An Experimental Study on Flakiness and Fragility of Randoop Regression Test Suites

Samad Paydar, Aidin Azamnouri

An Experimental Study on Flakiness and Fragility of Randoop Regression Test Suites

Samad Paydar and Aidin Azamnouri

Ferdowsi University of Mashhad, Mashhad, Iran
s-paydar@um.ac.ir
aidin.noori@mail.um.ac.ir

Abstract. Randoop is a well-known tool that proposes a feedback-directed algorithm for automatic and random generation of unit tests for a given Java class. It automatically generates two test suites for the class under test: 1) an error-revealing test suite, and 2) a regression test suite. Despite successful experiences with applying Randoop on real world projects like Java Development Kit (JDK) which have led to creation of error-revealing tests and identification of real bugs, it has not been investigated in the literature how useful are the regression test suites generated by Randoop. In this paper, we have investigated flakiness and fragility of Randoop's regression tests during evolution of 5 open source Java projects with a total of 78 versions. The results demonstrate that the flakiness of the regression tests is not generally noticeable, since in our dataset, only 5% of the classes have at least one flaky regression tests. In addition, test fragility analysis reveals that in most versions of the projects under study, the regression tests generated by Randoop could be successfully executed on many of later versions. Actually, for 2 out of 5 projects in the experiments, the regression tests generated for each version could be successfully executed on all the later versions of the project.

Keywords: random testing · Randoop · fragility · flaky tests

1 Introduction

Randoop [16] is a well-known tool in the domain of random testing which employs a feedback-directed algorithm for automatically generating unit tests for Java programs. It takes a Java class as the class under test (CUT) and creates random sequences of method calls on the objects of that class. Further, by executing each generated method sequence, it decides whether the sequence is appropriate for being extended to generate longer sequences. During sequence generation, Randoop tries to execute sequences to check the CUT against a set of predefined contracts that every Java class is expected to be compatible with. Should a contract is violated, Randoop stores the corresponding method sequence as an error-revealing test. If a sequence does not violate any contract, it is stored as a regression test that has captured the current behavior of the system. Finally,

Randoop generates two test suites from these two types of test: 1) an error-revealing test suite, and 2) a regression test suite.

Based on our experience (including the experiments discussed in this paper) with running Randoop on well-known open-source projects, we have observed that usually the error-revealing test suite is empty for a CUT, since the default contracts considered by Randoop are very general and they are not violated by those projects which are implemented by professional programmers. Therefore, for a test practitioner, the main output of Randoop is usually the regression test suite that it generates. This test suite is aimed at revealing regressions during the evolution of the CUT. In other words, if the behavior of the CUT which is captured by Randoop regression tests is changed in the later versions, the corresponding regression tests are expected to fail. Then, the tester needs to analyze the code to see whether the new behavior is correct or it is the result of an error introduced in the new version.

It is interesting to investigate how effective is the generated regression test suite during the evolution of the CUT. In this paper, we discuss an experimental study which mainly focuses on flakiness and fragility of Randoop's regression tests generated for real-world Java programs. The results are expected to shed light on the required improvements on Randoop that can increase the potentials of its application in real-world projects.

There are some works in the literature that study flaky tests [9], i.e. the tests that their pass or fail result is not deterministic and hence, they fail in some executions and pass in some other executions. In addition, a fragile test is a test that is successfully executed on a version of the CUT, but it fails to execute, e.g. due to a compile error, on the successive version of the CUT. Studying the causes for a test to become fragile during the evolution of the software, and also the possible automated fixes, have set the stage for a line of research, specifically in GUI testing domain [2]. While studying the flaky tests and fragile tests has been considered in the literature, there is not yet an established definition for measuring the level of flakiness and fragility in a given test suite. Consequently, in this paper, we first provide a set of metrics for this purpose, and then we use them to analyze the flakiness and fragility of Randoop regression test suites for open-source projects.

