ReTest: Framework for Applying TDD in the Development of Non-deterministic Algorithms

André A.S. Ivo[1(✉)] and Eduardo M. Guerra[2]

[1] Centro Nacional de Monitoramento e Alertas de Desastres Naturais (CEMADEN), São José dos Campos, SP, Brazil
andre.ivo@cemaden.gov.br

[2] Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil
eduardo.guerra@inpe.br

Abstract. TDD is a technique traditionally applied in applications with deterministic algorithms, when you have a known input and an expected result. Therefore, the challenge is to implement this technique in applications with non-deterministic algorithms, specifically when several random choices need to be made during its execution. The purpose of this paper is to present the ReTest framework, a JUnit extension, that allows an extension of the TDD technique, to enable its use for the development of non-deterministic algorithms.

Keywords: TDD · Non-determinism · Tests · Framework · JUnit · Metadata · Code annotations

1 Introduction

TDD (Test-Driven Development) is a software development technique in which tests are developed before code in short and incremental cycles [1]. The technique proposes for the developer to create a new flawed test, and then, to implement a little piece of code, in order to satisfy the current test set. Then, the code is refactored if necessary, to provide a better structure and architecture for the current solution [2,3].

TDD is traditionally applied in applications with deterministic algorithms, when there is a known input and one expected result. The challenge becomes, the use of TDD in applications with non-deterministic algorithms, where from executions with the same input it is possible to obtain different valid results. This type of approach usually uses several calls to functions that generates pseudo-random numbers during the algorithm execution in order to represent random decisions. Although it is not possible to know exactly what the output will be, it is usually possible to check whether the output received is considered valid or not. This scenario is very common in the development of scientific software [4].

The following factors make it difficult to develop non-deterministic software using TDD: (a) the result of the same execution may be different for the same

T. Silva da Silva et al. (Eds.): WBMA 2016, CCIS 680, pp. 72–84, 2017.
DOI: 10.1007/978-3-319-55907-0_7

inputs, which makes it difficult to compare with a return value; (b) obtaining a valid return for a test case execution does not mean that valid return will be returned on the next executions; (c) there are may be several random decisions and a variable number of such of decisions, making not viable the creation of Mock Objects [5,6] that return fixed results for these decisions; and (d) it is difficult to execute a previous failed test with the same random decisions made in its last execution.

The goal of this paper is to present an extension for the JUnit framework called ReTest, developed by the authors of this work, which allows an extension of TDD to enable its application for algorithms with non-deterministic characteristics. The main feature of ReTest is to allow a test case that receives a class responsible to generate pseudo-random numbers to be executed several times with different seeds, increasing the test coverage. From the result of these repetitions, the framework stores the seeds that generated failures and uses them in future tests, ensuring that a scenario where an error was detected in the past is executed again.

The paper is organized as follows: Sects. 2 and 3 give a brief introduction to TDD and JUnit; Sect. 4 presents the ReTest framework; Sect. 5 describes the use of ReTest in the context of TDD technique; and, finally, the conclusion and proposals for future work are presented in Sect. 6.

2 Test Driven Development (TDD)

TDD is a code development and design technique, in which the test code is created before the production code. There are several research reported by Guerra and Aniche (see [1]) that indicates that the use of TDD can improve the source code quality. One of the reasons for the popularization of TDD is its explicit mention as part of the agile methodology Extreme Programming (XP) [7], however today is widely use out of its context.

In TDD practice, the developer chooses a requirement to determine the focus of the tests, then writes a test case that defines how that requirement should work from the class client point of view. Because this requirement has not yet been implemented, the new test is expected to fail.

The next step is to write the smaller amount of code as possible to implement the new requirement verified by the test. At this point, the added test, as well as all other previously existing tests, is expected to run successfully. Once you have passed the tests, the code must be refactored so that its internal structure can be continuously evolved and improved. The tests help to verify that the behavior has not been modified during refactoring.

