

# Automated Regression Testing using Randoop

Steve Rowe, Peter Gebhard, Kaustav Chowdhury  
{scrowe,gebhard2,kchowdh2}@illinois.edu  
Department of Computer Science  
University of Illinois at Urbana-Champaign

## Abstract

*Regression testing is an important tool to keep new bugs from creeping into previously working portions of the code base. The intent of this study was to evaluate the effectiveness of feedback-directed random test generation in the form of Randoop for regression testing. In many software projects today, regression tests are hand-crafted. They may be some combination of unit and acceptance tests. It is our supposition that this is insufficient. There is a large portion of the state space that is not covered by such tests and breaks in these areas will go unnoticed. Randoop as an automatic regression testing tool claims to answer most of these concerns. The main intention behind this project was to evaluate the effectiveness of Randoop in finding regression differences across multiple software versions. Also, as first step, different Java source code classes were mutated with random changes and successfully caught by Randoop.*

## 1. Introduction

In the rush to ship a new software release or to complete a component's new feature, there can often be a desire to shortcut a formal, expected process. It is under these circumstances, however, that the need to properly stay focused is likely to be the greatest. After all, if a software team neglects proper quality control steps, they risk having to deal later with a loss of customers and tarnished reputation if critical bugs happen to be encountered in their work.

Since testing is such a critical step in the process of developing a quality software product, it should never be left to the last minute before it could be shipped to end users. Unexpected problems in a schedule and the constraints of a budget, though, could limit what a team can hope to accomplish. Fortunately, automated test generation and execution is able to free up a developer's time for it to be spent on more urgent tasks, but the utility of an automated test generator is only as great as how effectively its outputted tests can perform.

This paper focuses on the use of an automated test generation tool, Randoop, in performing regression testing. Whereas other studies are concerned with ways to improve the cost-effectiveness of regression testing [5], our study simply wishes to explore the use of Randoop for performing regression testing on real-world projects and to learn the benefits and drawbacks of automating regression testing.

The paper is divided into six sections as outlined. Section 1 gives an overview of regression testing, why it is important, and how used the Randoop tool in our testing. Section 2 continues with a more thorough explanation of Randoop's role in our experimentation test runs. Section 3 discusses the details of how testing was performed on each of the software project that were used in our modified Randoop test generation process. Section 4 explains the results of testing the various projects, as well as some of our concerns about the tool. Section 5 details our lessons learned from using the Randoop tool and performing regression testing on large open-source projects. Finally, Section 6 concludes with a brief description of our findings and some thoughts on potential future work.

## 2. Background

In this paper we examined the feasibility of applying Randoop as a tool for performing automated regression testing. A regression is a software defect that has been introduced into previously working code during the course of development. In other terms, a regression is considered a new failure that has appeared in previously correct code. This type of defect does not refer to any software bug that may appear in source code that has been newly developed from a previous checkpoint or milestone, but rather, it appears in existing functionality. Regressions are commonly caused by side effects of changes in the software. De-

