# On the effectiveness of manual and automatic 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 importance of testing has recently seen a significant growth, thanks to its benefits to software design (e.g. think of test-driven development), implementation and maintenance support. As a consequence of this, nowadays it is quite common to introduce a test suite into an existing system, which was not designed for it. The software engineer must then decide whether using tools which automatically generate unit tests (test suites necessary foundations) and how.*

*This paper tries to deal with the issue of choosing the best approach. We will describe how different generation techniques (both manual and automatic) have been applied to a real case study. We will compare achieved results using several metrics in order to identify different approaches benefits and shortcomings. We will conclude showing the measure how the adoption of tools for automatic test creation can shift the trade-off between time and quality.*

## 1 Introduction

Testing has such an important practical utility in software design, implementation and maintenance that “the main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests” [3]; even if testing can only identify the presence of defect, not prove their absence. Unit test writing is unfortunately often difficult and time consuming (more than 50% of development efforts should be devoted to it [7]). For this reason different automatic test generation tools have been developed. These tools are apparently really appealing, as they promise to produce satisfactory test cases in a short time with very limited effort. However it is difficult to understand if the resulting unit tests could be comparable to manually implemented ones, and to what extent. Moreover, testing involves different tasks and objectives, some of which could be better manually fulfilled.

“The research community has invested a lot of effort during recent years into developing tools for completely automatic testing” [5]. For this reason, there are many comparisons between automatic strategies (e.g. [8]), but they don’t thoroughly examine those techniques against manual ones. In this paper we show a comparison between automatic and manual strategies using them in depth to generate test suites for a real software project. This way, we hope to drive the software engineer towards the best approach to tackle the tests creation problem.

We decided to concentrate our focus on unit testing because it builds the essential foundations of every test suite. Then, we chose object-oriented software, specifically Java based, because it offers the xUnit framework and the most updated tools for automatic unit test generation. Our case study is a software which is already deployed but still undergoing active development, missing any pre-existing test suite or any formal specification (as it frequently happens).

The automatic test generation tools we have chosen represent relevant examples on how testing generation research could be actually implemented. They handle two different test flavors: regression tests and defect revelation.

We conducted the experiment starting from the manual test case generation (Section 3). This allowed us to specify the domain of classes under test and gain a better knowledge of the tested system. Afterwards, we moved to the automatic tools (Section 4) and we generated tests for the same classes that we previously checked manually. Then, we determined some metrics which could be used to evaluate the quality of the results (Section 5). At last, we compared the different approaches (Section 6) and suggest “best practices” to merge their capabilities (Section 7).

## 2 Experiment environment and subject

The Freenet Project peer-to-peer software has been chosen as subject experiment, because it owns all the features we were interested in. The software is still in production and it has been implemented in Java without a significant test suite (about only 14 test methods for the whole code). As with much legacy softwares, it is not provided with

978-0-7695-3372-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ICSEA.2008.66

sufficient documentation (neither at low nor at high level) and many authors of parts of the source code have left the project during its lifetime. Although there is an active community of contributors, the project can count on only one full time developer. Despite these traits, Freenet evolves continuously and could receive great benefits from a reliable test suite.

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 hasn't been an issue, whereas we got serious benefits in computational time using a research workstation (dual Intel Xeon 2.8Ghz processor, with 2GB of Ram) when generating tests with the automatic tools and calculating the value of metrics (above all mutation score (Section 5)). The full access to the official Freenet subversion repository allowed us to cope with frequent Freenet evolution and to see the influence of our tests directly on production code. The tests developer was a graduate student with relevant (about three years) industrial experience in Java programming and a xUnit basic knowledge.

## 3 Manual tests implementation

We began the experiment dedicating three full-time work months (about 490 hours) to manual tests creation. We wanted to concentrate our efforts on "leaf" classes (i.e. without strong dependencies on other classes of the project), which were possibly referenced through the whole code. This to avoid spending too much time in studying the project architecture and, at the same time, to permit seeing effects of tests as broadly as possible. With the help of the Freenet community, we decided then to concentrate on the freenet.support package that contains miscellaneous helper classes, most of which are reused in other code.