This cycle is performed repeatedly until the tests added verify scenarios for all expected requirements of the class. The TDD cycle is presented in Fig. 1 [2,3].

With the use of TDD, the design of the code is defined in cycles. The idea is that with each new test added, create a small increment of functionality compared to the previous ones. TDD technique is described in several books, such as “Test-Driven Development by Example”, “Agile Software Development, Principles, Patterns, and Practices”, “Growing Object-Oriented Software, Guided

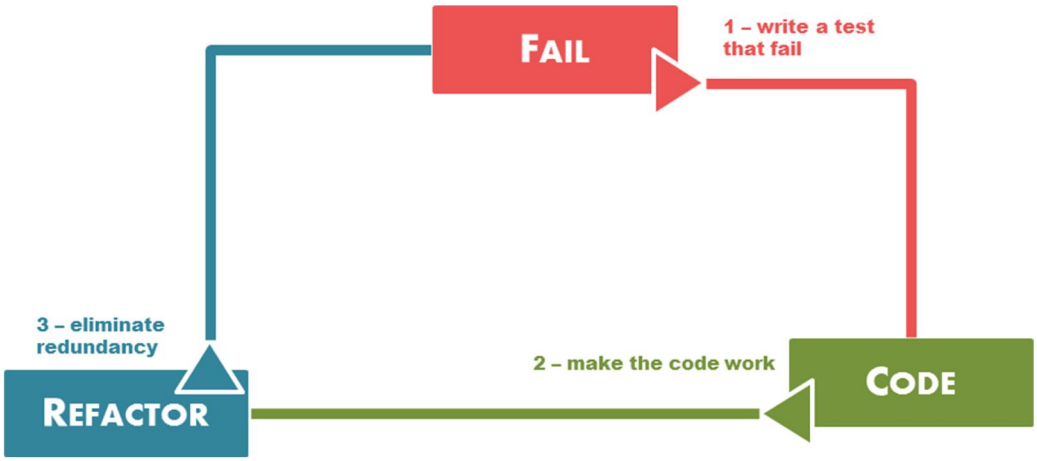

Fig. 1. TDD execution diagram

by Tests" and "Test-Driven Development: A Practical Guide" (see [2,3,8,9]), besides being widely used in industry.

3 JUnit Framework and Its Extension Points

JUnit is an open-source framework, created and developed by Erich Gamma and Kent Beck, for the creation of unit tests in the Java language. Its purpose is to be a basis for the creation of test automation code. It is widely used for the practice of TDD and its same model was used in the creation of test frameworks for other languages, being these frameworks referenced in general as XUnit. Some main features of such frameworks are the execution of test cases and the display of execution results [10].

JUnit, since version 4, provides extension points that allow the introduction of new functionality. Some of the most important JUnit extension points are represented by the classes Runner and Rule.

Runner is the class responsible for running the test methods from a test class. When a simple test class is executed with JUnit 4, it uses the class BlockJUnit4ClassRunner.class as the default runner. The Runner class hierarchy is represented in the diagram in Fig. 2.

In this way, to implement a Runner just create a new class and extend the Runner class shown in the diagram in Fig. 2.

To use just create a test project, and in the tests class include the annotation *@RunWith* and as argument pass the new class Runner.

This will replace all known JUnit 4 behavior. If you want to maintain the behavior, simply create a new Runner that extends the *BlockJUnit4ClassRunner* class.

Another extension point is known as Rule that other than Runner adds new behaviors mainly before and after the execution of each test. To write our own Rule, just create a class that implements the *TestRule* interface.

To use, just declare a public attribute in the test class and annotate it with *@Rule*, as shown in Code Snippet 1.