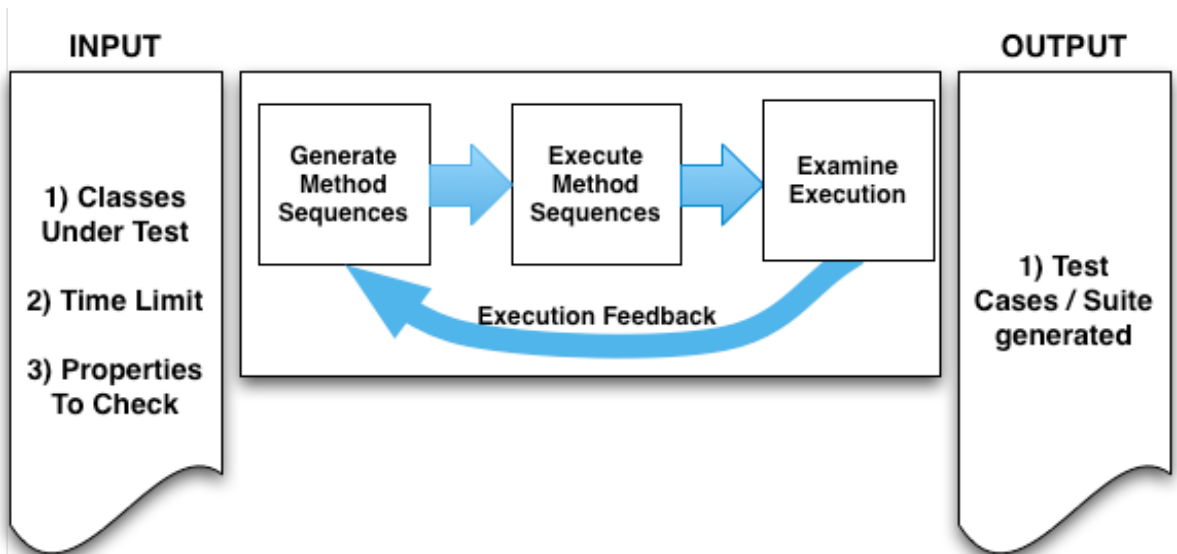


Figure 1: Randoop automated test generation process

velopers may not realize that the fix they made in one area has repercussions on another. Alternately, they can also be caused when different code branches are merged and a fix is accidentally lost.

Regression tests can be developed and executed both manually and automatically depending on the context of the test case, alongside other forms of testing. Regression testing suites will tend to expand throughout the entire lifecycle of a software system, as these tests can give the system's developers confidence that past fixes have remained intact and no previous behavior has been unknowingly modified. The cases will be known to pass on one version of software, after which they are then also run against subsequent versions. If any of the tests fail, the developers should be warned that a regression has taken place. The typical process for performing regression testing consists of running a suite of tests against every build of a project, and the desired results should consist of passing tests such that any failure can be confidently attributed to a regression.

In order to study a testing method that could improve both the time of development and the thoroughness of regression test suites, we chose to focus on the Randoop automatic test generator as we experimented with automating regression test suite generation. The end goal for using Randoop in a regression test suite generation role was to show that it is possible to extend the tool to allow for regression testing in an automated manner. Fortunately, our understanding of the tool's opera-

tion convinced us that the automation of regression test suite generation could still be done with a minimal amount of external controller code.

### 3. Randoop

Before beginning experimentation using the Randoop tool, it was important that we knew how it operated. Randoop is a feedback-directed random test generation tool. It is an enhancement to random test generation which uses test outputs as inputs to additional tests. Provided with a Java class, it is capable of generating an applicable unit test suite. It utilizes feedback obtained from executing the test sequence during creation to steer the test generation toward new, legal sequences. [6] Randoop's output consists of a suite of JUnit tests which can then be compiled and executed as any JUnit test is. In addition to looking for contract violations and exceptions, it produces tests for regression testing. This suite of JUnit tests is in the form of java source files containing test methods and assertions. The assertions will fail if the behavior varies from the initial baseline.

Randoop's regression testing capabilities are fairly primitive. It runs a test sequence and then records the primitive output of observer methods. Randoop uses the following criteria to determine if a method is an observer: 1) it has no parameters, 2) it is public and non-static, 3) it returns values of a primitive type or string, and 4) its name is size, count, length, toString or begins with get or is. It is thus quite restrained in what it can record. It