The reason why we focus on test flakiness and test fragility lies in the fact that the more a regression test suite generated by Randoop contains fragile or flaky tests, the less successful it would be in serving its main purpose, which is assuring the quality of software during its evolution. A flaky test fails to capture a consistent behavior of the CUT, and hence it cannot be used as a reference to determine whether the behavior is changed during the evolution of the CUT. Moreover, a fragile test first needs to be analyzed and modified so that it can be successfully executed. Identifying flaky or fragile tests and performing possible fixes on them might take a lot of time and effort, and hence, should Randoop generates a large number of flaky or fragile tests, its applicability in real world projects becomes quite questionable. The main purpose of this paper is to investigate this issue.

The rest of the paper is organized as follows. Section 2 briefly reviews the related works on Randoop. In Section 3, the experimental study and its elements are discussed, followed by the analysis of the results and discussion of the findings of the experiments in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

In this section, we briefly review some works that are aimed at improving Randoop effectiveness in unit test generation. First and foremost, the low code coverage of the Randoop tests has attracted many researchers [10, 21, 23, 8]. Due to the fact that the method sequences are created randomly and without any background knowledge or human intervention, it is quite difficult for Randoop to provide the methods under test with the appropriate inputs so that various states of the objects are covered. As a result, different parts of the code that require specific inputs are not covered by Randoop tests.

To address this limitation, GRT [10] provides noticeable improvements over Randoop by employing a two-step analysis method. In the first step, static analysis is performed to collect the required information from the class under test. In the second step, a dynamic analysis is performed with regards to the feedback received from the execution of the method sequences in addition to the information collected in the first step that lead to making a good decision on choosing which sequences should be extended. In [8], the idea of mutating an object under test is employed for the purpose of improving code coverage of the tests.

Another limitation of Randoop is that it cannot properly generate sequences from useful methods; thus, as a solution in [22], Seeker is introduced which employs dynamic and static analysis to create more useful sequences.

Due to the importance of generating appropriate input arguments for method calls, in [23], the TestMiner tool is introduced which extracts literals from the source code of the tests and uses them to create the required input strings. In [12], reusing the test cases from the libraries of the software under test, which resulted in the better generation of test cases, were investigated. Another issue with Randoop is that it does not test the private methods of CUT. In this regard, the authors in [1] suggest using Java Reflection and having access to private fields, which can result in false positives, but better code coverage. This issue of code visibility is also considered in [11].

Regarding the flaky tests, i.e. the tests that their pass/fail behavior is non-deterministic, the authors in [9] have discussed an empirical analysis of flaky tests in real world projects. Their main goal has been to identify the root causes for test flakiness, determine how and when the flaky behavior is manifested, and also to describe the mechanisms that are usually used by developers to fix the flaky tests. The focus in [9] is on flakiness of manually-written tests, while in this paper, we specifically target the tests automatically generated by Randoop.

There are also some works that focus on repairing failed unit tests. For instance in [4], the ReAssert technique is introduced which uses both static and dynamic analysis to suggest a repair for a failed unit test. The suggested re-

pair is mainly in terms of modification in the assertion statements, for instance, replacing assertTrue with assertFalse, or replacing literal values in the assert statements. It is worth noting that ReAssert is not intended to repair those tests that cannot be compiled due to recent changes in the program under test. In other words, it does not modify the unit tests to eliminate compile errors, but just to make a failing test pass.

There is another line of works that have focused on evaluating automated unit test generation tools. For instance, in [15], the methodology and the results of the 6^{th} JUnit testing tool competition is discussed. In this competition, a total of 59 CUTs from 7 open-source Java projects are selected and four automated JUnit tests generation tools, i.e. EvoSuite [5], JTexpert [19], T3 [17] and Randoop are executed with different time budgets to generate test suites for these CUTs. Finally, the performance of the tools are evaluated in terms of structural code coverage metrics and through mutation analysis. In a similar work [3], Randoop is also compared with EvoSuite and 4 other test input generator tools, again in terms of code coverage, efficiency and mutation adequacy.