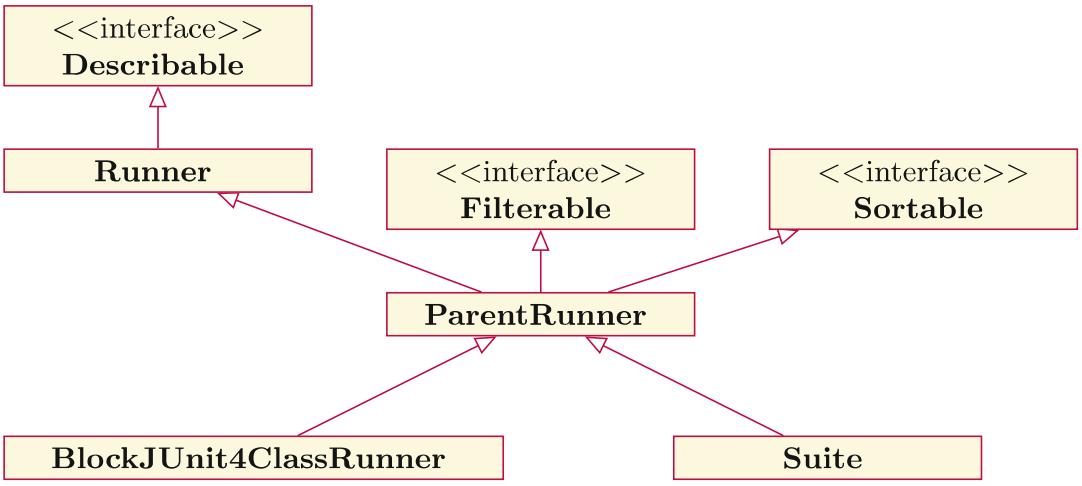

Fig. 2. Core class diagram of ReTest framework

Code Snippet 1. @Rule use example

```
public class TestClass {

  @Rule
  public NewRule newRule = new NewRule();

  public void testMethod(@RandomParam Random random){
    Object result = nonDeterministicAlgorithm(random);
    assertResult(result);
  }
}
```

In the example shown in Code Snippet 1, when executing the test project, who should call the *testMethod()* is newRule, responsible for adding the desired behaviors before and after the tests.

4 ReTest: Test Framework for Non-deterministic Algorithms

The ReTest framework, Random Engagement for Test, aims to extend JUnit to provide a framework for testing non-deterministic algorithms. It provides to its users a mechanism for managing the seeds used to generate random data in the algorithm being tested. Consequently, the same test can be repeated and the seeds used in failed runs can be repeated. These features facilitates the application of TDD for the development of non-deterministic algorithms. The ReTest framework is open-source and can be found at https://github.com/andreivo/retest.

4.1 Overview

To use ReTest the developer needs to create a test project using JUnit 4, and include the *@RunWith* annotation with *ReTestRunner.class* argument in the test class.

In the test methods the developer needs to include annotations to configure how it should be executed and annotations in the parameters that need to receive values generated and managed by the framework. The framework managed parameters are meat to be used as input data for the tests. The Code Snippet 2 shows a simple example of use.

Code Snippet 2. Simple example of how to use ReTest

```
@RunWith(ReTestRunner.class)
public class TestClass {

  @Test
  @ReTest(10)
  @SaveBrokenTestDataFiles(filePath = "/data/file1.csv")
  @LoadTestFromDataFiles(filePath = "/data/file1.csv")
  public void testMethod(@RandomParam Random random){
    Object result = nonDeterministicAlgorithm(random);
    assertResult(result);
  }
}
```

In the code shown in Code Snippet 2, the test method is marked with the *@ReTest(10)* annotation, which configures the framework to execute it 10 times. At each execution, the framework will initialize the parameter marked with *@RandomParam* received by the test method with a different seed. Notice that this object is passed as an argument to the method being tested, called *nonDeterministicAlgorithm()*. The class *Random* is used internally by the test method for the generation of its random numbers and, consequently, as a basis for its non-deterministic decisions. The *assertResult()* method used checks whether the return of the algorithm is considered valid. This test will be executed multiple times with Random initialized with different seeds, simplifying the execution of a large number of scenarios.