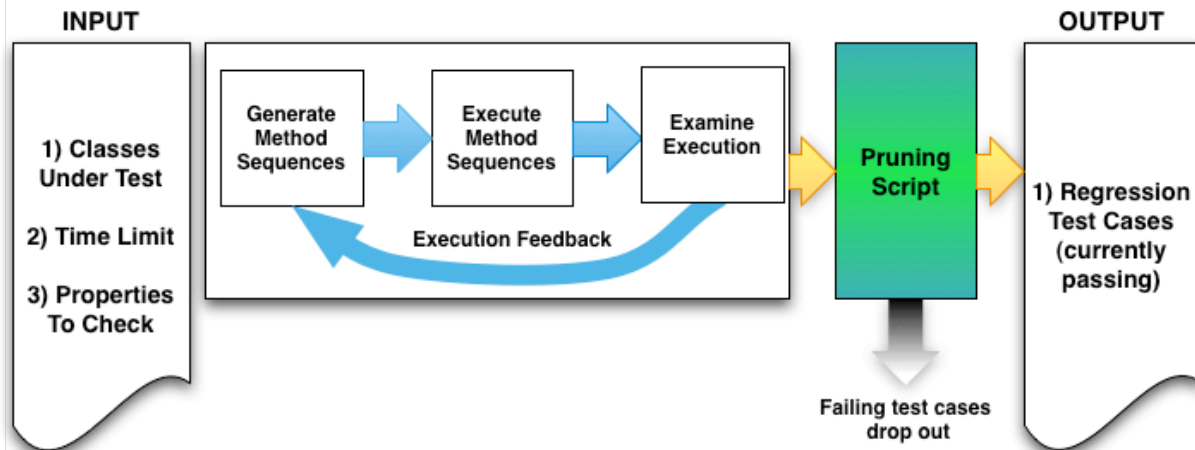


Figure 2: Randoop automated test generation process with pruning modification

cannot record non-primitive types, results from complex functions, or intermediary outputs. Figure 1 depicts the ordering and details of Randoop's processing.

Once Randoop has run, the JUnit tests can be run against a different version of the software and new failures are regressions. In practice we found that much of what Randoop records as regression tests is data that changes from one run of the program to the next. This makes determining regressions difficult. To overcome this, we wrote a tool to iterate over a set of tests until the results were clean. Figure 2 depicts the modification made to the typical Randoop process in order to remove unwanted, failing test cases.

#### 4. Experimentation

To test Randoop's ability to find regressions, we tested it against real projects. Many previous attempts to validate regression testing software have run the tools against mutated software with known faults. This does not allow an adequate exploration of whether the tools can find regressions in real code. There is no guarantee that the sorts of errors introduced in mutation are the same sort that would be introduced in regression testing. We chose to run Randoop across several open-source tools including Google Collections, jGnash, jEdit, and jGraph. We first created a baseline of tests and then ran the same tests against more recent versions of the software.

Successful regression testing requires a clean baseline to compare against. To overcome Randoop's limitations in creating the necessary clean baseline, we wrote a tool to iterate over a set of

tests until the results were clean. The idea was suggested by Tao Xie[10]. The tool, rd.pl, uses Randoop to generate a suite of tests. It then compiles the tests and runs them in JUnit. Any assertions that fail during this run are removed from the test source code. The source is then recompiled and the tests run again. This process is continued until all failing assertions are removed. It is necessary to repeat the process several times because JUnit test methods will exit after the first assertion fails. If there are several failing assertions in one test method, only the first will be executed. In practice, we found that the tests typically have to be pruned 3 times before they run cleanly.

Pruning failing tests is acceptable because we are looking for regressions. Regressions are failures in a newer build of a program that didn't exist in an older one. If an assertion fails against the version of the software it was created with, it is useless for regression testing purposes. Depending on the sort of software under test, there may be many cases that fail against the original software or there may be few. The cause of the failing baseline cases was in all cases random data. Some methods return data that is intentionally random. For instance, we tried testing a tool called Jasypt [11] which is an encryption library for Java. Out of the 2,700 test cases generated by Randoop, more than 2,600 of them failed. Examination of the failures showed that they were caused by different cyphertext output each run. This is expected behavior. Another example of baseline failures comes from the Google Collections suite. Some number of the test assertions were validating the toString methods of the various classes. Each of these strings contains a number specific to that

instance of that object. These identifying numbers vary each run and thus cannot be used for regression testing.

We considered two other approaches to dealing with this random data. These were setting any random seed/salt and pruning during test case creation. The first is untenable. There are no standardized ways of setting the random seeds for all programs. A generic testing framework like Randoop cannot expect to handle this. The second turns out not to work. The idea is to check the return from a given method twice. If the results do not match, do not write an assertion for that case. This turns out to only solve part of the problem. It can handle the cyphertext case because that output will vary each time the method is called. It will not, however, handle the toString case. Those numbers only vary with each invocation of the program and thus will be the same each time a method is called within a given instance.

After pruning all tests with inconsistent results, we then ran the tests against the jar or .class files of a newer version of the software. Any assertion failures found at this point would be regressions. Our detailed findings can be seen below.

We found that sometimes the code was changed between builds or releases. A method's signature may have been modified or the method may have been removed entirely. In these cases an error is generated. We did not create any tools to prune errors, just failures. The reason for this is that the errors were already separated from failing cases in the results and thus easy to remove from collection.

## 5. Results

### 5.1. Google Collections Library [1]

The Google collections library is an extension to the standard Java collections framework which adds new types such as BiMap, Multiset, and Multimaps along with several utility classes. This fits well in the sweet spot for Randoop as the classes require no special configurations or data and have deterministic behavior across runs.

