Ivo *et al. Journal of Software Engineering Research and Development* (2018) 6:9
https://doi.org/s40411-018-0053-5

Journal of Software Engineering Research and Development

RESEARCH Open Access

An approach for applying Test-Driven Development (TDD) in the development of randomized algorithms

André A. S. Ivo[1,2*], Eduardo M. Guerra[2], Sandy M. Porto[2], Joelma Choma[2] and Marcos G. Quiles[3]

*Correspondence:
andre.ivo@gmail.com
Development of the technique; Framework development; bibliographic research in TDD; design, conduction, and analysis of the experiment; lead the paper writing.
[1]Centro Nacional de Monitoramento e Alertas de Desastres Naturais (CEMADEN), Estrada Dr. Altino Bondensan, 500 - Coqueiro, 12247-016 São José dos Campos - SP, Brazil
[2]Instituto Nacional de Pesquisas Espaciais (INPE), Av. dos Astronautas, 1.758 - Jardim da Granja, 12227-010 São José dos Campos - SP, Brazil
Full list of author information is available at the end of the article

Abstract

TDD is a technique traditionally applied in applications with deterministic algorithms, in which the input and the expected result are known. However, the application of TDD with randomized algorithms have been a challenge, especially when the execution demands several random choices. The goal of this paper is to present an approach to allow the use of TDD in the development of randomized algorithms, and the Random Engagement for Test (ReTest) framework, a JUnit extension that provides support for the proposed approach. Furthermore, this paper present the results of a single-subject experiment carried out to assess the feasibility of the proposed approach, and a study on developer experience when using ReTest framework. Our results support the claim that the proposed approach is suitable for the development of randomized software using TDD and that the ReTest framework is useful and easy to use.

Keywords: TDD, Randomized, Tests, Framework, JUnit, Metadata, Code, Annotations

1 Introduction

When the execution of a software system always leads to an expected result, it is considered a deterministic algorithm. However, there are several algorithms whose execution might result in different correct outputs for the same input. This kind of algorithm with non-deterministic behavior is named randomized algorithms (Cormen et al. 2001). Commonly, these algorithms depend on various random decisions made during their execution.

Randomized algorithms are widely used to solve problems that might have several correct or acceptable answers for a particular input. For instance, to simulate stochastic systems or in approximation algorithms to solve intractable problems. Some classic examples of randomized algorithms are *primality testing*, heuristic algorithms to the *traveling salesperson*, *k-clique*, and *multiprocessor scheduling*.

Randomized algorithms use functions that return a pseudo-random number in a given range or a random element from a given set. In some cases, when this function is used only a few times, the use of automated testing becomes feasible, because it is possible to execute the algorithm simulating several return values for the random functions under consideration. However, if the random function is used several times or a variable number of times, it is not reasonable to simulate the value of several calls. Therefore, it is not

practical to simulate all of these paths to test a randomized algorithm with several random choices. Moreover, even if it were feasible to simulate all these answers, they would not be truly random. As any code with a lack of tests, when these algorithms need maintenance, or new solutions need to be developed, faults can be inserted into the system without notice.

TDD is a software development technique in which tests are developed before the code, in short and incremental cycles (Guerra and Aniche 2016). This 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 (Beck 2002; Astels 2003).

TDD is traditionally used with deterministic algorithms when there is a known input and one expected result. The challenge addressed in this work is to use TDD in applications with non-deterministic behavior as stated before. Although it is not possible to know exactly what the output will be, it is usually possible to check whether the generated output is valid or not.

The following factors make it difficult to develop randomized software using TDD: (a) results for each execution may be different for the same inputs, which makes it difficult to validate the return value; (b) obtaining a valid return for a test case execution does not mean that valid return will be delivered on the next executions; (c) the random decisions and their paths number make it not viable to create Mock Objects (Mackinnon et al. 2001; Freeman et al. 2004) that return fixed results for these decisions; and (d) it is difficult to execute a previous failed test with the same random decisions undertaken in its former execution.

In this paper, we present (1) an approach that allows an extension of TDD for its application in algorithms with non-deterministic characteristics, and (2) a framework for the JUnit called ReTest (Ivo and Guerra 2017b), developed by the authors in order to support the random algorithms testing. In short, the proposed approach uses multiple verification to validate of the algorithm return and executes each test several times with different seeds to increase test coverage. From the result of these repetitions, the seeds from pseudo-random test cases that generated failures are stored and used in future tests, ensuring that a scenario where an error was detected in the past is executed again.

Furthermore, we present the results of two studies: a single-subject experiment carried out to assess the feasibility of the proposed approach in an algorithm implementation involving multiple random decisions in sequence, and a study to assess the developer's experience when using ReTest framework that automates the approach steps.

The paper is organized as follows. Section 2 introduces the background necessaries knowledge. Section 3 describes the method we applied in this research. Section 4 presents our approach outlined from a set of patterns that assist in the test automation of randomized algorithms. Section 5 presents a single-subject experiment of an implementation that used the proposed approach. Section 6 presents the ReTest framework, including an example of use in the context of TDD. Section 7 describes a study carried out to assess the developer's experience when using ReTest framework. Section 8 presents the discussion and limitations of the research. Finally, Section 9 presents the conclusions and future perspectives.

2 Background

In this section, we present concepts and techniques related to the approach for applying TDD in the development of randomized algorithms. Section 2.1 presents the types of randomized algorithms that can be developed with the proposed approach. Section 2.2 describes TDD as a coding and design technique, and Section 2.3 introduces the JUnit, one of the most used frameworks used for creating unit tests, used as the basis for the ReTest framework.

2.1 Randomized algorithms

Many algorithms might consider, at some point, one or more random steps during their executions. For instance, algorithms used to find an approximate solution to NP-Hard problems or Monte Carlo Algorithm, usually take random choices into account.

These randomized algorithms are commonly categorized into two classes (Cormen et al. 2001):

- Monte Carlo Algorithms: define a class of algorithms that generally run quickly and return the correct answer with high probability, albeit, it might return a wrong answer. Approximation algorithms, such as Genetic Algorithms, are variations of the Monte Carlo Algorithms.
- Las Vegas Algorithms: which are those algorithms that always return a correct answer, but running time is random, although finite (Galbraith 2012).

The algorithm used in the study to verify the viability of our approach (Section 5) is the Las Vegas type algorithm that always gives to a correct answer (or so the proposed technique guarantees), but performs at random time. Using a graph as input, the algorithm applies a random sequence of transformations, and the output should be a graph that has some specific metrics. Notwithstanding, the approach proposed in this study can be applied to both classes of randomized algorithms.

Finally, it should be stated that the stochasticity or non-determinism considered here are not related to the theoretical concept of non-determinists algorithms. These non-determinists algorithms rely on the architecture of a non-deterministic Turing Machine (Floyd 1967). In our scenario, non-determinism is related to the stochastic behavior of the algorithm.

2.2 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 works reported by Guerra and Aniche (see (Guerra and Aniche 2016)) that indicates that the use of TDD can improve 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) (Beck and Andres 2004), however today it is also used out of this 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 smallest amount of code possible to implement the new requirement verified by the test. At this point, the added test, as well as all other previously existing tests, are expected to run successfully. Once the tests are successfully

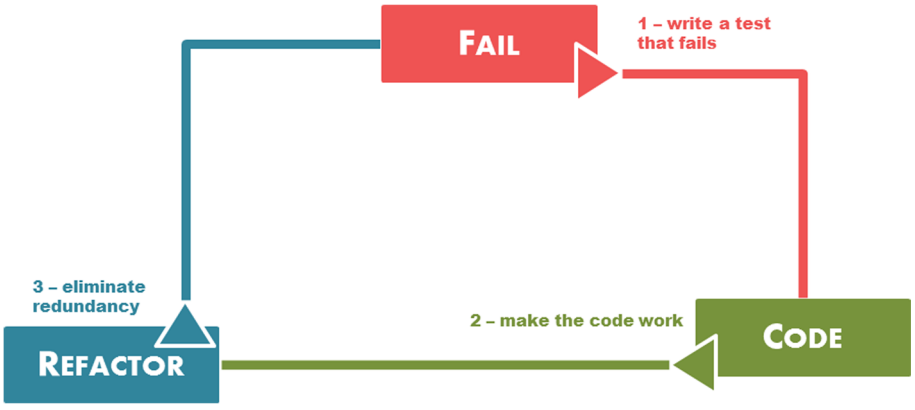

Fig. 1 TDD execution diagram

executed, the code can be refactored so that its internal structure can be continuously evolved and improved. The tests help to verify if the behavior has not been modified during refactoring.

This cycle is performed repeatedly until the tests added verify scenarios for all expected class requirements. The TDD cycle is presented in Fig. 1 (Beck 2002; Astels 2003).

With the use of TDD, the code design is defined in cycles. The idea is that with each new test added, a small increment of functionality is made compared to the previous version. TDD technique widely used in industry, being described in several books, such as "Test-Driven Development by Example" (Beck 2002); "Agile Software Development, Principles, Patterns, and Practices" (Martin 2003); "Growing Object-Oriented Software, Guided by Tests" (Freeman and Pryce 2009); and "Test-Driven Development: A Practical Guide" (Astels 2003).

