Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects

Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, Martin Monperrus

Noname manuscript No.
(will be inserted by the editor)

Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects

Benjamin Danglot · Oscar Luis Vera-Pérez · Benoit Baudry · Martin Monperrus

March 2018

Abstract In the literature, there is a rather clear segregation between manually written tests by developers and automatically generated ones. In this paper, we explore a third solution: to automatically improve existing test cases written by developers. We present the concept, design and implementation of a system called DSpot, that takes developer-written test cases as input (JUnit tests in Java) and synthesizes improved versions of them as output. Those test improvements are given back to developers as patches or pull requests, that can be directly integrated in the main branch of the test code base. We have evaluated DSpot in a deep, systematic manner over 40 real-world unit test classes from 10 notable and open-source software projects. We have amplified all test methods from those 40 unit test classes. In 26/40 cases, DSpot is able to

B. Danglot
Inria Lille - Nord Europe
Parc scientifique de la Haute Borne
40, avenue Halley - Bt A - Park Plaza
59650 Villeneuve d'Ascq - France
E-mail: danglot@inria.fr

O. Vera-Pérez
Inria Rennes - Bretagne Atlantique
Campus de Beaulieu, 263 Avenue Gnral Leclerc
35042 Rennes - France
E-mail: oscar.vera-perez@inria.fr

B. Baudry
KTH Royal Institute of Technology in Stockholm
Brinellvgen 8
114 28 Stockholm - Sweden
E-mail: baudry@kth.se

M. Monperrus
KTH Royal Institute of Technology in Stockholm
Brinellvgen 8
114 28 Stockholm - Sweden
E-mail: martin.monperrus@csc.kth.se

automatically improve the test under study, by triggering new behaviors and adding new valuable assertions. Next, for ten projects under consideration, we have proposed a test improvement automatically synthesized by DSpot to the lead developers. In total, 13/19 proposed test improvements were accepted by the developers and merged into the main code base. This shows that DSpot is capable of automatically improving unit-tests in real-world, large scale Java software.

1 Introduction

Over the last decade, strong unit testing has become an essential component of any serious software project, whether in industry or academia. The agile development movement has contributed to this cultural change with the global dissemination of test-driven development techniques [6]. More recently, the DevOps movement has further strengthened the testing practice with an emphasis on continuous and automated testing [24].

In this paper we study how such modern test suites can benefit from the major results of automatic test generation research. We explore whether one can automatically improve tests written by humans, an activity that can be called "automatic test improvement". There are few works in this area: the closest related techniques are those that consider manually written tests as the starting point for an automatic test generation process [18,11,29,31,8, 30]. To this extent, automatic test improvement can be seen as forming a sub-field of test generation. Automatic test improvement aims at synthesizing modifications of existing test cases, where those modifications are meant to be presented to developers. As such, the modifications must be deemed relevant by the developers themselves (the corollary being that they should not only maximize some criterion).

For our original study of automatic test improvement, we have developed DSpot, a tool for automatic test improvement in Java. DSpot adapts and combines two notable test generation techniques: evolutionary test construction [25] and regression oracle generation [27]. The essential adaptation consists in starting the generation process from the full-fledged abstract syntax trees of manually written test cases. The combination of both techniques is essential so that changes in the setup together are captured by changes in the assertion part of tests.

Our study considers 10 mature Java open source projects. It focuses on three points that have little, or never, been assessed. First, we propose 19 test improvements generated by DSpot to the developers of the considered open source projects. We present them the improvement in the form of pull requests, and we ask them whether they would like to merge the test improvements in the main repository. In this part of the study, we extensively discuss their feedback, to help the research community understand the nature of good test improvements. This reveals the key role of case studies, as presented by Flyvberg [10], to assess the relevance of our technique for developers. Second, we

perform a quantitative assessment of the improvements of 40 real-world test classes from our set of 10 open-source projects. In particular, we consider the difficult case of improving strong test classes. Third, we explore the relative contribution of evolutionary test construction and of assertion generation in the improvement process.

Our key results are as follows: first, seven GitHub pull requests consisting of automatic test improvements have been definitively accepted by the developers; second, an interesting empirical fact is that DSpot has been able to improve a test class with a 99% initial mutation score (*i.e.* a really strong test); and finally, our experiment shows that valuable test improvements can be obtained within minutes.