We tested for regressions in the 8/20/2008 alpha build from the 5/30/2008 alpha build. A series of classes were tested resulting in 4,209 test cases. Running these against the same build (0530) resulted in 820 failures. Running them against the updated build (0820) resulted in 734 failures but 549 errors. It is difficult from these results to determine whether there were any regressions or not.

Manual investigation showed that the tests failing in both cases were toString cases. The errors were because methods no longer existed. They had been renamed or their signatures changed.

We then ran the same list of classes through the pruning tool. This time 4,174 test cases were generated. The delta is because of the random nature of Randoop. All 4,174 test cases passed when run against the same build they were generated against. When run against the updated build, there were 0 failures and 522 errors. Errors again were because of restructuring. The lack of failures indicates that there were no regressions found. This particular collections library is very well tested. Google boasts 25,000 unit tests for it. It is thus not unexpected that no regressions were found.

### 5.2. jGnash [2]

jGnash is a Java cash management program for the general public. It is a rich client application that includes a GUI, basic accounting operations, network-aware architecture, and sophisticated output generation for exporting and printing files and reports. The choice to experiment with running Randoop-generated regression testing against jGnash was guided by the fact that it offered a different form of software than the Google Collections library. In the case of jGnash, it was not possible to get access to the revisions of the source code beyond each individual release.

We were able to download several versions of jGnash against which we could apply Randoop and our test case pruning script. The two versions that were chosen to be tested were versions 2.0.0 and 2.0.1. The thought behind choosing such a minor bump in the version number was that we might minimize the possibility of false positives that could appear if a program has changed significantly between major versions due to the refactoring of interfaces. Those modified interfaces will cause problems when test cases generated on a previous version are tied to a particular interface, and they will result in those tests being marked as failures.

After running Randoop on jGnash 2.0.0, any tests that were failing on that version were pruned before continuing on to run those same tests against jGnash version 2.0.1. In the final test run of the original 18,294 jGnash 2.0.0 test cases against the jGnash 2.0.1 code base, there were no JUnit failures and there was only a single error which occurred due to a method no longer being present in the more recent version. These results

could be interpreted several ways. First, we could have taken the lack of failures as a show of jGnash's robustness in its releases. Second, we could have inadequately tested relevant classes that would have changed between these versions, as we selectively chose classes from the application to test. Third, Randoop's test generation algorithm was not able to generate sufficiently thorough tests for finding faults in the jGnash software. Keeping in mind the decision we made in choosing the two closely released versions of jGnash, we hoped to find different results in testing revisions of jEdit which had many hundreds of revisions between them.

### 5.3. jEdit [3]

jEdit is a popular Java text editor which is designed with a programmer's needs in mind. It contains many features that are desirable to a software developer such as syntax highlighting, extensibility through plugins, and a great degree of customizability among others. By choosing to test jEdit after having tested jGnash, we are able to maintain some similarities in the type of software that will be tested. Certainly, jEdit has a GUI that is hardly less complex than that of jGnash. Also, jEdit has a large amount of complexity in its backend from its plugin architecture and support for customizability.

An important reason for choosing the jEdit project for our testing was because it had over 14,000 revisions in its Subversion repository on SourceForge. This large number of revisions meant that we would be able to choose from a wide range of releases and intermediate states. The goal when studying jEdit was to be sure that we would choose revisions that were adequately distant in when they were committed such that our regression testing could be run against versions of the software that would have more modifications and greater potential for regression bugs to be found.

Three revisions were chosen to be used in the Randoop testing: rev. 13041, rev. 14103, and rev. 14138. Between these three revisions, many of the modifications that were being made by jEdit developers were involved with different elements of the GUI. Frequently, changes were made to various individual plugins, however, those changes were not our primary focus as those were seen to be external to the application. We focused on testing the core components of jEdit, and in this case, its GUI classes were the ones given to Randoop for test generation. Unfortunately, our attempts to

complete testing of jEdit hit obstacles in trying to fully compile the generated JUnit tests. Randoop had been able to complete its process of generating the appropriate JUnit tests, but it had not taken into consideration any external dependencies on other components of the Java API. The tests continued to have errors when attempting to compile. This experience is just one example of Randoop's deficiencies that we came across in our work.