2.3 JUnit framework and its extension points

JUnit is a Java open-source framework for the creation of unit tests. Its purpose is to be a basis for the creation of test automation code. It is used for the practice of TDD, and its model has been taken into account to create test frameworks for other languages. Some of the main features of these frameworks are the execution of test cases and the presentation of the execution results (Beck and Gamma 2000).

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 as the default runner. The Runner class hierarchy is represented in the diagram in Fig. 2.

In this way, to implement a new test runner it is necessary to create a new class extending Runner. To use it in a test class, the annotation *@RunWith* should be configured receiving in its attribute *value* the new class that replaces the default test runner. If it is necessary to extend the existing functionality, the new runner should extend the *BlockJUnit4ClassRunner* class.

Another JUnit extension point is known as Rule and it allows the addition of new behaviors mainly before and after the execution of each test. To introduce a new Rule,

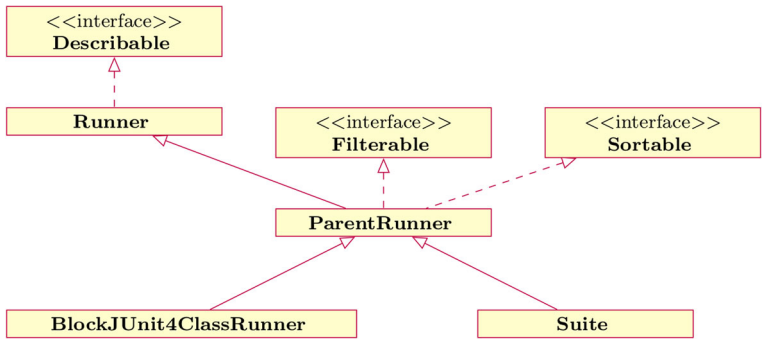

Fig. 2 Core class diagram of ReTest framework

it is necessary to create a class that implements the *TestRule* interface. To use it, a public attribute should be declared in the test class and annotated with *@Rule*, as shown in Listing 1.

Listing 1 @Rule use example

```
1  public class TestClass {
2
3    @Rule
4    public NewRule newRule = new NewRule();
5
6    @Test
7    public void testMethod(){
8      Object result = TestedClass.testedMethod();
9      assertResult(result);
10   }
11 }
```

In the example shown in Listing 1, when executing the test project, the newRule is called before and after the *testMethod()*. The annotation @Rule is responsible for adding the desired behaviors before and after each test.

3 Research method

The need of this research was raised during the development of a randomized algorithm in the National Institute of Space Research. The developer was familiar with the TDD technique but did not know how to apply it in that context. Based on that need from a real implementation the approach was developed.

The proposed approach was initially evaluated in a feasibility study, performed through a single-subject experiment that used the algorithm that motivated its development. Single-subject studies are one way to provide empirical support for software-engineering techniques, in which only one subject performs the tasks in the study (Harrison 2000). Single-subject studies usually are analyzed by determining whether the treatment's effects are noticeable, and therefore does not require elaborate statistical analysis (Harrison 2005).

In this single-subject experiment, our approach was applied in the algorithm implementation involving multiple random decisions in sequential performing. The results of this implementation using our approach was compared to the same algorithm previous implemented in the traditional way, using the same programming language and without

unit tests. Information about the effects of using and not using the approach was provided by the researcher, the single subject who implemented twice the same algorithm.

The set of practices identified as the core of the proposed technique were documented as three patterns (Ivo and Guerra 2017a). The published paper focused on practitioners and provided a detailed description of each pattern. It has a different goal from the present paper that focuses on the research involved in the development and evaluation of the approach.

Based on the documented practices and in the experience of using the approach for a complex algorithm implementation, a testing framework for Java called ReTest was developed. This framework has features that make easy the usage of the proposed approach, such as the generation of seeds, the repetition of test execution and the support for re-executing previous failed tests.

After the framework implementation, we performed a study to evaluate the applied approach with the support of the ReTest framework in terms of ease of use, usefulness, perceived benefits, drawbacks, and difficulties. In this study, 10 participants implemented a simple algorithm using the ReTest framework for tests and following an established procedure to guarantee the application of TDD. At the end of the activity, a questionnaire with open and closed questions was used to gather the opinion and perceptions of the developers-participants regarding their experiences with the use of the framework in the implementation task. Studies on the developer experience have been important in several areas of software development, such as in the software process improvement (Fagerholm and Münch 2012). As a consequence, studies based on developers' perception can give valuable input for analyzing and adjusting new development techniques, procedures and tools.

4 Approach to automate tests randomized logic

In general, existing test techniques are suitable for applications with deterministic algorithms, when one has a known entry and an expected result. More traditional testing strategies often focus on exploring the input domain by evaluating the values used as input and obtained as outputs to find combinations of input parameters that can cause failures.

Several testing techniques focus on indicating a set of tests necessary to a proper coverage of the input domain. *Combinatorial testing*, for example, combines all possible rearrangements of the input parameters and submits this list to the algorithm to be tested (Anand et al. 2013), such as Pairwise testing (Bach and Schroeder 2004). Unfortunately, the coverage of the input domain is not enough for randomized algorithms, because even for the same input the result can be different. In other words, a successful execution might fail in the next time it is executed with the same inputs.

In order to automate tests of randomized logic, we have proposed an approach based on three software patterns: *Deterministic Characteristic Assertion* - to create an assertion that verifies the validity of the algorithm result; *Re-test With Different Seeds* - to execute the test several times with different seeds; and *Recycle Failed Seeds* - to persist seeds used in failed tests to be reused in future test executions. Next, we present an overview of each pattern. And, after the explanation of the patterns, we present the application of the proposed approach through an illustrative example.

4.1 Deterministic characteristic assertion

The *Deterministic Characteristic Assertion* pattern proposes the usage of assertions based on a criteria that verifies deterministic characteristics of the result at the end of the execution. If all criteria are met, the results can be considered valid. This pattern can be associated with test oracles. An ideal test oracle provides a pass/fail judgment for any possible program execution, judged against a specification of expected behavior (Baresi and Young 2001).

This pattern is related to how the assertion should be performed, and it is related to the correctness of the algorithm. In theoretical computer science, the correctness of an algorithm is asserted when the algorithm is correct concerning a specification. Functional correctness refers to the input-output behavior of the algorithm (i.e., for each input it produces the expected output) (Dunlop and Basili 1982).

The solution is summarized in the selection of fundamental characteristics that determine the validity of the algorithm result. An analysis of the algorithm should be performed with the intention of identifying the main requirements and behaviors that do not change in a valid result. Then, these items should be related to a list of deterministic characteristics that can be verified.

For each deterministic characteristic, an assertion procedure should be developed. After developing the set of characteristics verification, the algorithm should be executed, and its result submitted to the assertion set. If all set of characteristics tests run successfully, it means the output is valid, regardless of its exact value.

In general, there is an intention of evaluating the behavior of the algorithm independently of the outcome. In some cases, it is possible to extract the characteristics of the result itself, in others, it is not. As a good practice, there may be cases where it is necessary to divide the algorithm into some parts so that it is possible to isolate the randomized parts and to verify deterministic characteristics from its intermediate results.

4.2 Re-test with different seeds

This pattern is focused on increasing test coverage, which is not the same thing as functional correctness, which is covered by the *Deterministic Characteristic Assertion* pattern. Test coverage is not an aspect of product quality; it is just one measure of how well exercised a product is. Thus, broader code coverage correlates with fewer expected undiscovered product bugs.

For this reason, the proposed approach applies this pattern combined with *Deterministic Characteristic Assertion*. To use this pattern, the developed algorithm should be able to receive the object used to generate the pseudo-random inputs or the seed that should be used. The test using *Deterministic Characteristic Assertion* pattern should be executed several times varying pseudo-random inputs generator or its seed. The seed for each execution can also be implemented using a pseudo-random input generator.

In general, a library with functionality to generate pseudo-random values based on a seed is present in several programming languages. A seed is generally an integer and acts as a "key-primary" in pseudo-random generations. When the pseudo-random generation algorithm receives a seed, following generations must always result in the same sequence, therefore the so-called "pseudo-random" generation.

The combination of pseudo-random inputs and the number of repetitions causes the algorithm to be exercised traversing different paths. So, the larger is the number of repetitions, the wider is the code coverage.

For cases where the complexity of the code and the number of features is high, it is recommended to increase the number of repetitions to generate a more extensive code coverage.

It is important to state that the use of this pattern does not guarantee that the tests will cover all possibilities. However, especially in an environment where the tests are frequently executed, after several executions, it is possible to have good coverage, providing confidence that the algorithm is implemented correctly.

4.3 Recycle failed seeds

This pattern is used to enable the re-execution of test cases that failed in previous executions. One of the most significant problems in randomized algorithms is debugging, since its stochastic behavior makes the reproduction of an error a complex task. This feature makes it hard to apply specific modern techniques in software quality, for example, regression tests. The primary goal of regression testing is to provide confidence that the newly introduced changes do not generate errors on existing and/or unchanged parts of the software system (Yoo and Harman 2012).