Currently, a formal method to manual testing approach is not clearly defined (some research is trying to introduce it [11]). Thereby, we based our implementation on widely-accepted best practices [7]: boundary-value analysis, data structures analysis, control-flow path execution, error management paths execution (i.e. exception handling checking), mock objects and environments generation.

We used the white-box testing method [1], that led to an accurate analysis of the source code of each tested class. With this approach we managed to resolve some documentation and design issues we encountered in the tested code. Moreover, we found some semantic defects, which we fixed with the aid of the Freenet community.

At the end of this experiment section, we immediately noted that the number of classes we had been testing was under our expectations. The cause can be attributed to the serious lack of documentation in the project: it was not sufficient to spend time reading the classes source code to understand expected behavior, so we had to interact repeatedly with the Freenet developers to ask for clarifications. At the same time, we had 160 test cases documenting the tested classes through easy-to-read examples, we discovered 14 defects within 15 classes and we led to source code refactoring and updating.

## 4 Automatic tests generation

The automatic test generation tools we have chosen try to deal with the creation of regression and defect revealing tests. The main difference between those two issues is within the approach used to solve the "oracle problem" [1]. In the first case, the tools create tests that characterize the actual behavior of the code. They record and test not what the code is supposed to do, but what it actually does. They consider the tested software as the oracle (i.e. whichever output it produces is correct). We selected two tools (JUnit Factory and Randoop) characterized by good usability and a very reasonable learning curve.

On the other hand, the tools which generates defect-revealing tests could solve the oracle problem either by asking the user to define what he expects from a class or by relying on language error handling. We wanted to find the easiest and most effective solution available, so we chose two tools: Randoop and JCrasher. While the latter bases its decisions on Java raised exceptions, Randoop only requires the developer to describe invariants he needs on the classes through a Java interface implementation. Hence, we prevented the engineer from wasting time learning new formalisms to describe the tested system.

We generated regression tests on the source code present at the end of the manual test implementation. On the contrary we performed automatic defect revealing test generation on the source code as it was before the manual implementation. This permitted a better comparison between manual and automatic test generation approaches. Thereby, we could verify whether manually revealed defects were the same as detected by tools and the ability to detect inserted behavior modifications.

Although a test that is used to reveal a defect could be later used as a regression test, we decided to divide regression test purpose and defect revelation purpose in our analysis. In fact defect revealing tests are only a little percentage in code coverage, so they don't provide a sufficient impact when analyzing regression tests quality.

### 4.1 Randoop

Randoop is an experimental tool distributed under the MIT Open Source license. It is based on the research of Dr. Ernst and C. Pacheco [9].

Randoop can generate both regression JUnit tests and error-revealing test cases. The input to Randoop is a set of Java classes to test, a time limit and an optional set of contract checkers. The resulting output is a JUnit test suite.

Regression tests are tests that don't violate provided contracts but capture the current implementation, while contract-violating tests could be error-revealing.

"Randoop generates unit tests using feedback-directed random testing, a technique inspired by random testing that uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs" [9]. The tool creates method call sequences by randomly selecting a method to call and choosing arguments from previously constructed sequences. When created, a new sequence is executed and checked against a set of contracts. Sequences that lead to contract violations are output to the user as contract-violating tests. Sequences that exhibit normal behavior (no exceptions or contract violations) are output as regression tests. To avoid useless computation, sequences exhibiting illegal behavior are discarded.

Using Randoop for regression testing was really simple. We only needed to specify which classes to test and some helper classes that were needed to execute certain methods (e.g. to check java.util.Collections it would have been necessary to provide an actual implementation such as java.util.TreeSet). Randoop then returned several test cases that could be inserted into the project and used with the standard JUnit framework. As Randoop is designed to be deterministic when the code under test is deterministic itself, during the experiment we ran the tool with different random seeds to increase result quality.