In another work [20], the authors have evaluated effectiveness of three automated test generation tools Randoop, EvoSuite and AgitarOne in terms of being able to detect real faults in the Defects4J dataset. They have also analyzed the flaky tests generated by these automated tools. The results have shown that on average, 21% of the Randoop tests were flaky, i.e. their pass/fail behavior is non-deterministic. In [7], the authors have discussed their experiences with deploying Infer, a static analysis tool, and Sapienz, a dynamic analysis tool, at Facebook. They have described the open problems that need to be considered by software testing researchers, one of which is the flakiness of tests. In this regard, the authors emphasize the highly stochastic behavior of the systems deployed in real-word situations, and propose that we need to "Assume all Tests Are Flaky". Proposing the theoretical discussions behind this idea, a set of research questions are provided on how to deal with flaky tests based on this assumption. In [18], the authors have discussed their experience with applying Randoop for automated test generation for GUI testing of an industrial project. It is mentioned that integrating Randoop with the build process for the purpose of regression testing has resulted in many false positives. In other words, due to the high rate of intended changes, most of the failing regression tests do not indicate a real bug, but an intended change. However, the paper does not provide quantitative analysis of this problem.

The review of the related work shows that while the limited coverage, readability, and other aforementioned factors regarding Randoop and other automated unit test generation tools is taken into consideration by many researchers, the flakiness and fragility of the Randoop regression tests during the evolution of a project is not considered. This paper seeks to conduct the first investigation in this regard.

3 Experimental Study

In this section, different elements of the experimental study of Randoop regression tests are described. First, the research questions are introduced and then, the preparation of the dataset used in the experiments is described. Finally, the experiment procedures and the evaluation metrics are presented.

3.1 Research Questions

The goal of the current study is to investigate how effective are Randoop regression tests in terms of being able to reveal potential regressions during software evolution. For this purpose, we have focused on evaluating flakiness and fragility of Randoop regression tests. However, a prerequisite for this assessment is to determine whether Randoop is able to create any regression test for the CUT. In other words, if Randoop fails to create any regression test for a large ratio of the classes in a program under test, then the effectiveness of the generated regression test suites is questionable since they might not cover an appropriate amount of the program's code base. As a result, we first seek to determine for what percentage of the input classes, Randoop has been able to create at least one regression test.

Next, we consider two types of problems affecting the effectiveness of the regression tests: 1) flakiness and 2) fragility of the tests. Assuming that a regression test T is created over version i of the program under test, T is flaky if the result of executing T on the same version of the program, i.e. version i, is non-deterministic and hence varies over different executions [9, 14]. A flaky test is not useful from the point of view of regression tests, since it has not captured a stable behavior of the CUT, and hence, it is unable to judge about regressions in future versions of the CUT. In addition, T is a fragile test with regards to a successive version j, j>i, if it cannot be executed on version j of the program under test due to a compile error in T. The more flaky or fragile tests exist in the test suite generated by Randoop, the less is effectiveness of the test suite.

It is worth noting that flakiness is an inherent weakness of a regression test, since when a regression test which is created on version i is failed on the same version, it has not been able to correctly and consistently capture the behavior of that version of the CUT. Therefore, addressing the test flakiness issue requires improving the Randoop algorithm details to prevent generation of the flaky tests. Test fragility, on the other hand, is not necessarily rooted in the weakness of Randoop or the regression tests it generates, since it is caused by the changes made in the successive versions of the program under test. For instance, if Randoop has created a regression test for class C_1 in version i of the program under test, and this class is renamed in version j, j>i, then the corresponding test will fail to compile on version j and hence it becomes a fragile test with regard to this version. However, this cannot be considered as the weakness of that test. Actually, it depends on how we define a change in the behavior of the program under test. If we consider renaming of a class as a change in the behavior of the program under test, it can be argued that having the test failed is exactly what

we expect, since the regression test is expected to fail to reveal the change in the behavior of the program under test. If we exclude this kind of change from the definition of behavior change, then the test is not expected to fail. Regardless of which argument a test is in favor of, it is more appealing if it was possible to repair the test so that it can be compiled and executed to see whether it passes or fails. For instance, if it is possible to make the test executable just by renaming the corresponding class in the test code, it is interesting to keep the test in the regression test. However, this requires analyzing the source code to see what is the reason for the compile error and what changes are required to eliminate the error so that the test is compiled successfully. Apparently, this is not an easy task and it might be quite challenging and time-consuming, specifically for a large test suite. This increases the cost of using Randoop for practitioners, and hence, this is why we consider test fragility to indirectly reduce the effectiveness of the Randoop's regression tests.