Regression testing may become very large during the process of integration tests. Thus, it is unpractical and less efficient to re-run all the tests for each modification (Xiaowen 2013). Because of these difficulties, alternative approaches to the Regression Test Selection (RTS) technique has been proposed in Agrawal et al. (1993); Chen et al. (1994); Harrold and Souffa (1988); Hartmann and Robson (1990); Rothermel and Harrold (1997). RTS approach recommends that only a select subset of the tests be run to optimize and minimize the time of the execution. This subset must be selected according to the need and characteristics of the software system being developed (Kim et al. 2000).

As a complete solution for tests of randomized algorithms, as well as Adaptive Random Testing (ART), this article proposes the division of the input domain into two sets. However, unlike ART, the first set of the input domain is only a projection of the actual test cases, the second set is the test cases that have presented errors in executions.

To get the first set of the input domain, one must use the pseudo-random generations as presented in the *Re-test With Different Seeds* pattern. To obtain the second set of the input domain, the seeds associated with the failed tests should be preserved. Select the seeds from the failed tests, and perform new tests with these seeds, inducing the randomized algorithms to repeat the same paths and behaviors that led to failure. Like the *Re-test With Different Seeds* pattern, this pattern is focused on increasing code coverage.

4.4 Approach applied: an illustrative example

To represent a method whose return is random, consider an algorithm that generates an array of "n" positions, with pseudo-random numbers varying between 10 and -10, whose total sum of elements is always zero. Note that in this example a number of possible returns grow exponentially with "n". The following formula calculates the number of possible returns, where "n" is the array size and "e" is the number of elements:

$$Possibilities = e^n \tag{1}$$

Since between -10 and 10 we have 21 elements (including 0), the following calculates the number of possibilities for an array of 3 positions:

$$Possibilities = 21^3 = 9261 \tag{2}$$

Each of the possibilities presents a path that must be tested, so the amount of testing rises exponentially with the number of possibilities. In fact here the question is not the number of possible answers, but the fact that there are several correct answers and the algorithm can return different answers to each execution.

The Listing 2 shows a simple example of how to use the proposed test approach. The first software pattern (*Deterministic Characteristic Assertion*) is introduced with *assertValueInterval()* and *assertSumEqualZero()* functions. These functions represent the point where all assertions of the characteristics are performed.

The second software pattern (*Re-test With Different Seeds*) is represented by a software loop. On the Listing 2, the use of this pattern can be observed in the first *for* statement. The focus of this pattern is to increase the code coverage and for a better performance (functional correctness) should be used in conjunction with the pattern *Deterministic Characteristic Assertion*. For this reason, the assertion methods are invoked inside this loop.

Listing 2 also demonstrates how to use the *Recycle Failed Seeds* pattern, the third software pattern proposed. This pattern proposes to save the seeds when the characteristics assertion fails, so the same seeds can be used later in future executions.

Still in Listing 2, the invocation of the method *saveFailureSeed()* represents the logic to save the seed. The seed can be saved in memory, in a file or even in a database, depending on the test requirements.

Listing 2 Simple example of how to use the proposed test approach

```
1   @Test
2   private void testFixedCharacteristics(){
3     int n = 2;
4
5     //First set of the input domain.
6     //Number of repetitions.
7     int numberRepetitions = 100;
8     for(int i=1;i<=numberRepetitions;i++){
9       //New seed is introduced based on the computer clock.
10      long new_seed = System.currentTimeMillis();
11      Random rSeed = new Random(new_{s}eed);
12      int[] result =
            ArrayFactory.generateArrayBasedRandomSeedWithSumZero
            (rSeed, n);
13
14      if(assertValueInterval(result, n)==false){
15      saveFailureSeed(new_{s}eed);
16      };
17      if(assertSumEqualZero(result, n){
18        saveFailureSeed(new_{s}eed);
19      };
20    }
21
22    //Second set of the input domain.
23    long[] allSeeds = getAllFailuresSeeds();
24    for(int i=0;i<=allSeeds.length-1;i++){
```

```
25        //Get failed seed.
26        long fail_{s}eed = allSeeds[i];
27
28        //A pseudo-random input is generated based on failed seed
29        Random rSeed = new Random(fail_{s}eed);
30        int[] result =
              ArrayFactory.generateArrayBasedRandomSeedWithSumZero
              (rSeed, n);
31
32        assertValueInterval(result, n);
33        assertSumEqualZero(result, n);
34      }
35    }
36
37    private void assertValueInterval(int[] arr, int arraySize) {
38      int result = 0;
39      //verify if value are between -10 and 10
40      for (int i = 0; i < arraySize; i++) {
41        assertTrue(arr[i] >= -10 && arr[i] <= 10);
42      }
43    }
44
45    private void assertSumEqualZero(int[] arr, int arraySize) {
46      int result = 0;
47      //verify if sum equals 0
48      for (int i = 0; i < arraySize; i++) {
49        result = result + arr[i];
50      }
51
52      //verify the sum
53      assertEquals(0, result);
54    }
```

In Listing 2, the second part of the input domain is represented by a loop with the failed seeds, implemented in the second *for* statement. Inside the loop, note that the input values are generated through a pseudo-random mechanism that receives the seed that failed as a the parameter, which then reproduces the same inputs, inducing the same behavior in the test.

5 Feasibility study: a single-subject experiment

In this section, we report the results of the first study investigating the effects of proposed approach applied for implementation of a complex randomized algorithm involving the creation and evolution of dynamic graphs. First, we provide an overview on domain and related concepts, giving an idea about the complexity of the problem handled. Then, we describe the single-subject experiment carried out by the researcher proponent of the algorithm for dynamic graphs and, finally, we report the results of this study.

5.1 Dynamic graphs

A dynamic graph is a graph that is constantly changing. They are those in which their elements, vertices, and edges, are constantly being added or deleted. The benchmark proposes a controlled dynamic environment by providing functions that simulate transformations that may occur in a graph considering its structure in communities.

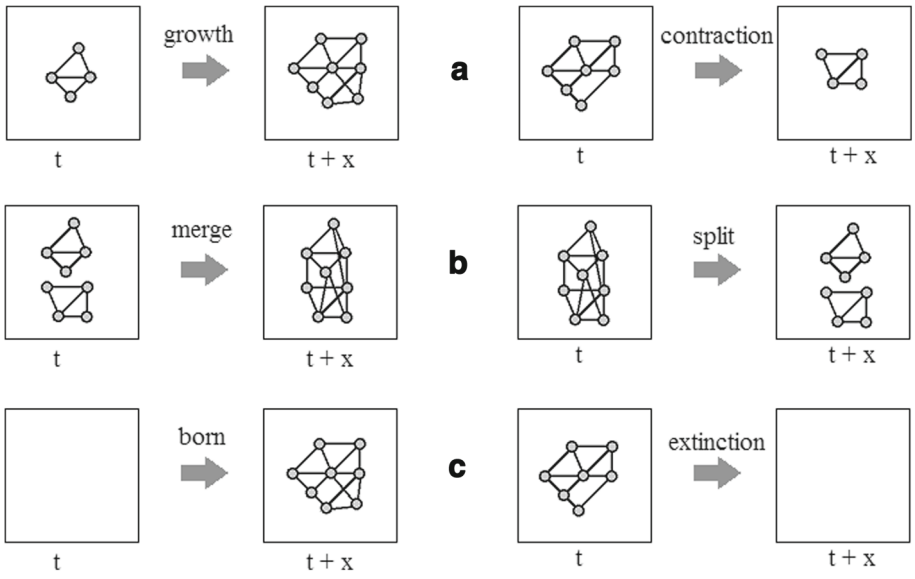

Fig. 3 Possible changes that a community can suffer (Porto and Quiles 2014)

The community structure of a graph is a metric that can be used to cluster its vertices. One can define which groups of vertices form a community through the comparison between the density of the graph and the density of the group. The density is calculated considering the number of edges within the vertices of the community, where more edges mean higher density.

All functions that simulate transformations reach their goal by making changes in the graph elements. Basically, there are three types of transformations, which are illustrated in Fig. 3: (a) one-to-one, which involves the *growth* or *contraction* of the community; (b) one-to-many or many-to-one, that are related to the *splitting* or *merging* phenomena, i.e. one community can give rise to several others or several communities can merge into a single one; and (c) one-to-zero or zero-to-one, represented by the *birth* or *termination* of a community (Porto and Quiles 2014).

5.2 Dynamic communities algorithm

The algorithm described in this study creates a sequence of graphs, each containing its own structure in communities, to simulate the behavior of a dynamic graph. Dynamic communities algorithm is composed of six functions that simulate these behaviors, named: born, growth, extinction, contraction, merge, and split. Hence, the output this algorithm is a graph that passes through a sequence of random changes considering the six possible functions stated before; this response must have specific properties.

The properties are some topological measurements extracted from the graph. The output graph must preserve the measures of the input graph according to some previously established parameters. Here, the measurements are the average degree of the network and the maximum degree, in which the degree of a vertex is the sum of the number of edges connected to it; the mixing parameter, which controls the fraction of edges that a vertex shares with vertices belonging to other communities; and, finally, the minimum and maximum size of the communities. While it is simple to check these properties, it is not trivial to implement an algorithm that generates a final graph that complies with

them. As a consequence, there must be a test routine to facilitate and give confidence to the code.