To sum up, our contributions are:

- DSpot, a system that performs automatic test improvement of Java unit tests;
- the design and execution of an experiment to assess the relevance of automatically improved tests, based on feedback from the developers of mature projects;
- a large scale quantitative study of the improvement of 40 real-world test classes taken from 10 mature open-source Java projects.
- fully open-science code and data: both DSpot[1] and our experimental data are made publicly available for future research[2]

The remainder of this article is as follows. Section 2 presents the main concepts of automatic test improvement and DSpot. Section 3 presents the experimental protocol of our study. Section 4 analyses our empirical results. Section 5 discusses the threats to validity. Section 6 discusses the related work. and Section 7 concludes the article. Note that a previous version of this paper can be found as Arxiv's working paper [4].

2 Automatic Test Improvement

In this section, we present the concept of automated test improvement, and its realization in the DSpot tool.

2.1 Goal

The goal of automatic test improvement is to synthesize modifications to existing test cases to increase test quality. These modifications are meant to be given to developers and committed to the main test code repository. The quality assessment is driven by a specific test criterion such as branch coverage or mutation score. In this paper, we focus on improving the mutation score of an existing test suite but automatic test improvement is more general and it is not bound to the mutation score.

[1] https://github.com/STAMP-project/dspot/

[2] https://github.com/STAMP-project/dspot-experiments/

2.2 DSpot

DSpot is an automatic test improvement tool for Java unit tests. It is built upon the algorithms of Tonella [25] and Xie [28].

2.2.1 DSpot inputs

The input of DSpot consists in a set of existing test cases, manually written by the developers. As output, DSpot produces variants of the given test cases. These variants are meant to be added to the test suite. By putting together existing test cases and their variants, we aim at strictly improving the overall test suite quality. By construction, the enhanced test suite is at least as good, or better than the original one w.r.t. the considered criterion.

Concretely, DSpot synthesizes suggestions in the form of diffs that are proposed to the developer: Figure 1 shows such a test improvement.

```
146         ByteArrayOutputStream out = new ByteArrayOutputStream();
    -       writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
147 +       final int bytesWritten = writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
148 +       assertEquals(0, bytesWritten);
149         byte[] data = out.toByteArray();
```

Fig. 1 Example of what DSpot produces: a diff to improve an existing test case.

DSpot is an automatic test improvement system because it only modifies existing test cases. As such, all test improvements, by construction, are modifications to existing test cases. DSpot's novelty is twofold: 1) first, it combines those algorithms in a way that scales to modern, large Java software 2) second, it makes no assumption on the tests to be improved, and works with any arbitrary JUnit test.

2.2.2 DSpot's Workflow

The main workflow of DSpot is composed of 2 main phases: 1) the transformation of the test code to create new test inputs inspired by Tonella's technique, we call this "input space exploration"; this phase consists in changing new test values and objects and adding new method calls, the underlying details will be explained in details in subsubsection 2.4.1. 2) the addition of new assertions per Xie's technique [28], we call this phase "assertion improvement". The behavior of the system under test is considered as the oracle of the assertion, see subsubsection 2.4.2. In DSpot, the combination of both techniques, *i.e.* the combination of input space exploration and assertion improvement is called "test amplification".

DSpot keeps the modifications that add the most value for the developers. To do so, DSpot uses the mutation score as a proxy to the developers assessed value of quality. In essence, developers value changes in test code if they enable

```
testIterationOrder() {
  // contract: the iteration order is the same as the insertion order

  TreeList tl=new TreeList();
  tl.add(1);
  tl.add(2);

  ListIterator it = tl.listIterator();

  // assertions
  assertEquals(1, it.next().intValue());
  assertEquals(2, it.next().intValue());
}
```

Listing 1 An example of an object-oriented test case (inspired from Apache Commons Collections)

them to catch new bugs, that is if the improved test better specifies a piece of code. This is also reflected in the mutation score: if the mutation score increases, it means that a code element, say a statement, is better specified than before. In other words, DSpot uses the mutation score to steer the test case improvement, following the original conclusions of DeMillo *et al.* who observed that mutants provide hints to generate test data [9]. To sum up, DSpot aims at producing better tests that have a higher potential to catch bugs.

2.3 Definitions

We first define the core terminology of DSpot in the context of object-oriented Java programs.

Test suite is a set of test classes.

Test class is a class that contains test methods. A test class is neither deployed nor executed in production.

Test method or **test case** is a method that sets up the system under test into a specific state and checks that the actual state at the end of the method execution is the expected state.