Based on the viewpoint described above, the main research question in this research is:

RQ. How useful are the regression tests generated by Randoop? To answer this question, the following specific research questions are considered:

RQ_1. For what percentage of the classes under test, Randoop is able to generate any regression test?

RQ_2. What percentage of the Randoop regression tests are flaky?

RQ_3. What percentage of the Randoop regression tests generated for a version i of a CUT can be executed on the version i+1 of that CUT?

RQ_4. How long does a Randoop regression test last as a non-fragile test during the evolution of the program under test? In other words, what is the maximum value of k-j so that the regression tests generated over version j can successfully execute over version k, k>j?

In this paper, we have conducted an experimental study to answer these questions using a dataset of real world open source projects. While the results of the experiments are not meant to be applicable to every project, we believe they can provide a general understanding of the effectiveness of Randoop over similar projects.

3.2 Dataset

For the purpose of the experimental evaluations, first, a dataset is prepared including different versions of five Java open source projects from the Apache Commons family. Table 1 shows the basic information about the selected projects. We have selected these projects since they are well-known real-world projects, each having released more than 10 version. Actually, on average, about 16 versions have been available for each project and the size of each project, in terms of the number of classes, is increased by a factor of 10 from its first version to its last version. Hence, while the number of projects included in the data set is small, but the volume of the changes in these projects is noticeable and they are good candidates for representing the concept of evolution in a real-world project.

In addition, to create a dataset of Randoop regression tests, for every public class in each version of each project, we have executed Randoop with a time limit of 10 seconds and the resulting regression tests are stored in the dataset. It is worth noting that the default time limit of Randoop for test generation for a single class is 100 seconds, however due to the large number of classes in the dataset, we have used a smaller time limit to keep the execution cost of the experiments reasonable. Further, we have set *testsperfile* parameter of Randoop to 1, so that each regression test is created as a separate Java file declaring a test class with a single test method. The information about the generated regression tests is shown in Table 1.

Table 1. Dataset used in the experiments

		Versions			**Number of Classes with Test**			
Project	**Project Name**	**First**	**Last**	**Count**	**Min**	**Max**	**Total**	**Avg. Test per Class**
P_1	BeanUtils	1.0	1.9.3	19	3	78	1025	338
P_2	Codec	1.1	1.11	11	9	49	336	278
P_3	Collections	1.0	4.4.1	11	21	220	1264	190
P_4	Compress	1.0	1.16.1	20	36	126	1709	294
P_5	Digester	1.0	3.3.2	17	11	67	599	353

3.3 Experiments

In order to answer the first research question, it is needed to analyze the regression tests generated by Randoop and identify the cases where it has failed to generate any test for a given class. As for answering the next research questions, i.e. RQ_2 to RQ_4, we designed two experiments:

1. Flakiness Experiment. Since a flaky test has different behaviors in different executions, in order to determine the flaky tests, we have executed every regression test generated for each version i of each project, on the same version i of the same project until whether the test is failed or it is executed for 10 times. If the test is failed in one of its executions, it is considered to be a flaky test. This experiment is designed for answering RQ_2. Actually, repeating a test for 10 times is not guaranteed to reveal its flakiness, however, to control the execution cost of the experiment, we have used the value of 10 as a reasonable threshold, since this strategy of 10 reruns is common in practice [13, 6].
2. Fragility Experiment. In this experiment, we have executed every regression test generated on each version j of each project, on all the versions k (k>j) of the same project. However, we have ignored the flaky tests identified in the previous step. This experiment is considered to answer RQ_3 and RQ_4.

3.4 Metrics

In order to analyze the results of the experiments, we have defined a set of metrics which are introduced below.

- *TestGenSuccess* of Randoop on a specific version of a project P is the percentage of the classes in that version for which Randoop has successfully generated at least one regression test.
- *Flakiness* of a class is the percentage of the tests generated for that class that are flaky.
- $Fragility_{j,k}$ of a class is the percentage of the non-flaky tests generated for a class in version j of the corresponding project which are fragile with respect to version k, k>j.
- *Fragility-Free Length* of a project is the maximum value of k-j where the regression tests generated for version j can be executed on version k. A great value for this metric points to a long period in the evolution of the project during which the regression tests of older versions have no fragility with regards to later versions.