However, in addition to the challenges already mentioned above, another challenge to test algorithms treated in this work is the *input domain* (Elva 2013). The input domain is any possible combination of all input parameters of a method. However, if the parameters can be chased from an infinite set, which is the case in this experience, it means that the input domain is also infinite. Therefore, not every possible combination is available.

As already stated, the algorithm output is a graph which undergoes a sequence of random changes using the six possible change functions. The input of each function is also a graph, so the output of one function can be used as input to the next one, without any interference. However, in the tests, each output is checked whether it is under the properties established before, and it is used as input to the next function.

Equation 3 gives an example of a possible sequence executed by the algorithm, where g_i is the initial input graph and g_o is the final output graph.

$$g_o = born(extinction(born(growth(g_i)))) \tag{3}$$

5.3 Randomized algorithm

Randomness plays an important role in the algorithm. The six functions aim to create a dynamic network with community structure evolution, step by step, to serve as a benchmark for community detection algorithms. It means that the randomness is not only a consequence of the way the algorithm has been implemented but a requirement.

Each function, born, extinction, growth, contraction, merge, and split will perform a series of modifications in the network. Such modifications involve adding and removing vertices and edges of one or more communities. Also, despite the algorithm having control of which are the vertices and edges of the network that may undergo such changes, the number of times that they occur is decided only at runtime. The only estimate one can make is that this amount is proportional to the number of vertices that are being treated by the function.

For example, the *merge* function first chooses how many (first random decision) communities will undergo changes, which can be at least two and up to all the communities. Then choose which (second random decision) communities will merge with which. From there, it sums up the total amount of vertices in the communities and calculates the minimum expected number of edges (density) to merge the selected communities. Then, pairs of vertices are selected (multiple random decisions in sequence) and become adjacent until the desired density is achieved.

Randomness was present in all steps of the merge function, as described in the previous paragraph, and is also present in all other functions. The first and second random decision of the merge function define how many decisions will be made after and the same principle applies to all other functions as well. This is a significant problem when testing computer code, after all, before running there is no indication of how many arbitrary decisions will be needed. It is only known that this quantity will be proportional to the number of vertices treated in the function. Furthermore, even this amount is only determined at runtime.

From the input graph, each function generates a sequence of graphs that correspond to the evolution (or transformation) of the graph in time. Thus, community detection algorithms can be tested in the most diverse situations regarding the order of application

of each one of the six functions of the algorithm. Ideally, this order must be as random as possible, as well as the number of transformations. Both order and quantity are chosen at runtime.

5.4 Experimental design

A single-subject experiment was performed to verify the feasibility of the approach proposed to applying TDD in the development of the randomized algorithm for creation and evolution of dynamic graphs. The single-subject in this study is a computer scientist who develops research in the area of Machine Learning using Complex Networks, in the Associated Laboratory of Computation and Applied Mathematics at the INPE. The task assigned to researcher-developer was to implement the Dynamic Communities Algorithm using our approach, which had originally been implemented by herself in R programming language (R Core Team 2015) in a traditional way. The participant received previous training on TDD which consisted of theoretical classes and practical exercises, during a course in software design at the INPE. On this occasion, the approach to using TDD with randomized algorithms was presented.

The independent variables of the study are (1) the programming language, (2) the randomized algorithm, and (3) the developmental approach that is also the treatment variable. The objectives dependent variables observed in this study are: lines of code, number of functions, lines of code per function, lines of test code, and number of test cases. The developer's view about the effects of applied treatment is a subjective variable that was considered in this study.

Single case experimental designs usually involve an initial period of observation referred to as the "baseline" or "A Phase". Upon completion of this phase, the independent variable is manipulated during the second period of observation referred to as the "treatment" or "B Phase", in order to verify the dependent variables. This procedure is known as an A-B design (Harrison 2000).

5.5 Results

As aforementioned, the Dynamic Communities algorithm was twice implemented. Regarding implementation time, in the first time, the researcher-developer took approximately three months to implement the algorithm in the traditional way, using R as the programming language (A Phase). In this phase, the long implementation time was due to the fact that the solution was not well defined from the beginning. During this implementation, a period of research was necessary to study the application requirements and define the logic that would be employed.

In the second time, the researcher-developer took approximately three weeks to implement the algorithm using the same programming language, but this time following the proposed approach to the use of TDD with randomized algorithms (B Phase). Less time was needed in this phase since the research had already been consolidated with a well-defined algorithm. The implemented code is hosted on GitHub[1].

In the second implementation, the output graph (the last one in the sequence) is tested for its metrics, being the average degree, maximum degree, mixing parameter and size of the smallest and largest graph community. The mentioned metrics form the group of characteristics that are evaluated in the output. That is, in this case, is employed the *Deterministic Characteristic Assertion* pattern, as described in the Section 4.1.

Furthermore, if the graph passes all tests, it is used as input to the next function; if not, the seed used is stored to be used in the regressions tests, employing *Recycle Failed Seeds* pattern, as described in the Section 4.3. All of these metrics must be within a previously established threshold for the graph be considered correct. Note that the *Re-test With Different Seeds* pattern was not used in this implementation because it met a very specific situation.

Listing 3 summarizes the tests applied in each experiment. First, the seeds of this battery of tests are chosen, the randomness of the choice is to fulfill the *Re-test With Different Seeds* pattern (Section 4.2). Each seed is an increase on the test coverage. For each seed, an initial graph is created and tested. Then, the size of the sequence of functions is chosen. For each function to be applied in the current graph, the output is tested. If any test fails during execution, the seed is stored. The initial set of seeds can be altered to receive the set of seeds that led to errors.

Listing 3 Resume of the tests performed in the experiment

```
1   q = 100; //arbitrarily chosen
2   seeds = random_{n}umbers(q);
3   for(seed in seeds){
4       g_{0} = create_{i}nitial_{g}raph();
5       if(test(g_{0}) fails) store(seed);
6       n = choose_{q}uantity(); //size of the sequence
7       for(i in 1:n){
8           f = choose_{f}unction(); //choose one of the six
                functions
9           g_{i} = f(g_(i-1)); //applying the choosen function
10          if(test(g_{i}) fails) store(seed);
11      }
12  }
```

The tests performed in each graph, besides checking the metrics already mentioned, also make specific checks considering which function was applied. For example, if the born function was used, one community with a specific size (number of vertices) was added to the graph, so it is necessary to verify if the number of n_c communities increased by exactly one unit and if the number of vertices in the graph increased precisely the size of the new community. In the extinction function, the inverse applies, one community with a specific size was eliminated of the graph, so one needs to verify if the number of communities decreased by exactly one unit and if the number of vertices decreases the size of the community eliminated.

In the growth and contraction functions, it is necessary to verify if the number of communities has remained, but the number of vertices has increased (or decreased). In the merge and split functions, it is necessary to verify if the number of vertices remained the same, after all, only the edges are modified in these functions, but the number of communities must decrease (or increase).

At this point, the researcher-developer highlighted that some situations which could lead to errors had not been detected during the first implementation. Such as (a) empty graph as input, (b) graph with only one community as input, and (c) pre-checking the value of the mixing parameter before changing any edges. The latter situation allowed the algorithm to become more efficient. The researcher-developer recognized that such improvements were only possible with the use of our approach.

Table 1 Results of the two phases of implementation

	A Phase	B Phase
Lines of code	522	766
Number of functions	8	24
LOC per function	65.25	31.91
Lines of test code	0	417
Number of test functions	0	112

In the GitHub repository, there is a file named "Erros.dat"[2] containing all the seeds that exposed an error along with a message describing the detected error. The file contains 236 rows, each row being a seed that was stored, so during the algorithm's development, 236 errors were detected and corrected.

There were also tests related to the storage and transfer of graph sequences from main memory to secondary memory. Table 1 shows the code metrics reported in the first phase of implementation (A Phase) using a traditional process, and in the second phase (B Phase) using the approach applying TDD. The first thing that can be noticed in the table is the presence of test code in Phase B.

By the numbers in Table 1, it is possible to notice an increase in the number of functions. This more granular division of the code favors the testing activity since it is easier to evaluate the behavior if the function has a more limited scope. Smaller and more granular functions also indicate that the code in B Phase have a better design and it is easier to maintain.

An increase in the number of lines of code can also be noticed by comparing the two implementation phases. Despite the additional code for handling scenarios that were not considered in the initial version, this increase could also be justified by the developer attention to code readability. With automated tests, it was safer to refactor the code continuously during its implementation.

In the opinion of the subject researcher-developer of this study, the rewriting of the code using the TDD following the proposed approach demonstrated that *"even considering the complexity and randomness of the algorithm reported, the code gained a lot of quality, flexibility, maintainability, and confidence in the performance of the algorithm. Additionally, the organization of the tests also helped in the verification and validation of the results, comparing them with other results of the literature."*

6 ReTest: test framework for randomized algorithms

To support our approach for testing randomized algorithms, we developed a framework called ReTest (Random Engagement for Test), available at https://github.com/andreivo/retest. This framework provides 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 using the seeds used in previously failed executions. These features facilitate the application of TDD for the development of randomized algorithms.