### 5.4. jGraph [4]

jGraph is a freeware Java Graph utility used for the visualization and layout of graphs. Unit tests were generated and initially run against multiple classes in versions 5.8.3.1. The same set of tests were then ran against versions 5.9.2.2 and 5.10.1.4. A few of the failed tests were noticed mainly because change in method signatures. In most of the newly introduced failed cases either the number of parameters passed or data types of parameter passed changed. Total number of test cases used were 3146 spanning across multiple classes. When ran against 5.9.2.2 it resulted in 613 failures and when ran against 5.10.1.4 it resulted in 589 failures and 634 errors. From the results it was not easy to infer whether any of those are actual regression differences. On further closer analysis some of the differences were noted to be because of the difference in method parameters.

## 6. Lessons Learned

During the course of this project we learned a lot about automated regression test generation, both where it can be useful and where it is very difficult to apply.

It is hard for a tool like Randoop to create tests for software that requires complicated setups or which operates on particular data formats. For example, it is hard to test an XML parser with Randoop because Randoop doesn't generate XML very well. Likewise, it can be difficult to test a framework which is to be inherited and extended. Network facing tools require configuration pointing at specific servers which can also be difficult to facilitate. This is unfortunate because if one looks over the projects on SourceForge or Google code, most of them fall into the categories of tools that are hard to regression test.

Regression testing cannot survive large-scale code restructuring. One tool we tried to test was called CSPoker [12]. Randoop was able to easily generate tests for it, but the code was moved

around to new namespaces between builds and Randoop could not handle this well. Because of this, it is best to run regression tests between minor versions of the software, not between major releases. The further apart the code is, the more likely it is to be restricted in a fundamental way.

Regression testing is ineffective when most of the output varies from program instance to program instance. The Jasypt program mentioned above had less than 10% of its tests that were stable in against the same build.

## 7. Conclusion

We have outlined the basics of a process for testing the real-world effectiveness of a regression test tool. It is necessary to begin by obtaining a clean baseline against one build of the program. After that, any failures when running the same test against a newer build are regressions. Errors caused by movement or change of methods should be discounted. In the programs we tested we found no actual regressions, and while this is unfortunate for our own study, it is not fully unexpected considering the fact that the selected software project are already quite well tested and Randoop is also still in a very experimental state. Building off the work we have done here, future studies might focus on improving gathering a canonical set of applications and versions to test against. More work on the regression testing abilities in Randoop would be helpful. Many of the techniques pioneered by Orstra [10] could be successfully paired with the feedback-based random test generation provided by Randoop. Particularly, there is more work to be done to determine the right type of methods to record. Randoop is very conservative and records only obvious observer methods. Orstra sometimes records the internal state of an object which violates the principle of encapsulation, and a balance at some point in the middle is likely preferable.

## 8. References

[1] Google Collections Library, <http://code.google.com/p/google-collections/>

[2] jGnash, <http://sourceforge.net/projects/jgnash/>

[3] jEdit - Programmer's Text Editor, <http://www.jedit.org/>

[4] J. Andrews, A. Groce, M. Weston, R. Xu. Random Test Run Length and Effectiveness. In Automated Software Engineering, 2008.

[5] H. Do, G. Rothermel, A. Kinnear. Empirical Studies of Test Case Prioritization in a JUnit Testing Environment. Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), p.113-124, November 02-05, 2004.

[6] C. Pacheco, S. Lahiri, M. Ernst, T. Ball. Feedback-directed Random Test Generation. Proceedings of the 29th International Conference on Software Engineering, p.75-84, May 20-26, 2007.

[7] C. Pacheco, S. Lahiri, T. Ball. Finding Errors in .NET with Feedback-Directed Random Testing. Proceedings of the ISSTA '08, p.87-95, July 20-24, 2008.

[8] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, B. Davia. The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing. In Proceedings of the 24th International Conference on Software Engineering, pages 230–240, May 2002.

[9] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, M. Harrold. Test-suite Augmentation for Evolving Software. Proceedings of the 23th IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, September, 2008.

[10] T. Xie. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. Proceedings of the ECOOP, p.380-403, July 2006.

[11] Java Simplified Encryption, <http://www.jasypt.org/>

[12] CSPoker, <http://code.google.com/p/cspoker/>