4 Result Analysis

In this section, we analyze and discuss the results of the experiments and answer the research questions described in the previous section.

4.1 Regression Test Suite Generation

In order to answer RQ_1, we have computed for each version of each project the percentage of the classes in that version for which Randoop has been successful in generating at least one regression test. The results are shown in Table 2. For instance, the results demonstrate that considering different versions of P_1, Randoop has been able to create regression tests for 94% to 100% of the CUTs. Furthermore, across all the versions, on average, Randoop has created regression tests for about 97% of the classes in P_1. The worst performance of Randoop is associated with P_3 where Randoop has created regression tests for only 72% of the classes in version 3.0. Some sample classes with no regression tests are mentioned in Table 3. Finally, over all the projects, Randoop has created regression tests for an average of 95% of the classes under test. As a results, it is reasonable to answer RQ_1 by concluding that Randoop is powerful in creating regression tests for most of the classes under test.

It is interesting to analyze why Randoop has not been able to create regression tests for some of the classes. Our initial analysis demonstrates that we can attribute this issue to the inability of Randoop in preparing required arguments for calling the methods, including constructors, of the CUTst, since they require complex objects, not primitive values, as input parameter. Hence, Randoop has not been able to create any object from these class and call methods on those

Table 2. Test Generation Results

	TestGenSuccess for Different Versions (%)		
Project	**Min**	**Max**	**Avgerage**
P_1	94	100	97
P_2	97	100	99
P_3	72	100	85
P_4	97	100	99
P_5	85	100	95
Total	72	100	95

Table 3. Sample Classes with no Regression Tests

Project	**Version**	**Class Name**
P_1	1.4	org.apache.commons.beanutils.ResultSetIterator
P_2	1.10	org.apache.commons.codec.binary.BaseNCodecInputStream
P_3	3.0	org.apache.commons.collections.list.LazyList
P_4	1.10	org.apache.commons.compress.archivers.dump.DumpArchiveSummary
P_5	3-3.0	org.apache.commons.digester3.binder.CallMethodBuilder

objects. However, we admit that more precise analysis is required to identify any other possible cause for this problem.

Finally, it is interesting to mention that Randoop has created a non-empty Error Test Suite only for 89 classes[1], counting for about 1% of the classes in the dataset. In addition, the average number of error tests generated for these classes is 27. This supports our previous claim that from a practical point of view, Randoop rarely generates any error-revealing for the class under test.

4.2 Flakiness Analysis

Next, we have analyzed the results of the flakiness experiment by computing Flakiness for each CUT in each version of the projects in the dataset. The results, shown in Table 4, demonstrate that for different projects, between 3% to 9% of the classes have Flakiness>0. Specifically, for those classes in P_1 with Flakiness>0, the minimum, maximum and average Flakiness is respectively 1%, 79% and 11%. For other projects, the average flakiness of those classes with Flakiness>0 is greater, compared to P_1. Finally, across the entire dataset, 5% of the classes have Flakiness>0 and the average Flakiness of these classes is 54%. Based on these results, we can answer RQ_2 by concluding that for a low ratio of the CUTs Randoop generates any flaky tests, but for such classes, on average, about half of the generated tests are flaky. Some sample classes for which Randoop has generated flaky tests are introduced in Table 5.

[1] This includes 26 distinct classes, since some classes have error test suite in different versions.

Table 4. Results of the Flakiness Experiment for Classes with Flakiness > 0

				Flakiness		
Project	**Count**	**Ratio (%)**	**Count (Unique)**	**Min**	**Max**	**Avg.**
P_1	49	5	5	1	79	11
P_2	30	9	6	1	93	48
P_3	44	3	10	1	99	43
P_4	106	6	6	15	100	80
P_5	38	6	9	5	100	55
Total	267	5	36	1	100	54

We have not performed a detailed root causes analysis of the flaky tests. However, our initial investigation reveals that the way Randoop deals with side effect of modifying static members of the CUT needs to be improved. A good case in point is class *org.apache.commons.beanutils.ConvertUtils* in P_1, where during test generation, a method sequence modifies the static members defined in this class, e.g. *defaultDouble* and *defaultInteger*, and later a second method sequence reads the value of these members and uses them in the assertions. Later, when a test that is created from the second method sequence is executed, it is executed with no history of the changes that are performed by the first method sequence, and hence, the assertions fail. Listing 1.1 shows a sample regression test that is flaky due to this reason. It is worth mentioning that *test order dependency* is identified in [12] as the third most frequent cause of test flakiness, and *static field in CUT* is determined as one of the three identified sources of this dependency.