It should be mentioned that the framework does not have any feature to implement the pattern *Deterministic Characteristic Assertion* since it is something very specific to each algorithm. However, its features support the implementation of the two other patterns, *Re-test With Different Seeds* and *Recycle Failed Seeds*.

6.1 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 meant to be used as input data for the tests. The Listing 4 shows a simple example of use.

Listing 4 Simple example of how to use ReTest

```
1   @RunWith(ReTestRunner.class)
2   public class TestClass {
3     @Test
4     @ReTest(10)
5     @SaveBrokenTestDataFiles(filePath = "/data/file1.csv")
6     @LoadTestFromDataFiles(filePath = "/data/file1.csv")
7     public void testMethod(
8       Object result = nonDeterministicAlgorithm(random);
9       assertResult(result);
10    }
11  }
```

In the code shown in Listing 4, 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 stochastic decisions. The *assertResult()* is used to verify 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, enabling the debugging for a specific test scenario.

The ReTest framework has an API that allows you to:

(a) generate random data to be used in the test methods;
(b) create custom randomizers for data in the application domain;
(c) save seeds from failed tests;
(d) save seeds from tests that were 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.

6.2 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 the developer to indicate custom randomizers for the specific data types in the application domain, allowing the extension of the framework for the random generation of several types of data.

6.3 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 Section 6.2. The Fig. 4 shows the class diagram of the *ReTestRunner* implementation. In this diagram, it is possible to observe the first extension point of the framework for customization of the data files format,

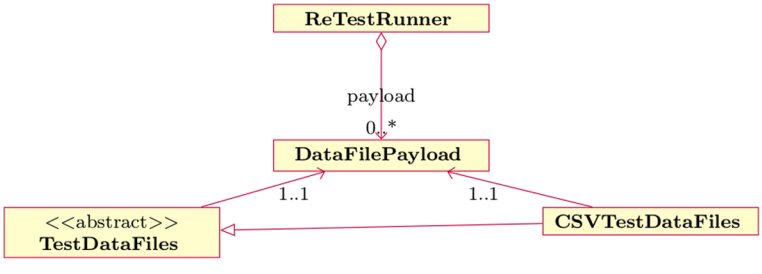

Fig. 4 Core class diagram of ReTest framework

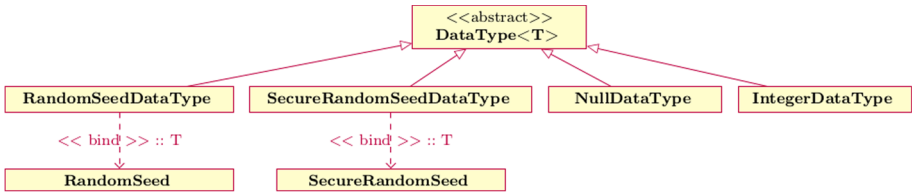

Fig. 5 Class diagram of randomized objects

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.

The Fig. 5 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 to be the data generator for a test, it should be configured as an attribute of the *@Param* annotation.

6.4 ReTest example

As mentioned earlier, TDD is not a technique normally used in the development of randomized algorithms, which depend on various random decisions made during their execution. Thus, one of the goals of the ReTest framework is to make the use of this technique feasible for these scenarios.

From the use of the proposed approach with the support of the ReTest framework, it is possible to complement the development cycle of TDD as observed in Fig. 6. 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;

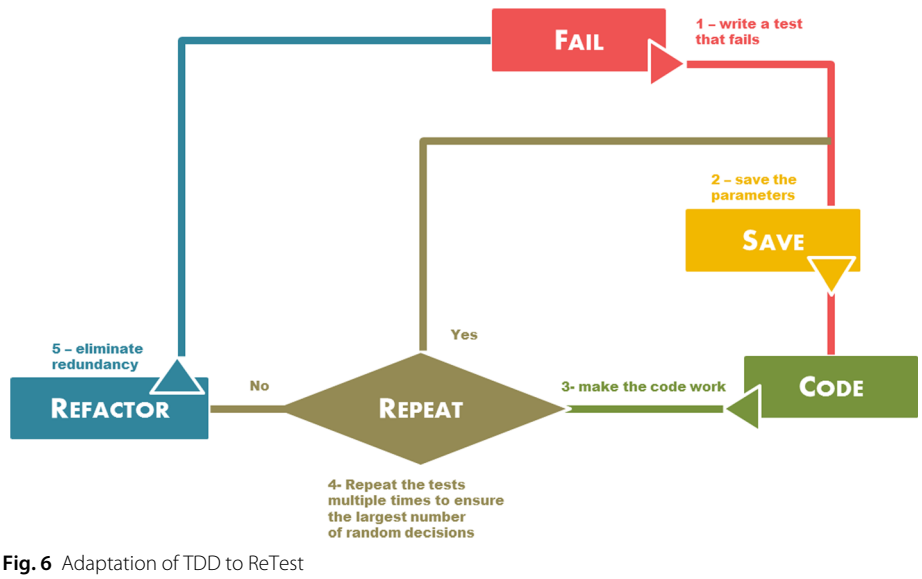

Fig. 6 Adaptation of TDD to ReTest

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 failed 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, presented in section 2.2, are included. 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 for generating automated tests for randomized 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, as described in Section 4.4. 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. For more information, see the code at https://github.com/andreivo/retest/blob/master/sources/retest/src/test/java/initial/case01

(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 the 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 Listing 5;

Listing 5 Method for evaluating rules

```
1   private void assertElements(int[] arr, int arraySize) {
2     int result = 0;
3     //verify if all
4     for (int i = 0; i < arraySize; i++) {
5       assertTrue(arr[i] >= -10 && arr[i] <= 10);
6       result = result + arr[i];
7     }
8
9     //verify the sum
10    assertEquals(0, result);
11  }
```

(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

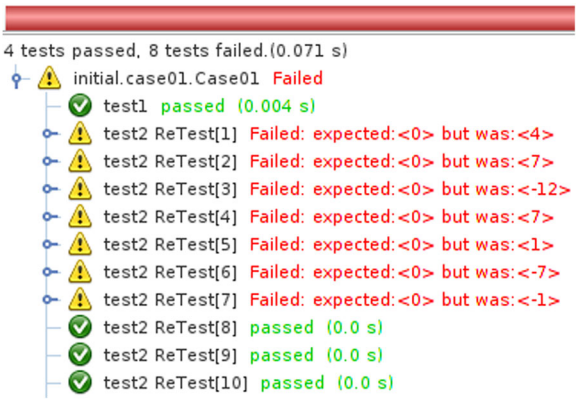

Fig. 7 Result of using ReTest for 2-position array

framework to execute it 10 times. The Fig. 7 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, the information about the failed tests 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, on the information of failed tests is stored and executed again, as can be seen in Listing 6; From this point the test code for other methods is similar to this one, varying only the parameter "n" passed to the function *generateArrayWithSumZero()*;

Listing 6 Example of test method

```
1  @Test
2  @ReTest(10)
3  @SaveBrokenTestDataFiles(filePath = "/tmp/dataTest.csv")
4  @LoadTestFromDataFiles(filePath = "/tmp/dataTest.csv")
5  public void test2(@RandomParam Random r) {
6    int n = 2;
7    int[] result = ArrayFactory.generateArrayWithSumZero(r,
         n);
8          assertElements(result, n);
9  }
```

(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. 8;

(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 9 shows the execution of the tests for

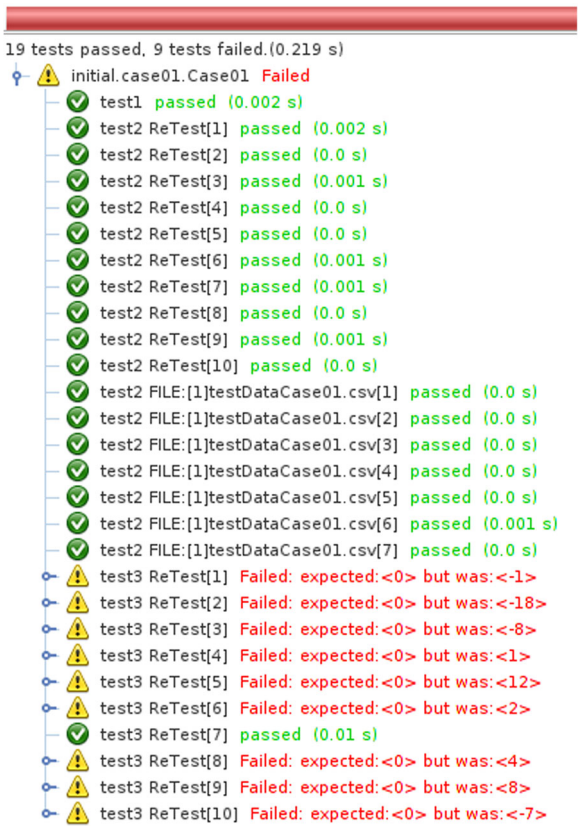

Fig. 8 Result of using ReTest for 3-position array with previous tests

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 randomized algorithm. Note that test cases are gradually being introduced and implementation is also occurring incrementally.

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.

7 ReTest framework: a developer experience assessment

This section describes a study carried out to assess our approach from the software developer experience, applying it through the ReTest framework.