Unit test is a test method that specifies a targeted behavior of a program. Unit tests are usually independent of each other and execute a small portion of the code, i.e. a single unit or a single component of the whole system.

2.3.1 Modern Test Cases

DSpot improves the test cases of modern Java programs, which are typically composed of two parts: input setup and assertions. The input setup part is responsible for driving the program into a specific state. For instance, one creates objects and invokes methods on them to produce a specific state.

The assertion part is responsible for assessing that the actual behavior of the program corresponds to the expected behavior, the latter being called the oracle. To do so, the assertion uses the state of the program, *i.e.* all the observable values of the program, and compare it to expected values written by developers. If the actual observed values of the program state and the oracle are different (or if an exception is thrown), the test fails and the program is considered as incorrect.

Listing 1 illustrates an archetypal example of such a test case: first, from line 4 to line 6, the test input is created through a sequence of object creations and method calls; then, at line 8, the tested behavior is actually triggered; the last part of the test case at 11 and 12, the assertion, specifies and checks the conformance of the observed behavior with the expected one. We note that this notion of call sequence and complex objects is different from test inputs consisting only of primitive values.

2.4 Algorithms

2.4.1 Input Space Exploration Algorithm

DSpot aims at exploring the input space so as to set the program in new, never explored states. To do so, DSpot applies code transformations to the original manually-written test methods.

I-Amplification: *I-Amplification*, for Input Amplification, is the process of automatically creating new test input points from existing test input points.

DSpot uses three kinds of *I-Amplification*.

1) Amplification of literals: the new input point is obtained by changing a literal used in the test (numeric, boolean, string). For numeric values, there are five operators: $+1$, -1, $\times 2$, $\div 2$, and replacement by an existing literal of the same type, if such literal exists. For Strings, there are four operators: add a random char, remove a random char, replace a random char and replace the string by a fully random string of the same size. For booleans, there is only one operator: negate the value;

2) Amplification of method calls: DSpot manipulates method calls as follows: DSpot duplicates an existing method call; removes a method call; or adds a new invocation for an accessible method with an existing variable as target.

3) Test objects: if a new object is needed as a parameter while amplifying method calls, DSpot creates a new object of the required type using the default constructor if it exists. In the same way, when a new method call needs primitive value parameters, DSpot generates a random value.

DSpot combines the different kinds of *I-Amplification* iteratively: at each iteration all kinds of *I-Amplification* are applied, resulting in new tests. From one iteration to another, DSpot reuses the previously amplified tests, and further applies *I-Amplification*.

```
testIterationOrder() {
  TreeList tl=new TreeList();
  tl.add(1);
  tl.add(2);
  tl.removeAll(); // method call added

  // removed assertions
}
```

Listing 2 An example of an *I-Amplification*: the amplification added a method call to *removeAll()* on *tl*.

For example, if we apply an *I-Amplification* on the example presented in Listing 1, it may generate a new method call on *tl*. In Listing 2, the added method call is "removeAll". Since DSpot changes the state of the program, existing assertions may fail. That is why it removes also all existing assertions.

2.4.2 Assertion Improvement Algorithm

To improve existing tests, DSpot adds new assertions as follows.

***A-Amplification*:** *A-Amplification*, for Assertion Amplification, is the process of automatically creating new assertions.

In DSpot, assertions are added on objects from the original test case, as follows: 1) it instruments the test cases to collect the state of a program after execution (but before the assertions), *i.e.* it creates observation points. The state is defined by all values returned by getter methods. 2) it runs the instrumented test to collect the values, the result of this execution is a map that gives, for each test case object, the values from all getters. 3) it generates new assertions in place of the observation points, using the collected values as oracle. The collected values are used as expected values in the new assertions. In addition, when a new test input sets the program in a state that throws an exception, DSpot produces a test asserting that the program throws a specific exception.

For example, let consider *A-Amplification* on the test case of the example above.

First, in Listing 3 DSpot instruments the test case to collect values, by addding method calls to the objects involved in the test case.

Second, the test with the added observation points is executed, and subsequently, DSpot generates new assertions based on the collected values. On Listing 4, we can see that DSpot has generated two new assertions.