Table 5. Sample Classes with Flaky Tests

Project	**Version**	**Class Name**
P_1	1.0	org.apache.commons.beanutils.ConvertUtils
P_2	1.10	org.apache.commons.codec.digest.Crypt
P_3	1.0	org.apache.commons.collections.BeanMap
P_4	1.0	org.apache.commons.compress.archivers.zip.ZipArchiveEntry
P_5	2.1	org.apache.commons.digester.Digester

Listing 1.1. A sample flaky test generated by Randoop for version 1.0 of P_1

```
@Test
public void test1() throws Throwable {
    org.apache.commons.beanutils.ConvertUtils convertUtils0 = new
        ↪ org.apache.commons.beanutils.ConvertUtils();
    float f1 = convertUtils0.getDefaultFloat();
    org.junit.Assert.assertTrue(f1 == 10.0f);
}
```

4.3 Fragility Analysis

In order to analyze the results of the fragility experiment, we have first measured $Fragility_{j,k}$ for each class under test in a source version j with respect to all subsequent versions k, k>j. Next, we have computed the average value of $Fragility_{j,k}$ over all the classes in version j. The results are shown via the heat maps in Figure 1 to Figure 3. In these heat maps, a cell with a red color represents the maximum value among all the cells and a green cell shows the minimum value. For the purpose of brevity, the heat maps related to projects P_1 and P_4 are not shown since all their cells have a value of 0.

	Target Version (k)									
Source Version (j)	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11
1.1	11	11	11	17	17	17	17	17	17	17
1.2		0	0	4	4	4	4	4	4	4
1.3			0	6	6	6	6	6	6	6
1.4				3	3	3	3	3	3	3
1.5					0	0	0	0	0	0
1.6						0	0	0	0	0
1.7							0	0	0	0
1.8								0	0	0
1.9									0	0
1.10										0

Fig. 1. Average $Fragility_{j,k}$ for P_2

The results demonstrate that the average fragility of the regression tests generated for every version of P_1 with regards to every later version is 0%. This interesting observation means that if Randoop was used to generate regression tests for each version of P_1, those test would have been executable on every successive version of P_1 to control any change in the behavior during the evolution of the P_1. The same is true about P_4 where the average fragility of the regression tests of each version with regards to successive versions is 0%. For P_3, the average fragility of the regression test of version 1.1 with regard to version 1.2 is 11%, meaning that 11% of the tests generated for version 1.1 cannot be compiled and hence executed on version 1.2. Further, 17% of these tests cannot be compiled on version 1.5 and later versions. However, the tests that are generated for version 1.5 and later, have no fragility with regard to all their successive versions. For P_3, the tests generated for versions before 4.4.0, all have a noticeably high fragility with version 4.4.0 and later versions. This is a symptom of a noticeable change in version 4.4.0. Actually, our investigation reveals that this is due to renaming the main package of P_3 from *collections* to *collections4*. This has made all the tests generated for previous versions fail to compile. For P_5, the results are similar to P_3, and the regression tests generated for versions before 3.3.0 have almost complete fragility with regard to the version 3.3.0 and later versions. Similar to P_3, this can be attributed to the fact that the main package of P_5 is renamed from *digester* to *digester3*.

Finally, among 18 versions of P_1 (the last version is not considered since it has no successive version), all have the characteristic that their regression tests have no fragility on their immediate successive version. For P_2, P_3, P_4 and P_5, this is respectively 8 out of 10, 7 out of 10, 19 out of 19, and 14 out of 16.