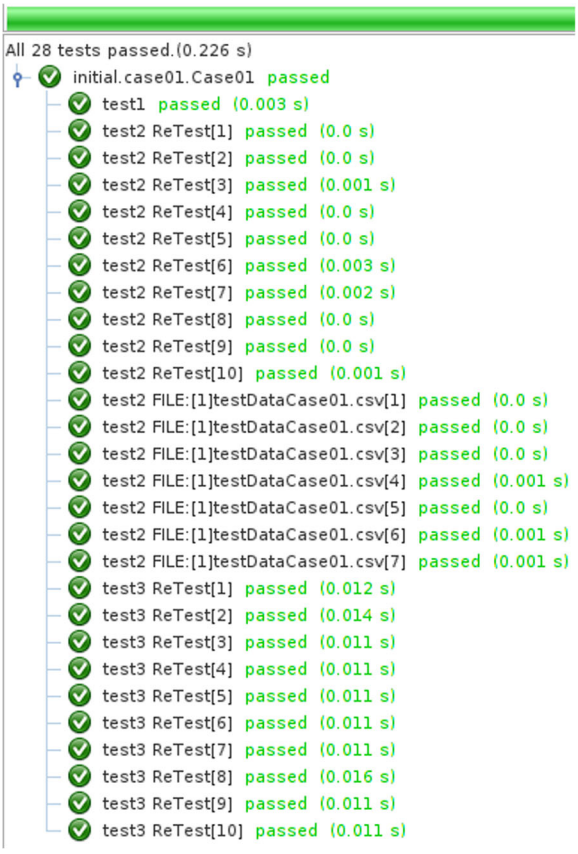

Fig. 9 Final result of the example with all tests running

Initially, the idea was to perform a comparative study to the development with TDD without a previous knowledge of the proposed approach. However, in the first attempt to conduct an experiment, most participants claimed that they did not know how to use TDD to develop a randomized algorithm. Additionally, we also considered that a comparative study with participants using the proposed technique with and without ReTest would not be fair since the framework automates the steps that would be manually implemented without it. Based on that, we decided to perform a study to investigate whether the proposed approach could be used successfully by developers, assessing their experience when using it.

7.1 Goals

The purpose of this study is to assess some aspects such as ease of use, difficulties, usefulness, and benefits of this approach that allows the use of TDD in the development of randomized algorithms supported by the ReTest framework from the viewpoint of software developers in the context of the software design course at the INPE. In so doing, we have investigated the following research questions:

RQ1: Does the proposed approach supported by ReTest framework can be efficiently applied by less experienced developers?

RQ2: What were the main difficulties encountered by the developers using the proposed approach?

Table 2 Participants background in terms of years of experience

Experience	Group	Min	Mean	Max	StDev
Programming	Group 1	01.0	03.0	05.0	1.58
	Group 2	10.0	14.6	20.0	4.21
Java language	Group 1	01.0	02.8	05.0	1.78
	Group 2	02.0	09.8	16.0	5.76

RQ3: What are the benefits and drawbacks of the proposed approach from the point of view of the developers?

RQ4: What is the usefulness of the proposed approach perceived by the developers?

7.2 Participants

Ten participants were selected among enrolled in graduate courses in the field of software design at the National Institute of Space Research. As a requirement, in order to participate in the study, the participant needs to have experience in programming using Java language and knowledge of code annotations.

Despite they were master and Ph.D. students, most of the participants at that moment works on industry and were doing the course as special students. During the analysis of the results, we divided the 10 participants into two distinct groups, where 5 participants with experience less than 10 years were assigned to Group 1, and the remaining 5 with experience equal or greater than 10 years were assigned to the Group 2. Table 2 shows some statistics about participants background in terms of years of experience as programmers; and years of experience with the Java Language.

As for participants expertise on TDD techniques, most of them—90% (9 on 10 participants)—had some experience with it. Nevertheless, all the participants received a training about TDD in order to homogenize their knowledge. This training involved 16 hours of theoretical classes and practical code exercises on the course of software design at the INPE.

7.3 Instrumentation and procedures

When preparing the activities of the study we considered two key requirements: (a) an algorithm that is easy to understand and develop; and (b) to ensure that everyone executes the task based on TDD concepts.

After the training period, the participants had two weeks to perform the task of implementing a randomized algorithm. This task was performed extra-class because of the lack of an appropriate laboratory for this activity. The time was measured by the participants themselves. To mitigate the risk of a wrong measurement, we elaborated a procedure to assist the participants during their activities. The procedure was presented to them as follows:

- Set aside a time between 1 and 2 hours when you can perform the task without interruption and seek a quiet environment where you can focus on the task.
- Use a time marker (stopwatch) and start timing the task at that time. The accuracy of the time marking can be in minutes.
- If there are unavoidable interruptions during the task, stop the time and resume when you return. Record how long you needed to interrupt your task (this will be asked later in the questionnaire).

- Stop and record the time by the time you complete all necessary steps and tests.

The minimum time per phase was not defined because this is a metric that was searched in the study. After completion of the tasks, the participants had to fill a questionnaire to answer about their experience. In the first part of the questionnaire, the participants had to answer the following topics regarding the execution of the task:

- the time spent to complete the task (without interrupt time);
- the activity had to be interrupted, and why;
- the difficulties to complete the task;
- additional information was necessary and what kind of information.

In the last part of the questionnaire, the participants had to highlight the benefits and drawbacks of the proposed approach based on their experience. In addition, they needed to report whether the framework could be used in other applications.There was also an open text field on this form in which participants could use to add any other comments.

To answer the research question RQ1, we have considered the amount of time spent completing the task by each of the participants, who were divided into two groups according to their professional experience. RQ2, RQ3, and RQ4 are answered by means of the analysis of the participants' answers to the other questions in the questionnaire.

7.4 Implementation task

The implementation task was based on the illustrative example presented in the Section 4.4. That is, a method to generate an array of "*n*" positions, with random numbers ranging from -10 to 10, whose total sum of elements will be always zero (0). This method has an input parameter the size of the array that will be generated, parameter "*n*" and a Random object (which must be used by the method to generate random numbers). In this problem, non-determinism is represented by the Random object, which at each execution will have a different and undetermined seed.

In this task, the participants should develop this function using TDD and the proposed approach with ReTest framework. The following instructions were presented to the participants as the algorithm to be implemented:

(1) Generate an integer random number between -10 and 10 using the expression "random.nextInt (21) - 10" and assign it to each position of the array;
(2) Record the sum of all numbers generated for the array;
(3) While the total sum does not equal 0, do:
    (a) Find a random array index (also using the Random class's nextInt () method);
    (b) Verify that the value of this index could be changed to a value between -10 and 10 that would make the sum closer to or equal to zero;
    (c) Verify that the value of the array at that index has the same sign as the sum and, if it does, generate a random number with the opposite sign within the range of -10 and 10, assign the array to the index and update the value of the sum (which will cause the sum value to closer to zero).
(4) Return the generated array.

Aiming to ensure that everyone follows the TDD minimum steps, a simple roadmap was presented to developers:

(1) Develop a test that ensures that the method creates an array of size 1. In this case, there is only one valid return that is 0;
(2) Implement the method with expected return;
(3) The second test introduced should ensure that the method creates an array of size 2, initially checking only if the response has the appropriate array size. At first, the test should fail because the method only returns the array of size 1;
(4) Introduce the implementation that creates an array with the size passed as a parameter (in this case, the parameter is two) and receives a random value generated within the range of -10 to 10. The test method should receive a parameter of type Random annotated with @RandomParam to be passed to the method being developed, as shown in Listing 7:

Listing 7 Example of test method for case study

```
1  @RunWith(ReTestRunner.class)
2  public class TestClass {
3    @Test
4    @ReTest(10)
5    public void testMethod(
6      Object result = nonDeterministicAlgorithm(random);
7      assertResult(result);
8    }
9  }
```

(5) Run the tests. At this point in development, the tests must be passing, but it is known that the validity of the response is not being verified;
(6) Then create an assertion helper method to check the validity of the output. This method checks if the array has the expected size, if the value of each element is in the range of -10 to 10, and if the sum of the elements is equal to zero;
(7) Modify the test code to $n = 2$ so that it uses the assertion method you created. The @ReTest annotation must be used for this test method to run 10 times;
(8) Change the method code so that the return is as expected and for all tests to run successfully;
(9) Annotate the tests with @SaveBrokenTestDataFiles and @LoadTestFromDataFiles so that from that point on, the failed tests are stored and run again;
(10) Create at least 3 more tests with @ReTest (10), for n equal to 3, 10 and 100, by varying only the n parameter of the function, updating the implementation. If necessary, create additional tests.

7.5 Results

All participants were able to complete the implementation task following the steps outlined in the procedure. Below, the results are presented according to our research questions.

7.5.1 Required experience (RQ1)

The results of the time spent (in minutes) per group are reported in Table 3. In this table, we present average values (Mean), standard deviation (StDev), minimal value (Min), median value (Median) and maximal value (Max). By examining the time spent per group,

Table 3 Descriptive statistics in terms of minutes spent

Group	Mean	StDev	Min	Median	Max
Group 1	91.2	27.8	47.0	90.0	120.0
Group 2	78.2	17.9	60.0	87.0	090.0

we noticed that the average times are close, and the difference between the two groups is small.