About defect revealing aspect, writing a contract consists in implementing a Java contract-checking interface. The developer specifies a property that should hold (e.g. a method postcondition or an object invariant) for classes under test: he receives the objects that have been randomly created by Randoop and he verifies whether their state is correct. For example, he could check if no loop cycles are present in a given linked data structure (as we did for HTMLNode class). We measured how this approach helps the developer to concentrate more on class characteristics than on the actual tests implementation. To the contrary we realized how this approach is excessive when creating tests for static methods. Those methods aren't based on object state, so there are no invariants to check. For this reason, writing test cases directly is far more effective.

### 4.2 JUnit Factory

JUnit Factory [4] is a service from Agitar Software capable of generating regression tests for given Java classes. It could be used only through a specific Eclipse plug-in that permits to choose classes to tests and outputs the resulting test cases.

It is not possible to see how this tool really operates, because it is an online service and computations are done on Agitar servers (this could be a security issue when dealing with proprietary software). The company explains that it is based in part on the same research used to create Randoop.

JUnit Factory usage is straightforward and it generated the test cases for the 15 classes in about an hour. It must be pointed out that result test require proprietary libraries, in addition to the JUnit framework, to be executed.

### 4.3 JCrasher

JCrasher is "an automatic robustness testing tool for Java code" [2]. It attempts to reveal defects causing the program to throw undeclared runtime exceptions. After a Java classes list is provided, JCrasher generates random but type-correct arguments for public methods. When an exception is raised it uses heuristics for determining whether it should be considered a defect. In this case it produces JUnit test cases which reproduce the error-revealing behavior.

JCrasher generates input data in a truly random fashion, but it covers also some boundary cases (such as negative numbers or zero for integer functions). This behavior is helpful in the testing unexpected scenarios. Then it uses only type-correct arguments, preventing useless exceptions computation and analysis. Finally, the heuristics used to determine exceptions meaning are important to output only cases which are relevant for the developer.

JCrasher process is completely automatic, only defects reporting test cases must be manually checked.

## 5 Metrics

Selected tools generate two sorts of tests: error revealing tests and regression tests. Those different kinds of tests need different metrics to evaluate their quality.

Defect-revealing tests could be mainly evaluated on defects they find. Although the fixed number of errors is the first and easiest metric to consider, it is not sufficient to evaluate their quality. However it is really difficult to determine an objective quality index in this domain. With this case study, we are trying to report our opinions and those of Freenet experienced developers about revealed defects.

The starting metric used for regression tests was code coverage. It computes the percentage of source code lines actually visited during tests execution. This is necessary but not sufficient to determine tests quality, because it "merely measures that a statement, block, or branch has been exercised" [10] giving no information about the meaningful execution of the code. Also 100% code coverage cannot guarantee exhaustive testing, since all the code would need to be executed with all possible combination of variable values, which is likely to be impossible.

Regression tests aim to capture the system behavior to avoid unwanted modifications when any controlled change is made. This leads to the introduction of an efficient quality metric for this kind of test: mutation analysis, which is based on three simple steps. First, producing a mutant (i.e. slight change the source code). Second, running the tests to verify whether they can kill the mutant (i.e. there is at least one test failing). Third, rolling-back to the code before the mutation and preparing to produce a different code change. The process can be repeated many times, each one producing a slight change in the source code. We used the tool Jester [6] to do the automatic mutation analysis and we kept the default configuration offered for the mutation kinds. The source code we used for regression testing was the same as for manual implementation, so mutants that we created were identical for each technique. We did not need to remove equivalent mutants, as they had no impact on the final comparison. For each class and tests generation approach, we eventually obtained a "mutation score", measuring the ratio between killed and total mutants.

## 6 Results comparison

We conducted results comparison at different stages: tests generation time (it includes learning time for automatic tools), number of produced test cases, code coverage and mutation score reached, and revealed defects.

### 6.1 Tests produced and generation time

In three full-time work months, we carefully tested only 15 classes. The developer software knowledge increases along with tests implementation, however there was no serious impact on the tests implementation rate. This experience shows we cannot expect more than one reliably tested class per day (7 hours).