Based on the discussion above, we can answer RQ_3 by saying that fragility of the regression test of a version is usually 0% or low with regard to at-least a few successive versions. Consequently, we can conclude that if Randoop is being used for regression testing during the evolution of a project, it could be of great help in controlling regressions. However, as mentioned for P_3 and P_5, in some points during the evolution of the project, the previous regression tests might become fragile due to major changes introduced in a new version. What is needed in that situation, is an effective technique for automatically performing the possible repairs on the regression tests so that they can be compiled on the new versions. We believe development of such a repair technique is both feasible and valuable in improving the effectiveness of Randoop regression tests.

	Target Version (k)									
Source Version (j)	2.0	2.1	2.1.1	3.0	3.1	3.2	3.2.1	3.2.2	4-4.0	4-4.1
1.0	6	6	6	40	40	40	40	40	95	95
2.0		0	0	36	36	36	36	36	97	97
2.1			0	21	21	21	21	21	98	98
2.1.1				24	21	21	21	21	98	98
3.0					0	0	0	0	99	99
3.1						0	0	0	99	99
3.2							0	0	99	99
3.2.1								0	99	99
3.2.2									99	99
4-4.0										0

Fig. 2. Average $\text{Fragility}_{j,k}$ for P_3

It is interesting to identify the cause of the fragility of the tests. Through analysis of the compilation results of the tests, we have identified the top-5 errors most frequently raised by the compiler during the fragility experiment. The results are shown in Table 6. In this table, the compiler errors are abstracted by replacing the project-specific identifiers, e.g. class name or package names. While this requires thorough investigation to identify the types of changes that have led to these compiler errors, our initial analysis have demonstrated that changing package names, moving classes to new packages and changing access level of the class members (e.g. from public to private) are among the most frequent changes that have caused compiler errors and test fragility. For instance, in version 3.0 of P_3, the class *FilterIterator* is moved from *org.apache.commons.collections* package to *org.apache.commons.collections.iterators* package. This change makes all the regression tests that are created for previous versions and use *FilterIterator*

Source Version (j)	Target Version (k)															
	1.1	1.1.1	1.2	1.3	1.4	1.4.1	1.5	1.6	1.7	1.8	1.8.1	2.0	2.1	3.3.0	3.3.1	3.3.2
1.0	0	0	0	0	0	0	0	36	36	36	36	36	36	100	100	100
1.1		0	0	0	0	0	0	29	29	29	29	29	29	100	100	100
1.1.1			0	0	0	0	0	29	29	29	29	29	29	100	100	100
1.2				0	0	0	0	19	19	19	19	19	19	100	100	100
1.3					0	0	0	19	19	19	19	19	19	100	100	100
1.4						0	0	17	17	17	17	17	17	100	100	100
1.4.1							0	19	19	19	19	19	19	100	100	100
1.5								20	20	20	20	20	20	95	95	95
1.6									0	0	0	0	0	98	98	98
1.7										0	0	0	0	98	98	98
1.8											0	0	0	98	98	98
1.8.1												0	0	98	98	98
2.0													0	98	98	98
2.1														99	99	99
3.3.0															0	0
3.3.1																0

Fig. 3. Average $\text{Fragility}_{j,k}$ for P_5

to fail to compile on version 3.0. We believe it is promising to seek to develop new techniques for automatically repairing the regression tests to cope with these changes. This is the main direction of our future work. To answer RQ_4, it is required to compute Fragility-Free Length for each project. This can be achieved by identifying the length of the longest sequence of zeros in the rows of Figure 1 to Figure 3. The Fragility-Free Length for each project is shown in Table 7. The results emphasize the effectiveness of Randoop regression tests since it demonstrates that the regression tests that are generated for the versions in which the fragility-free period starts, could have been used to perform regression testing on an interesting number of later versions.