Figure 10 presents the boxplots that allow us graphically comparing the spent time values per groups. As shown in the boxplots, the more experienced developers spent less time on the implementation task. However, in order to verify whether such differences between the two groups are significant, we compared the samples using t-Test. The result of the t-Test (p-value $= 0.3844$) revealed no statistically significant differences at the 0.05 level of alpha. Therefore, this analysis corroborates with the statement that the use of the Re-Test framework does not require wide programming experience. As aforementioned, all participants were able to use it to complete their tasks.

7.5.2 Difficulties encountered (RQ2)

With regard to ease of use, 60% of the participants stated that the framework was easy to use, while 40% of the participants stated a medium difficulty to use it, as shown in Fig. 11. Moreover, five participants reported some type of difficulty. Their reports were analyzed and their difficulties were classified as being related to the framework, to the algorithm or to the IDE. Table 4 presents the main difficulties encountered by them. Checking the results of the table, we can notice that only two of them found some type of difficulty with the test framework.

7.5.3 Benefits and drawbacks (RQ3)

Table 5 presents the benefits that were highlighted by participants. The main benefit highlighted by 90% of participants was the possibility of executing regression tests on randomized algorithms. However, some participants recognized as benefits the possibility of executing of TDD on randomized algorithms (60%), less complexity on a test of these algorithms (70%), and the simplicity of use of the ReTest framework (50%).

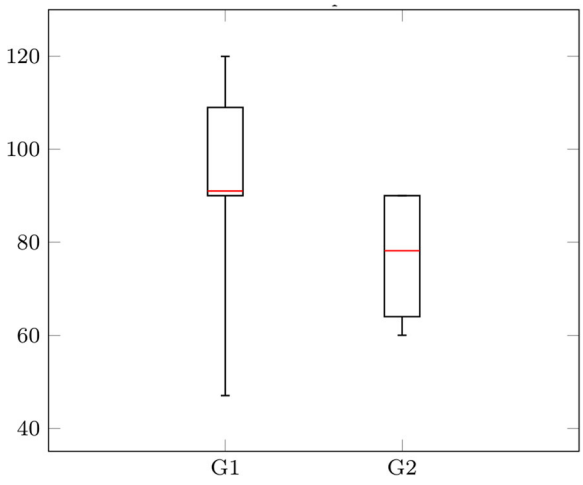

Fig. 10 Development time per group

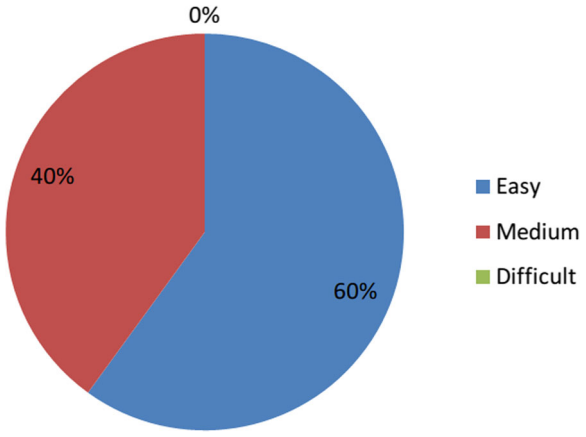

Fig. 11 Perceived ease of use

Table 6 presents the drawbacks perceived by participants. Only two participants reported drawback of the approach: the difficulty in debugging source code, and increased time to run the tests.

7.5.4 Perceived usefulness (RQ4)

At the end of the questionnaire, participants were inquired about the the usefulness of the framework. Figure 12 presents a graph of the percentage of participants who judge that can develop the proposed task using TDD without the help of the framework. Although 40% of the participants judge that have the ability to do the task without the ReTest framework, 90% of the participants stated that the framework helped a lot in the implementation task, as shown in the Fig. 13. Moreover, some participants highlighted the usefulness of the framework for other types of algorithms, such as: algorithms for aircraft

Table 4 Difficulties encountered

Type of difficulty	Participants
Difficulties with the test framework	2
Difficulties with algorithm logic	1
Difficulties with the IDE	2

Table 5 Perceived benefits

Benefits	Participants
Simplicity of use	5
Decreased complexity of randomized algorithm tests	7
Enables the use of TDD in randomized algorithms	6
Enables regression tests on randomized algorithms	9

Table 6 Perceived drawbacks

Drawbacks	Participants
Difficulty in debugging source code	1
Increased time to run the tests	1

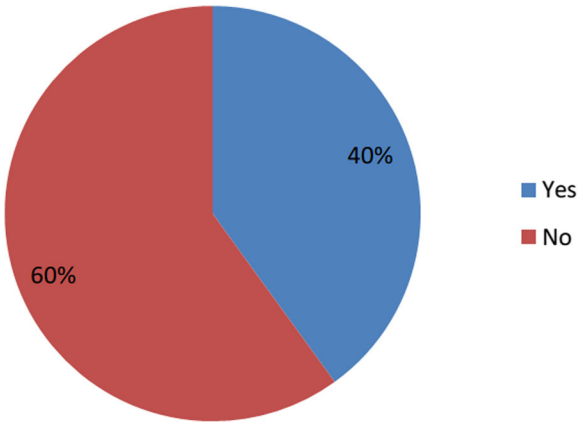

Fig. 12 Is it possible to make the proposed algorithm without ReTest?

performance, algorithms for load testing, algorithms for calculus in dynamic systems, and algorithms for simulation models with unpredictable characteristics.

8 Discussion and limitations

To date, we carried out two studies towards to evaluate our approach for applying TDD in the development of randomized algorithms. In the first study, we were able to evaluate the feasibility of the proposed approach in a real case carrying out a single-subject experiment which involved a randomized algorithm for creation and evolution of dynamic graphs.

In single-subject experimentation, the main challenges and limitations are associated with the internal and external threats to validity (Harrison 2000). Threats to internal validity concern analyses if, in fact, the treatment causes the effect (Wohlin et al. 2012). The internal threats identified in this study refer to lack of statistical analysis and the effect of subject learning in the experiment.

Statistical analysis usually does not apply in single-subject experiments because of the lack of replications. The analysis performed in this experiment was based on observing whether the effects of the treatment are important or not. However, an A-B-A design could provide stronger evidences than just an A-B design. In the additional A Phase,

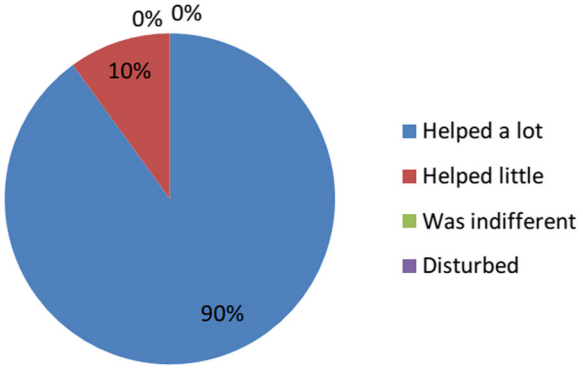

Fig. 13 TDD helped in the development of the proposed algorithm?

the treatment is removed, and another baseline phase entered to confirm the previous findings. For sure this type of design will be considered in our future single-subject experiments.

Regarding the effect of learning, the outcome of the experiment could be affected because the participant implemented the same algorithm twice, and in the second time the problem was already familiar. However, such an effect is minimized in this study, since the second code was implemented almost a year and a half after the first implementation.

Threats to external validity is concerned with the generalization of the results (Wohlin et al. 2012). Performing an experiment with a single-subject allowed us to evaluate our approach from real and complex problem in machine learning field. Having a real problem, instead of a toy one, increased the external validity of the experiment, especially with regard to algorithms with non-deterministic behavior which are rarely addressed in software engineering research. Ideally, evaluation studies should involve realistic projects with various subjects, and rigor to ensure reliability on outcomes derived from the treatment that was applied (Sjoberg et al. 2007). However, comparative group experiments can be time-consuming, expensive, and difficult to control mainly when involving various subjects. Moreover, many tasks are constructed to be small toy examples in order to observe the effects of the treatment within a short period of time (Harrison 2005). The biggest challenge to evaluate our approach, however, has been to find expert developers in the development of random or non-deterministic algorithms. The single participant of this study had the appropriate developer profile for the experiment and a real case to be applied our approach.

In the second study, we were able to assess the using the ReTest framework from developer experience. The purpose of the study was not to compare our approach with any other since we find out that most developers did not know how to apply TDD in randomized algorithms without a previous knowledge of the patterns. However, the proposed task was adequate to assess the use of the ReTest framework. Proper use of the framework requires knowledge in applying TDD and design patterns which is a fundamental prerequisite for TDD. Such requirements were filled with the training performed. Certainly, the simple task and the roadmap provide to developers also helped the participants to complete their tasks, even the developers without so much experience. Through this study, we found that most developers recognized the usefulness of the framework, reporting more benefits than drawbacks. And, only two participants reported difficulties properly using the framework.

9 Conclusion

The main contribution of this paper is to present an approach to allow the use of the TDD as development and design technique of applications involving randomized algorithms, specifically when several random choices need to be made during its execution. The proposed approach consists of an extension of TDD to enable its application for algorithms with non-deterministic characteristics, based on a set of software patterns to guide the application of this approach. In addition, a testing framework called ReTest was developed to support such algorithms in the Java programming language.