On the contrary, it took about 15 hours to learn how to use JUnit Factory and produce regression tests for the same 15 classes. We needed almost twice as much to learn and generate Randoop regression tests and JCrasher defect-revealing tests. Randoop failure-revealing tests creation costs more time because the developer must write contracts.

In Table 1 we can see that the test case number is significantly lower in the manual approach. We considered our work as reliable with about 11 test methods per class, while automatic tests generation tools needed at least 10 times this value. JCrasher was the least efficient tool in this case, because it created more than 10,000 tests.

| **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**

| **freenet.support** | **Code Coverage (%)** | | | |
|---|---:|---:|---:|---:|
| **class** | **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* | *77.46* | *97.99* | *68.95* | *51.17* |

**Table 2. Code coverage**

### 6.2 Code Coverage

Table 2 shows the code coverage reached in directly tested classes only. JUnit Factory obtained the best results, followed firstly by the manual implementation, then by Randoop. It is interesting to notice that JCrasher reached a 50% code coverage, even if its purpose is only defect revealing tests generation.

Code coverage is really fast to compute so we used it during manual implementation to verify if we covered all source code parts. It has been a useful index to select and prioritize tests.

In the same manner we invoked the Randoop tool several times, with different random seeds, to improve code coverage as much as possible. We reached the maximum value (i.e. further repetitions didn't increase the code coverage) in about four repetitions.

| freenet.support<br>class | mutation score (%) | | | |
|---|---|---|---|---|
| | 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 | 12 | 19 | 0 |
| URLDecoder | 71 | 86 | 19 | 10 |
| URLEncoder | 67 | 87 | 54 | 7 |
| *Average* | *61* | *77* | *47* | *3* |

**Table 3. Mutation Score**

### 6.3 Mutation Analysis

In the mutation analysis, JUnit Factory maintained the best results, as seen in Table 3; the manual implementation and Randoop followed – as in code coverage. Although JCrasher code coverage was about 50%, mutation score tends to zero. Confirming that code coverage must not be used as a quality metric. The tests generated by JCrasher cannot be used as regression tests: the developer should discard them if they don't reveal any fault.

### 6.4 Defect detection

In Table 4 we see the number of defects detected by different tools. The manual approach obtained the best results also in detected defects relevance, which was confirmed by Freenet developers. We detected defects in the handling of arguments of methods in different classes. In HTMLNode class we removed some bogus and useless methods, we added relevant checks on node attributes and we found that it was possible to create an undesired loop in the data structure. In HTMLEncoder and HTMLDecoder we added a missing HTML entity. In SimpleFieldSet we fixed a data structure method. In BitArray and HexUtil we fixed two problems concerning boundary cases.

When using Randoop we could reach interesting results, similar to the manual ones. We found the same loop problem in HTMLNode, and the defect in SimpleFieldSet. Clearly it did not detect any semantic defect.

JCrasher revealed four defects that were never detected using other techniques. One was found in BitArray class and three in HexUtil class. All defects concerned erroneous method arguments handling. We solved each problem by adding an argument checking and throwing an "IllegalArgument" exception when needed.

| Approach | Found defects |
|---|---|
| Manual | 14 |
| Randoop | 7 |
| JCrasher | 4 |

**Table 4. # of defects found, by approach**

## 7 Best practices

The work and the results depicted in this paper could be also used to outline the most suitable approach to test a software taking advantage of both the manual and automatic test generations.

In a not yet tested class it is useful starting with defect revelation. The best practice here is to use a tool like JCrasher which could find defects without the need for a prior tested class knowledge. Once failing tests are obtained, the developer could check the tested class to see if they correspond to errors. So he begins acquiring some knowledge of the class and removes the first defects from it. As seen in table 3, when using tools such as JCrasher we found it is not worth to keep tests that didn't find errors, because they have a very low quality as regression tests. For this reason, at the end of this phase it is useful to keep only the tests which revealed errors, in order to use them later as regression tests.