The seeds used in failed tests will be stored in the file "data/file1.csv", because the test method is marked with the *@SaveBrokenTestDataFiles* annotation. When executed again, in addition to the 10 repetitions configured by the *@ReTest* annotation, the test method will also run with the seeds stored in the "data/file1.csv" file, which is configured by the *@LoadTestFromDataFiles* annotation. That way, by running the failed tests again, you can check that the error has been corrected in addition to maintaining a set of regression tests.

Since in TDD the tests are executed frequently, throughout the development process the test executions should achieve good code coverage. This is reinforced by the fact that the tests that have failed previously are always executed again, creating data for regression tests.

4.2 Features

The ReTest framework has an API that allows you to:

(a) generate randomic data to be applied to the tests;
(b) create custom randomizers for data in the application domain;
(c) save the data from failed tests;
(d) save test data that has been successfully executed;
(e) save the return of the test method to generate a set of data based on random inputs and expected outputs;
(f) load test data from external files or sources;
(g) create custom mechanisms for handling external sources, both for saving and loading test data.

4.3 ReTest Annotation Set

In addition to the common JUnit annotations, the ReTest framework has a set of 4 annotations for the test methods and 4 annotations for the method parameters.

The annotations for the methods are:

(a) **@ReTest:** This annotation is responsible for performing the test repetition. In this annotation it is possible to indicate how many times the test method should be executed;
(b) **@SaveBrokenTestDataFiles:** When you mark a method with this annotation, the input data will be saved to the file when the test fails;
(c) **@SaveSuccessTestDataFiles:** When you mark a method with this annotation, the input data will be saved to file when the test is successful;
(d) **@LoadTestFromDataFiles:** When you mark a method with this annotation, the input data from this file will be loaded and used in the execution.

The annotations for the method parameters are:

(a) **@IntegerParam:** Annotation indicates that the ReTest framework should pass as a parameter a random integer;
(b) **@RandomParam:** This annotation indicates that the framework should pass an instance of an object of type Random, with a known seed, so that it can be stored and retrieved from files, making it possible to reconstruct the same test scenario;
(c) **@SecureRandomParam:** This annotation indicates that the framework should pass an instance of an object of type SecureRandom, with a known seed, so that it can be stored and retrieved from files, making it possible to reconstruct the same test scenario;
(d) **@Param:** This annotation allows to indicate custom randomizers for the specific data types in the application domain, allowing the extension of the framework for random generation of several types of data.

4.4 Internal Architecture and Extension Points

This framework is based on the implementation of a new Runner, which reads and interprets the annotations presented in the session Sect. 4.3. The Fig. 3 shows the class diagram of the *ReTestRunner* implementation. In this diagram it is possible to observe the first extension point of the framework for personalization of the format of the data files, in the form of the implementation of the abstract class *TestDataFiles*. To configure the newly created class, it should be passed as a parameter to the *@SaveBrokenTestDataFiles*, *@SaveSuccessTestDataFiles*, and *@LoadTestFromDataFiles* annotations.