Two studies were carried out to assess the feasibility of the proposed approach applied in an algorithm implementation involving multiple random decisions in sequence, and to assess the use of the Re-Test framework from the point of view of the developers.

In the first study, the results of the single-subject experiment suggest that it is possible to find more bugs with the proposed approach because their patterns enable the use of TDD in random or non-deterministic algorithms, something that up to now had not been used. Consequently, these types of algorithms can be now covered by unit tests, including regression tests. The results also suggest that the design of randomized algorithms can be improved with the proposed approach. And, the maintainability of the algorithm was another improvement aspect identified in this experiment.

In the second study on the developers' perceptions, the results suggest that the ReTest framework is easy to use and usefulness in the opinion of the majority of participants who used it. Also, we found that despite the difficulties, all participants were able to complete the task using the proposed approach based on provided documentation.

It is worthwhile stressing that both studies results are not intended to be conclusive statements because of their limitations and the threats to validity. Thus, all of the results reported in this paper must be confirmed with further research.

Until now the results were satisfactory and indicated that the strategy is quite promising. However, further investigations are needed to verify whether such findings can be extended to cases involving (i) more complex tasks that better reproduce the needs and reality of the developers; (ii) another type of randomized algorithms in other research fields; and (iii) the measurement of objective variables to better evaluate the ReTest framework.

Endnotes

[1] https://github.com/sandyporto/DynamicCommunities

[2] https://github.com/sandyporto/DynamicCommunities/blob/master/Erros.dat

Abbreviations

ART: Adaptive Random Testing; CEMADEN: Centro Nacional de Monitoramento e Alertas de Desastres Naturais; INPE: Instituto Nacional de Pesquisas Espaciais (National Institute of Space Research); ReTest: Random Engagement for Test; RTS: Regression Test Selection; TDD: Test-Driven Development; UNIFESP: Instituto de Ciencias e Tecnologia, Universidade Federal de São Paulo; XP: Extreme Programming

Funding

This article has no funding.

Availability of data and materials

The datasets supporting the conclusions of this article are available in the GitHub repository, https://github.com/andreivo/retest and https://github.com/sandyporto/DynamicCommunities.

Authors' contributions

AASI: Development of the approach; Framework development; bibliographic research in TDD; design, conduction, and analysis of the second study; lead the paper writing. EMG: Development of the approach; advised in the development of the framework and the single-subject experiment; design and conduction of the second study; collaboration in the paper writing. SMP: Development of the dynamic communities algorithm; bibliographic research in randomized algorithms; perform the single-subject experiment; collaboration in the paper writing. JC: Review of experimental software engineering concepts; collaboration in the paper writing. MGQ: Advised in the development of the single-subject experiment; bibliographic research in randomized algorithms; collaboration in the paper writing. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Author details

[1]Centro Nacional de Monitoramento e Alertas de Desastres Naturais (CEMADEN), Estrada Dr. Altino Bondensan, 500 - Coqueiro, 12247-016 São José dos Campos - SP, Brazil. [2]Instituto Nacional de Pesquisas Espaciais (INPE), Av. dos Astronautas, 1.758 - Jardim da Granja, 12227-010 São José dos Campos - SP, Brazil. [3]Instituto de Ciencias e Tecnologia,

Universidade Federal de São Paulo (UNIFESP), Avenida Cesare Monsueto Giulio Lattes, 1201 - Coqueiro, 12247-014 São José dos Campos - SP, Brazil.

Received: 19 February 2018 **Accepted:** 22 August 2018
Published online: 18 September 2018

References

Agrawal H, Horgan JR, Krauser EW, London S (1993) Incremental regression testing. In: Proceedings of the Conference on Software Maintenance. ICSM '93. IEEE Computer Society, Washington. pp 348–357. http://dl.acm.org/citation.cfm?id=645542.658149
Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, Mcminn P (2013) An orchestrated survey of methodologies for automated software test case generation. J Syst Softw 86(8):1978–2001
Astels D (2003) Test Driven Development: A Practical Guide. Prentice Hall Professional Technical Reference
Bach J, Schroeder PJ (2004) Pairwise testing: A best practice that isn't. In: 22nd Annual Pacific Northwest Software Quality Conference. pp 180–196. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.105.3811
Baresi L, Young M (2001) Test oracles. Technical report. University of Oregon, Dept. of Computer and Information Science, Eugene
Beck K (2002) Test Driven Development. By Example (Addison-Wesley Signature). Addison-Wesley Longman, Amsterdam, Amsterdam
Beck K, Andres C (2004) Extreme Programming Explained: Embrace Change. 2nd edn. Addison-Wesley Professional
Beck K, Gamma E (2000) More java gems. Cambridge University Press, New York. Chap. Test-infected: Programmers Love Writing Tests. http://dl.acm.org/citation.cfm?id=335845.335908
Chen Y-F, Rosenblum DS, Vo K-P (1994) Testtube: A system for selective regression testing. In: Proceedings of the 16th International Conference on Software Engineering. ICSE '94. IEEE Computer Society Press, Los Alamitos. pp 211–220. http://dl.acm.org/citation.cfm?id=257734.257769
Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to Algorithms. 2nd edn. The MIT Press
Dunlop DD, Basili VR (1982) A comparative analysis of functional correctness. ACM Comput Surv 14(2):229–244
Elva R (2013) Detecting semantic method clones in java code using method ioe-behavior. Doctor of philosophy (ph.d.), College of Engineering and Computer Science - University of Central Florida. http://stars.library.ucf.edu/cgi/viewcontent.cgi?article=3620&context=etd
Fagerholm F, Münch J (2012) Developer experience: Concept and definition. In: Proceedings of the International Conference on Software and System Process. IEEE Press. pp 73–77
Floyd RW (1967) Nondeterministic algorithms. J ACM 14(4):636–644
Freeman S, Pryce N (2009) Growing Object-Oriented Software, Guided by Tests. 1st edn. Addison-Wesley Professional
Freeman S, Mackinnon T, Pryce N, Walnes J (2004) Mock roles, not objects. In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications. OOPSLA '04. ACM, New York. pp 236–246. http://doi.acm.org/10.1145/1028664.1028765
Galbraith SD (2012) Mathematics of Public Key Cryptography. 1st edn. Cambridge University Press, New York
Guerra E, Aniche M (2016) Chapter 9 - achieving quality on software design through test-driven development. In: Mistrik I, Soley R, Ali N, Grundy J, Tekinerdogan B (eds). Software Quality Assurance. Morgan Kaufmann, Boston. pp 201–220
Harrison W (2000) N=1: an alternative for software engineering research. In: Beg, Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Engineering Research, Workshop, vol 5. Citeseer. pp 39–44
Harrison, W (2005) Skinner wasn't a software engineer. IEEE Softw 22(3):5–7
Harrold MJ, Souffa ML (1988) An incremental approach to unit testing during maintenance. In: Proceedings. Conference on Software Maintenance, 1988. pp 362–367
Hartmann J, Robson DJ (1990) Techniques for selective revalidation. IEEE Softw 7:31–36
Ivo A, Guerra EM (2017) A set of patterns to assist on tests of non-deterministic algorithms. In: Proceedings, Vancouver. Conference on Pattern Languages of Programs, 24. (PLoP)
Ivo AAS, Guerra EM (2017) Retest: framework for applying tdd in the development of non-deterministic algorithms. In: Silva TSd, Estácio B, Kroll J, Fontana RM (eds). Agile Methods: 7th Brazilian Workshop, WBMA 2016, Curitiba, Brazil, November 7-9, 2016. Springer, Curitiba-Brazil. pp 72–84. https://doi.org/10.1007/978-3-319-55907-0_7
Kim J-M, Porter A, Rothermel G (2000) An empirical study of regression test application frequency. In: Proceedings of the 2000 International Conference on Software Engineering, 2000. pp 126–135
Mackinnon T, Freeman S, Craig P (2001) Extreme programming examined. Addison-Wesley Longman Publishing Co., Inc., Boston. Chap. Endo-testing: Unit Testing with Mock Objects. http://dl.acm.org/citation.cfm?id=377517.377534
Martin RC (2003) Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River
Porto S, Quiles MG (2014) A methodology for generating time-varying complex networks with community structure. In: Murgante B, Misra S, Rocha AMAC, Torre C, Rocha JG, Falcão MI, Taniar D, Apduhan BO, Gervasi O (eds). Computational Science and Its Applications – ICCSA 2014. Springer, Cham. pp 344–359
R Core Team (2015) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna. https://www.R-project.org/
Rothermel G, Harrold MJ (1997) A safe, efficient regression test selection technique. ACM Trans Softw Eng Methodol 6(2):173–210
Sjoberg DI, Dyba T, Jorgensen M (2007) The future of empirical methods in software engineering research. In: Future of Software Engineering, 2007. FOSE'07. IEEE. pp 358–378
Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A (2012) Experimentation in Software Engineering. Springer
Xiaowen L (2013) Research on regression testing methods for industry applications. Int J Smart Home 7(6):111–122
Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: A survey. Softw Test Verif Reliab 22(2):67–120