Subsequently, the developer should start writing Randoop contracts for the tested class. In this way he gains a high level knowledge of the tested class functioning. As did for JCrasher results, it is useful to include tests that revealed defects in the test suite for later use as regression tests.

Afterwards there is the manual phase. The developer, who finally owns a good knowledge of the class, could manually remove semantic defects and add some documentation where needed. This phase is highly facilitated thanks to the previous ones.

Later, after having removed defects from the class, the developer could move on to regression tests. He could proceed in two different ways, depending on available time: if he has only a short time to finish testing the class it is sufficient to run automatic regression test generation tool, such as JUnit Factory, to obtain high quality tests that capture class behavior (as seen in table 3). The drawback is in the huge number of tests created, which could make difficult to run tests frequently. Otherwise, if there is sufficient time, the developer could manually create a few regression tests to reach a reasonable code coverage. Then he could inte-

| Approach | #(Test cases)/MS | Code Cov. | MS |
|---|---|---|---|
| Manual | 0.383 | 77.46 | 61 |
| JU.F | 0.071 | 97.99 | 77 |
| Rand | 0.007 | 68.95 | 47 |

**Table 5. Test case accuracy, by approach**

grate his tests using automatically generated ones. To avoid the big test quantity problem, it is important to add only the automatically generated tests which increase current code coverage or cause a not yet tested behavior. Even though there are several automatically created tests to check for the integration, a developer with a good knowledge of the tested class is not going to find the task highly time consuming.

Finally the developer could improve the tested class with some manual refactoring. This task is made a lot easier by the usage of just created regression tests. If the class functioning is not maintained during the refactoring, regression tests will reveal that it is failing.

Using these approaches it is possible to receive all the benefits that manual and automatic approaches supply. It also facilitates the decision about how to schedule time dedicated to manual testing.

[5] illustrates an Eiffel language testing environment which integrates manual and automatic generated tests. It uses a different approach than xUnit and it does not permit to integrate external automatic test generation tools.

## 8 Conclusion

This real case study shows that automatic test generation tools produce some valuable results that could facilitate the creation of a test suite.

First of all, automatic tools are really fast: they produce test cases for a big number of classes in a very short time, and they scale much better than manual implementation. Secondly, the regression tests they generate achieve a high score in code coverage and, what is more important, in mutation score. Therefore a developer can trust the quality that JUnit Factory or Randoop reach in characterizing a tested system. Finally, in the defect-revealing task, automatic test generation tools could help the developer creating unexpected scenarios (e.g. JCrasher found four defects that were not revealed by other approaches) and adding a further abstraction level to tests creation (e.g. when writing a Randoop contract, the developer could concentrate more on tested class behavior than on tests implementation).

However, we showed that there are negative aspects that cannot be underestimated when using these tools. First, the developer is not forced to study the tested code, and we saw that manual implementation imposed an accurate analysis of the source code, which added outstanding testing "side-effects" such as code refactoring, documentation updating and semantic defects finding. Secondly, as depicted in Table 5, automatic tools needed at least ten times more test cases to characterize classes than the manual implementation, and even more to find defects. This could be a problem when dealing with a big system because tests execution could be quite time-consuming and impact negatively on the overall development time.

Moreover, automatically generated tests are extremely terse and could have really confusing method and variables naming (e.g. Randoop and JCrasher test cases). On the contrary, carefully hand-crafted tests are self-explanatory and they serve as an efficient documentation by example.

In conclusion, we saw that automatic tools can be partially trusted and permit to increase the tests creation rate. However automatically generated tests must be correctly analyzed and tweaked; they have to be integrated with a manual approach which increases tests accuracy and pertinence and leads to important "side-effects" on tested code.

## Acknowledgements

**We thank the support of the Italian Ministry of University and Research, under project contract FIRB2005-TOCAI.**

## References

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

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

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

[4] JUnit Factory. http://www.junitfactory.com.

[5] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: The autotest experience. *hicss*, 00:261a, 2007.

[6] 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.

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

[8] S. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Softw. Eng.*, 27(10):949–960, 2001.

[9] 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.

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

[11] 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.