2.4.3 Test Improvement Algorithm

Algorithm 1 shows the main loop of DSpot. DSpot takes as input a program P and its Test Suite TS. DSpot also uses an integer n that defines the number of iterations. DSpot produces an Amplified Test Suite ATS, *i.e.* a

```
1  testIterationOrder() {
2    TreeList tl=new TreeList();
3    tl.add(1);
4    tl.add(2);
5    tl.removeAll();
6
7    // logging current behavior
8    Observations.observe(tl.size());
9    Observations.observe(tl.isEmpty());
10 }
```

Listing 3 In *A-Amplification*, the second step is to instrument and run the test to collect runtime values.

```
1  testIterationOrder() {
2    TreeList tl=new TreeList();
3    tl.add(1);
4    tl.add(2);
5    tl.removeAll();
6
7    // generated assertions
8    assertEquals(0, tl.size()); // generated assertions
9    assertTrue(tl.isEmpty()); // generated assertions
10 }
```

Listing 4 In *A-Amplification*, the last step is to generate the assertions based on the collected values.

better version of the input Test Suite TS according to a specific test criterion such as mutation score. For each test case t in the test suite TS (Line 1), DSpot first tries to add assertions without generating any new test input (Line 3), method $generateAssertions\,(t)$ is explained in subsubsection 2.4.2. Note that adding missing assertions is the elementary way to improve existing tests.

DSpot initializes a temporary list of tests TMP and applies n times the following steps (Line 6): 1) it applies each amplifier *amp* on each tests of TMP to build V (Line 8-9 see subsubsection 2.4.1 *i.e. I-Amplification*); 2) it generates assertions on generated tests in V (Line 11 see subsubsection 2.4.2, *i.e. A-Amplification*); 3) it keeps the tests that improve the mutation score (Line 12). 4) it assigns V to TMP for the next iteration. This is done because even if some amplified test methods in V have not been selected, it can contain amplified test methods that will eventually be better in subsequent iterations.

2.4.4 Flaky tests elimination

The input space exploration (see subsubsection 2.4.1) may produce test inputs that results in non-deterministic executions. This means that, between two independent executions, the state of the program is not the same. Since DSpot generates assertions where the expected value is a hard coded value from a

Algorithm 1 Main amplification loop of DSpot.

Input: Program P
Input: Test Suite TS
Input: Amplifiers $amps$ to generate new test data input
Input: n number of iterations of DSpot's main loop
Output: An Amplified Test Suite ATS

```
1: ATS ← ∅
2: for t in TS do
3:     U ← generateAssertions(t)
4:     ATS ← {x ∈ U|x improves mutation score}
5:     TMP ← ATS
6:     for i = 0 to n do
7:         V ← []
8:         for amp in amps do
9:             V ← V ∪ amp.apply(TMP)
10:        end for
11:        V ← generateAssertions(V)
12:        ATS ← ATS ∪ {x ∈ V|x improves mutation score}
13:        TMP ← V
14:    end for
15: end for
16: return ATS
```

specific run (see subsubsection 2.4.2), the generated test case may become flaky: it passes or fails depending on the execution and whether the expected value is obtained or not.

To avoid such flaky tests generated by DSpot, we run n times each new test case resulting from amplification ($n = 3$ in the default configuration). If a test fails at least once, DSpot throws it away. We acknowledge that this procedure does not guarantee the absence of flakiness. However, it gives incremental confidence: if the user wants more confidence, she can tell DSpot to run the amplified tests more times.

2.4.5 Selecting Focused Test Cases

DSpot sometimes produces many tests, from one initial test. Due to limited time, the developer needs to focus on the most interesting ones. To select the test methods that are the most likely to be merged in the code base, we implement the following heuristic. First, the amplified test methods are sorted according to the ratio of newly killed mutants and the total number of test modifications. Then, in case of equality, the methods are further sorted according to the maximum numbers of mutants killed in the same method.

The first criterion means that we value short modifications. The second criterion means that the amplified test method is focused and tries to specify one specific method inside the code.

If an amplified test method is merged in the code base, we consider that the corresponding method as specified. In that case, we do not take into account other amplified test methods that specify the same method.

Finally, in this ordered list, the developer is recommended the amplified tests that are focused, where focus is defined as where at least 50% of the newly killed mutants are located in a single method. Our goal is to select amplified tests which intent can be easily grasped by the developer: the new test specifies the method.

2.5 Implementation

DSpot is implemented in Java. It consists of 8800+ logical lines of code (as measured by cloc). For the sake of open-science, DSpot is made publicly available on Github[3]. DSpot uses Spoon[20] to analyze and transform the tests of the software application under amplification.