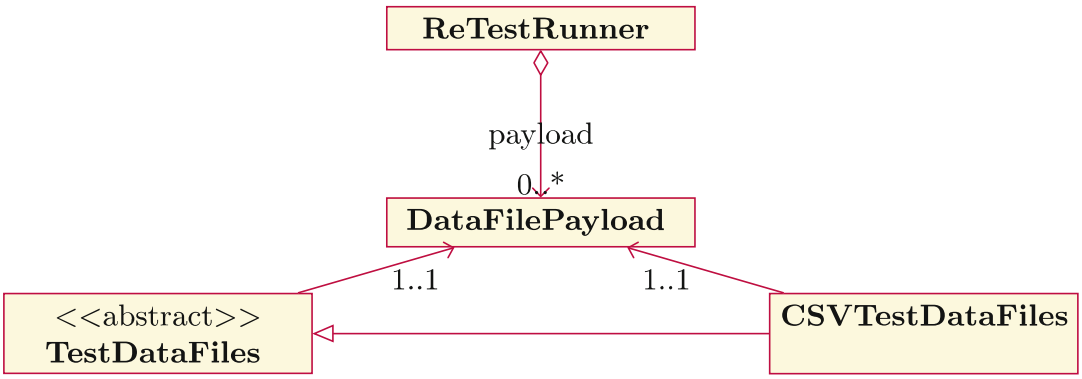

Fig. 3. Core class diagram of ReTest framework

The Fig. 4 shows the existing randomizers used to introduce parameters with random values in the test methods. At this point it is possible to observe the second extension point of the framework, in the form of the implementation of the abstract class *DataType*. To configure the new class created as the data generator for a test, it should be configure as an attribute of the *@Param* annotation.

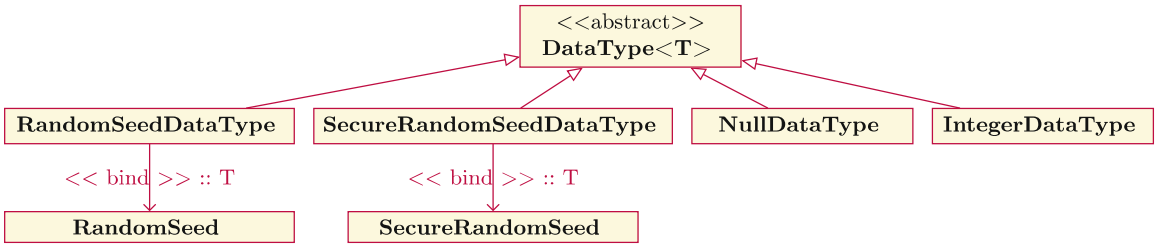

Fig. 4. Class diagram of randomized objects

5 TDD with ReTest

Because to the difficulties presented in the introduction of this article, TDD is not a technique normally used in the development of non-deterministic algorithms.

One of the goals of the ReTest framework is to make the use of this technique feasible for these scenarios.

From the use of ReTest is possible complement the development cycle of TDD as observed in Fig. 5. The steps of this new cycle consist of:

1. Create a new test that fails in at least one of its executions;
2. Store information of the failed scenarios to enable the verification if the changes in the production code make the failed scenario to pass;
3. Develop the simplest solution that makes the test suite run successfully for all inputs;
4. Run the test cases several times including new random generators with new seeds and with seeds that falied in previous test executions;
5. Refactor, if necessary, to provide a better internal structure for the final solution;

In this cycle, the steps of the original TDD are included, presented in Sect. 2. New steps were added as extensions proposed by the use of the ReTest framework, in order to ensure that TDD can be used as an application design technique and as a regression testing tool for non-deterministic algorithms.

To illustrate the use of this TDD cycle, consider the creation of a method to generate an array of "n" positions, with random numbers varying between 10 and −10, whose total sum of its elements is zero. This method receives as input parameter a Random object (used by the method to generate random numbers) and the size of the array to be generated.

