conf. on Tests And Proofs (TAP)*, volume 4454 of *LNCS*, pages 1–16. Springer, February 2007.

[40] K. Stobie. Too darned big to test. *Queue*, 3(1):30–37, 2005.

[41] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. of the 2nd XP Universe and 1st Agile Universe Conf. on Extreme Programming and Agile Methods*, pages 131–143, 2002.

[42] D. Thomas and A. Hunte. Mock objects. *IEEE Software*, 19(3):22–24, 2002.

[43] M. Utting and B. Legeard. *Practical Model-Based Testing*. Morgan Kauffman, November 2006.