Table 6. Top-5 compiler errors for fragile tests

Compiler Error Template
cannot find symbol {identifier}
incompatible types: {type1} cannot be converted to {type2}
package {package} does not exist
reference to {identifier} is ambiguous
{class member} has private access in {class}

Table 7. Results of Fragility Experiment

Project	P_1	P_2	P_3	P_4	P_5
Fragility-Free Length	18	6	4	19	7

5 Conclusion

In this paper, we have discussed an experimental evaluation of the effectiveness of the regression tests generated by Randoop. Specifically, we have investigated flakiness and fragility of Randoop's regression tests during evolution of 5 open source Java projects with a total of 78 versions. The results demonstrate that the flakiness of the regression tests is not generally noticeable, since in our dataset, only 5% of the classes have at least one flaky regression tests. In addition, test fragility analysis reveals that in most versions of the projects under study, if Randoop has been used to generate regression tests, those tests could be successfully executed on a noticeable number of later versions. Actually, for 2 out of 5 projects that are used in the experiments, the regression tests generated for each version could be successfully executed on all the later versions of the project. For some of the projects, there are some points during the evolution of the project in which the previous regression tests become fragile. We believe that it is possible to develop repair algorithms to automatically do the required modifications on some of the fragile tests to eliminate their fragility. Our future work is mainly focused on development of such a repair technique.

References

1. Arcuri, A., Fraser, G., Just, R.: Private API access and functional mocking in automated unit test generation. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017. pp. 126–137 (2017)
2. Coppola, R., Morisio, M., Torchiano, M.: Mobile GUI testing fragility: A study on open-source android applications. IEEE Trans. Reliability **68**(1), 67–90 (2019). https://doi.org/10.1109/TR.2018.2869227, https://doi.org/10.1109/TR.2018.2869227
3. Cseppento, L., Micskei, Z.: Evaluating code-based test input generator tools. Softw. Test., Verif. Reliab. **27**(6) (2017)
4. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: Reassert: Suggesting repairs for broken unit tests. In: ASE. pp. 433–444. IEEE Computer Society (2009)
5. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and 13th European Software Engineering Conference (ESEC-13), Hungary, September 5-9, 2011. pp. 416–419 (2011)
6. Gupta, P., Ivey, M., Penix, J.: Testing at the speed and scale of google. Google Engineering Tools Blog (2011)
7. Harman, M., O'Hearn, P.W.: From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: SCAM. pp. 1–23. IEEE Computer Society (2018)

8. Jaygarl, H., Kim, S., Xie, T., Chang, C.K.: OCAT: object capture-based automated testing. In: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Italy, July 12-16, 2010. pp. 159–170 (2010)
9. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 643–653 (2014)
10. Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J., Ramler, R.: GRT: program-analysis-guided random testing (T). In: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. pp. 212–223 (2015)
11. Ma, L., Zhang, C., Yu, B., Sato, H.: An empirical study on the effects of code visibility on program testability. Software Quality Journal **25**(3), 951–978 (2017)
12. Ma, L., Zhang, C., Yu, B., Zhao, J.: Retrofitting automatic testing through library tests reusing. In: 24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016. pp. 1–4 (2016)
13. Micco, J.: Continuous integration at google scale, 2013 (2013)
14. Micco, J.: Flaky tests at google and how we mitigate them (2016)
15. Molina, U.R., Kifetew, F.M., Panichella, A.: Java unit testing tool competition: sixth round. In: Proceedings of the 11th International Workshop on Search-Based Software Testing, ICSE 2018, Sweden, May 28-29, 2018. pp. 22–29 (2018)
16. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. pp. 815–816. ACM (2007)
17. Prasetya, I.S.W.B.: T3i: a tool for generating and querying test suites for java. In: ESEC/SIGSOFT FSE. pp. 950–953. ACM (2015)
18. Ramler, R., Klammer, C., Buchgeher, G.: Applying automated test case generation in industry: A retrospective. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018. pp. 364–369 (2018)
19. Sakti, A., Pesant, G., Guéhéneuc, Y.: Jtexpert at the fourth unit testing tool competition. In: SBST@ICSE. pp. 37–40. ACM (2016)
20. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T). In: ASE. pp. 201–211. IEEE Computer Society (2015)
21. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Mseqgen: object-oriented unit-test generation via mining source code. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009. pp. 193–202 (2009)
22. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Su, Z.: Synthesizing method sequences for high-coverage testing. In: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, Portland, OR, USA, October 22 - 27, 2011. pp. 189–206 (2011)
23. Toffola, L.D., Staicu, C., Pradel, M.: Saying 'hi!' is not enough: mining inputs for effective test generation. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. pp. 44–49 (2017)