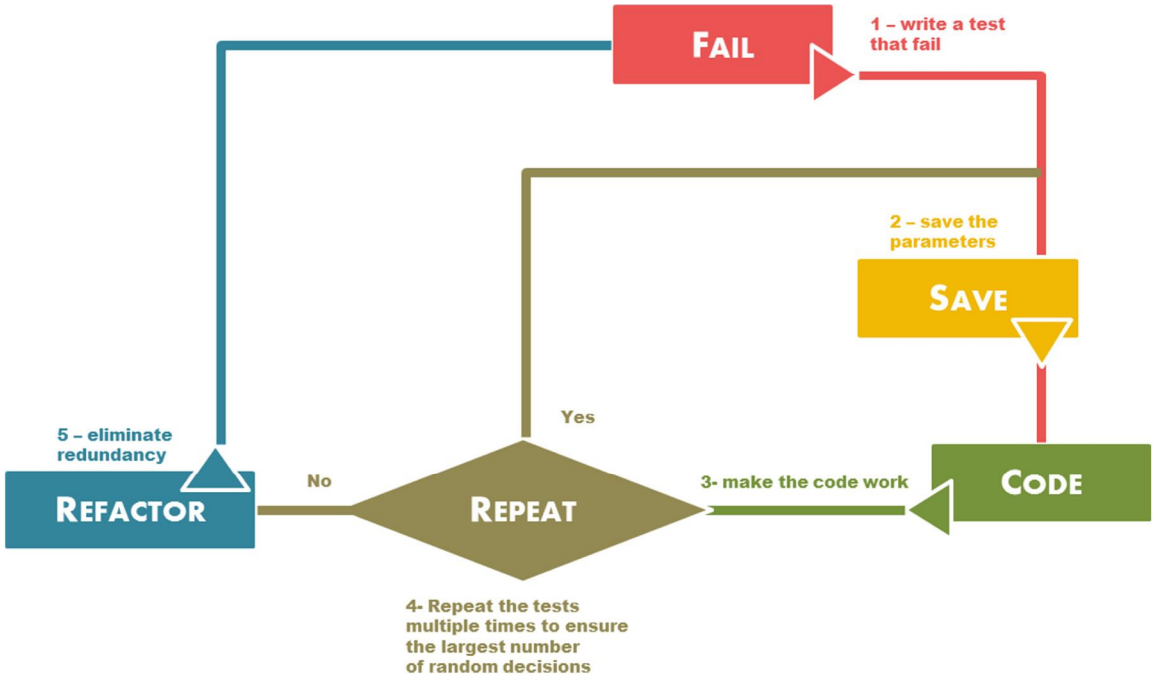

Fig. 5. Adaptation of TDD to ReTest

The following items describe the steps used to develop this function using TDD. Due to space limitations, the code for each of the steps will not be displayed and refactoring steps will be omitted.

(a) The first test asks the method to create an array with size 1. Since there is only one valid response for this case, which is 0, it is not necessary to use any ReTest annotations;
(b) It is written as the method implementation the return of a fixed value, and the test is executed successfully;
(c) The second test introduced invoke the method passing the parameter to create a size 2 array, initially checking only if the response has the appropriate array size. At first moment the test fails, because of the method in returning an array of size 1;
(d) As an initial implementation, an array of the size passed as a parameter is created and a random value generated within the range of −10 to 10 is set for each position;
(e) When executed, the tests pass, but it is known that the validity of the response is not being verified correctly;
(f) An auxiliary assertion method is then created to check the validity of the output according to the requirements. This method checks if the array has the expected size, if the value of each element is within range of −10 to 10, and if the sum of the elements is equal to zero, as shown in Code Snippet 3;

Code Snippet 3. Method for evaluating rules

```
private void assertElements(int[] arr, int arraySize) {
  int result = 0;
  //verify if all
  for (int i = 0; i < arraySize; i++) {
    assertTrue(arr[i] >= -10 && arr[i] <= 10);
    result = result + arr[i];
  }

  //verify the sum
  assertEquals(0, result);
}
```

(g) The test code for n = 2 is then modified so that it uses the assertion method created. The *@ReTest* annotation is used for this test method to configure the framework to execute it 10 times. The Fig. 6 shows the result of the test execution. Note that in 3 out of 10 scenarios the test runs successfully. As it is known that the implementation has not yet been performed, therefore the information about the failed test should not be saved yet;
(h) The code is changed so that the last array value is not randomly generated, but is the value that makes the sum to be equals to zero. The tests are run and now all pass successfully;
(i) The test is then annotated with *@SaveBrokenTestDataFiles* and *@LoadTestFromDataFiles* so that, from this point, that information of failed tests are

stored and executed again, as can be seen in Code Snippet 4; From this point the test code for other methods is similar to this one, varying only the parameter "n" passed to the function *generateArrayWithSumZero()*;