In this paper, we aim at improving the mutation score of test classes. In DSpot, we use Pitest[4] because: 1) it targets Java programs, 2) it is mature and well-regarded, 3) it has an active community.

An important feature of Pitest is that if the application code remains unchanged, the generated mutants are always the same. This property is very interesting for test amplification. Since DSpot only modifies test code, this feature allows us to compare the mutation score of the original test case against the mutation score of the amplified version and even compare the absolute number of mutants killed by both test case variants. We will exploit this feature in our evaluation.

By default, DSpot uses all the mutation operators available in Pitest: conditionals boundary mutator; increments mutator; invert negatives mutator; math mutator; negate conditionals mutator; return values mutator; void method calls mutator.

3 Experimental Protocol

Automatic test improvement has been evaluated with respect to evolutionary test inputs [25] and new assertions [28]. However: 1) the two topics have never been studied in conjunction 2) they have never been studied on large modern Java programs 3) most importantly, the quality of improved tests has never been assessed by developers.

We set up a novel experimental protocol that addresses those three points. First, the experiment is based on DSpot, which combines test input exploration and assertion generation. Second, the experiment is made on 10 active GitHub projects. Third, we have proposed improved tests to developers under the form of pull-requests.

We answer the following research questions:

[3] `https://github.com/STAMP-project/dspot`

[4] We use the latest version released: 1.2.0.`https://github.com/hcoles/pitest/releases/tag/1.2.0`

RQ1: Are the improved test cases produced by DSpot relevant for developers? Are the developers ready to permanently accept the improved test cases into the test repository?
RQ2: To what extent are improved test methods considered as focused?
RQ3: To what extent do the improved test classes increase the mutation score of the original, manually-written, test classes?
RQ4: What is the relative contribution of *I-Amplification* and *A-Amplification* to the effectiveness of automatic test improvement?

3.1 Dataset

We evaluate DSpot by amplifying test classes of large-scale, notable, open-source projects. We include projects that fulfill the following criteria: 1) the project must be written in Java; 2) the project must have a test suite based on JUnit; 3) the project must be compiled and tested with Maven; 4) the project must have an active community as defined by the presence of pull requests on GitHub, see subsection 4.1.

Table 1 Dataset of 10 active Github projects considered on our relevance study (RQ1) and quantitative experiments (RQ2, RQ3).

project	description	# LOC	# PR	considered test classes
javapoet	Java source file generator	3150	93	TypeNameTest^h NameAllocatorTest^h FieldSpecTest^l ParameterSpecTest^l
mybatis-3	Object-relational mapping framework	20683	288	MetaClassTest^h ParameterExpressionTest^h WrongNamespacesTest^l WrongMapperTest^l
traccar	Server for GPS tracking devices	32648	373	GeolocationProviderTest^h MiscFormatterTest^h ObdDecoderTest^l At2000ProtocolDecoderTest^l
stream-lib	Library for summarizing data in streams	4767	21	TestLookup3Hash^h TestDoublyLinkedList^h TestICardinality^l TestMurmurHash^l
mustache.java	Web application templating system	3166	11	ArraysIndexesTest^h ClasspathResolverTest^h ConcurrencyTest^l AbstractClassTest^l
twilio-java	Library for communicating with Twilio REST API	54423	87	RequestTest^h PrefixedCollapsibleMapTest^h AllTimeTest^l DailyTest^l
jsoup	HTML parser	10925	72	TokenQueueTest^h CharacterReaderTest^h AttributeTest^l AttributesTest^h
protostuff	Data serialization library	4700	35	TailDelimiterTest^h LinkBufferTest^h CodedDataInputTest^l CodedInputTest^h
logback	Logging framework	15490	104	FileNamePatternTest^h SyslogAppenderBaseTest^h FileAppenderResilience_AS_ROOT_Test^l Basic^l
retrofit	HTTP client for Android.	2743	249	RequestBuilderAndroidTest^h CallAdapterTest^h ExecutorCallAdapterFactoryTest^h CallTest^h

We implement those criteria as a query on top of TravisTorrent [7]. We randomly selected 10 projects from the result of the query which produces, the dataset presented in Table 1. This table gives the project name, a short description, the number of pull-requests on GitHub (#PR), and the considered test classes. For instance, javapoet is a strongly-tested and active project, which implements a Java file generator, it has had 93 pull-requests in 2016.