Code Snippet 4. Example of test method

```
@Test
@ReTest(10)
@SaveBrokenTestDataFiles(filePath = "/tmp/dataTest.csv")
@LoadTestFromDataFiles(filePath = "/tmp/dataTest.csv")
public void test2(@RandomParam Random r) {
  int n = 2;
  int[] result = ArrayFactory.generateArrayWithSumZero(r, n);
  assertElements(result, n);
}
```

(j) The third test added uses as parameter n = 3, so that an array of size 3 is generated. This test already receives the *@ReTest* annotation to be repeated 10 times. When performing the tests, some of the repetitions fail, because in some cases this approach does not generate a valid response, as can be observed in Fig. 7;

(k) The TDD process follows by having all the test running in the 3-element array generation scenario, and then placing the annotations so that failed executions are stored and included in the regression tests;

(l) The process is repeated in the introduction of new tests with the parameter "n" assuming the values 10, 100 and 1000. Figure 8 shows the execution of the tests for an array with 1000 elements, after successive changes in the algorithm being developed;

From the example, it is possible to have a more concrete vision of how ReTest can be used to support the use of TDD in the development of a non-deterministic algorithm. Note that test cases are gradually being introduced and implementation is also occurring incrementally.

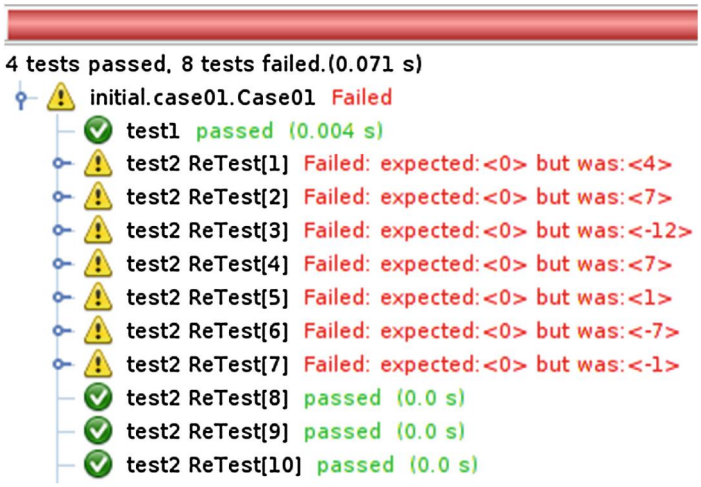

Fig. 6. Result of using ReTest for 2-position array

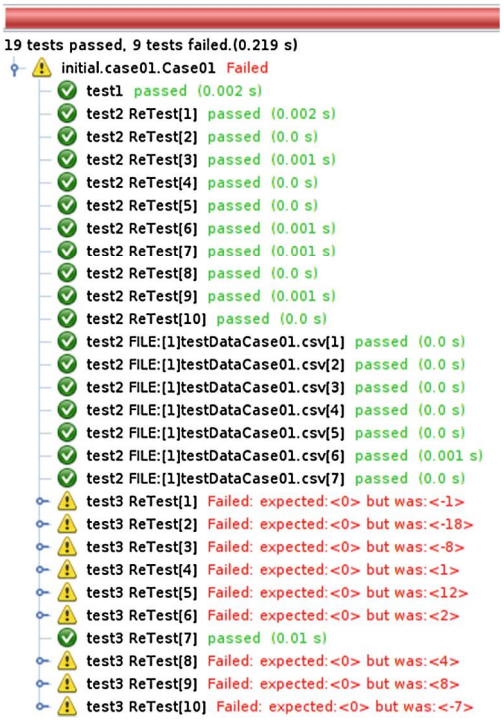

Fig. 7. Result of using ReTest for 3-position array with previous tests

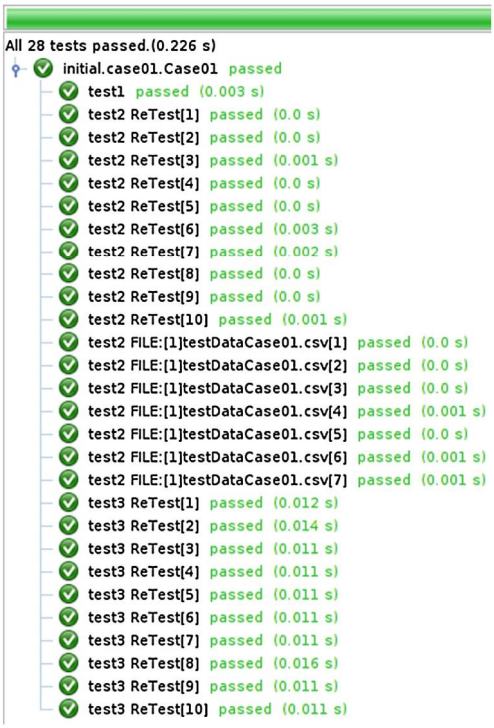

Fig. 8. Final result of the example with all tests running

The first point to emphasize is that when a test that needs to be repeated is executed, its execution is only considered correct when in all cases success is obtained. Note in Fig. 6, for example, that some executions always execute successfully, not because the implementation is correct, but because randomness leads to the correct solution in some cases. In this case, the repetition functionality of the framework is important because in each execution of the test suite it is possible to repeat the same test several times.

Another important point is in storing the seeds that generated failed test scenarios. Although it has not been commented, in the development of the example, in some cases modifications in code lead previous tests to fail in some scenarios. In this case, it was important to have the same test scenarios executing again to make sure that the problem was solved.

6 Conclusion

The goal of this work is to propose a test framework that facilitates the use of TDD for the development of non-deterministic algorithms. Some of the existing difficulties were to repeat exactly the same test cases flow that had failed previously and to be possible to have the test running successfully only in some executions. These difficulties are linked to the random decisions made during the execution of these algorithms.

The use of the ReTest framework makes it possible to use TDD for this type of algorithm, since it can repeat the same test several times and manage the seeds in order to repeat the failed test scenarios. The example presented in Sect. 5 showed how these functions can help us to follow the TDD flow to incrementally develop these algorithms.

As future work, we will evaluate the use of this framework for the development of a real non-deterministic algorithm using TDD. In addition, it is also intended to conduct an experiment with several developers to evaluate if they can use TDD in this way to develop such kind of algorithm.

References

1. Guerra, E., Aniche, M.: Achieving quality on software design through test-driven development. In: Mistrik, I., Soley, R., Ali, N., Grundy, J., Tekinerdogan, B. (eds.) Software Quality Assurance, pp. 201–220. Elsevier Inc., Amsterdam (2016)
2. Beck, K.: Test-Driven Development by Example. Addison-Wesley, Boston (2002)
3. Astels, D.: Test-Driven Development: A Practical Guide. Prentice Hall, Englewood Cliffs (2003)
4. Floyd, R.W.: Nondeterministic algorithms. J. ACM **14**, 636–644 (1967)
5. Mackinnon, T., Craig, P., Freeman, S.: Endotesting: unit testing with mock objects. In: Succi, G., Marchesi, M. (eds.) Extreme Programming Examined, pp. 287–301. Addison-Wesley Longman Publishing Co., Redwood City (2001)
6. Freeman, S., Mackinnon, T., Pryce, N., Walnes, J.: Mock roles, objects. In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, pp. 236–246. ACM (2004)

7. Beck, K.: Extreme Programming Explained. Addison-Wesley Professional, Boston (2004)
8. Martin, R.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs (2002)
9. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional, Boston (2009)
10. Beck, K., Gamma, E.: JUnit test infected: programmers love writing tests. In: Dwight Deugo, pp. 357–376. More Java Gems (2000)