3.2 Test Case Selection Process for Test-suite Improvement

For each project, we select 4 test classes to be amplified. Those test classes are chosen as follows.

First, we select unit-test classes only, because our approach focuses on unit test amplification. We use the following heuristic to discriminate unit test cases from others: we keep a test class if it executes less than an arbitrary threshold of N statements, *i.e.* if it covers a small portion of the code. In our experiment, we use $N = 1500$ based on our initial pilot experiments.

Among the unit-tests, we select 4 classes as follows. Since we want to analyze the performance of DSpot when it is provided with both good and bad tests, we select two groups of classes: one group with strong tests, one other group with low quality tests. We use the mutation score to distinguish between good and bad test classes. Accordingly, our selection process has five steps: 1) we compute the original mutation score of each class with Pitest (see subsection 2.5; 2) we discard test classes that have 100% mutation score, because they can already be considered as perfect tests (this is the case for eleven classes, showing that the considered projects in our dataset are really well-tested projects); 3) we sort the classes by mutation score (see subsection 3.3), in ascending order; 4) we split the set of test classes into two groups: high mutation score ($> 50\%$) and low mutation score ($< 50\%$); 5) we randomly select 2 test classes in each group.

This selection results with 40 test classes: 24 in high mutation group score and 16 in low mutation score group. The imbalance is due to the fact that there are three projects really well tested for which there are none or a single test class with a low mutation score (projects protostuff, jsoup, retrofit). Consequently, those three projects are represented with 3 or 4 well-tested classes (and 1 or 0 poorly-tested class). In Table 1, the last column contains the name of the selected test classes. Each test class name is indexed by a “h” or a “l” which means respectively that the class have a high mutation score or a low mutation score.

3.3 Metrics

We use the following metrics during our experiment.

Number of Killed Mutants ($\#Killed.Mutants$): is the absolute number of mutants killed by a test class. We use it to compare the fault detection power of an original test class and the one of its amplified version.

Mutation Score: is the percentage of killed mutants over the number of executed mutants. Mathematically, it is computed as follow:

$$\frac{\#Killed.Mutants}{\#Exec.Mutants}$$

.

Increase Killed: is the relative increase of the number of killed mutants by an original test class T and the number of killed mutants by its amplified version T_a. It is computed as follows:

$$\frac{\#Killed.Mutants_{T_a} - \#Killed.Mutants_T}{\#Killed.Mutants_T}$$

The goal of DSpot is to improve tests such that the number of killed mutants increases.

3.4 Methodology

Our experimental protocol is designed to study to what extent the test improvements are valuable for the developer.

- **RQ1** To answer to RQ1, we create pull-request on notable open-source projects. We automatically improve 19 test classes of real world applications and propose one test improvement to the main developers of each project under consideration. We propose the improvement as a pull request on GitHub. A PR is composed of a title, a short text that describes the purpose of changes and a set of code change (aka a patch). The main developers review, discuss and decide to merge or not each pull request. We base the answer on the subjective and expert assessment from projects' developers. If a developer merges an improvement synthesized by DSpot, it validates the relevance of DSpot. The more developers accept and merge test improvements produced by DSpot into their test suite, the more the amplification is considered successful.
- **RQ2** To answer RQ2, we compute the number of suggested improvements, to verify that the developer is not overwhelmed with suggestions. We compute the number of focused amplified test cases, per the technique described in subsubsection 2.4.5, for each project in the benchmark. We present and discuss the proportion of focused tests out of all proposed amplified tests.
- **RQ3** To answer RQ3, we see whether the value that is taken as proxy to the developer value – the mutation score – is appropriately improved. For 40 real-world classes, we first run the mutation testing tool Pitest (see subsection 2.5) on the test class. This gives the number of killed mutants for this original class. Then, we amplify the test class under consideration and we compute the new number of killed mutants after amplification. Finally, we compare and analyze the results.

– **RQ4** To answer RQ4, we compute the number of *A-Amplification* and *I-Amplification* amplifications. The former means that the suggested improvement is very short hence easy to be accepted by the developer while the latter means that more time would be required to understand the improvement. First, we collect three series of metrics: 1) we compute number of killed mutants for the original test class; 2) we improve the test class under consideration using only *A-Amplification* and compute the new number of killed mutants after amplification; 3) we improve the test class under consideration using *I-Amplification* as well as *A-Amplification* (the standard complete DSpot workflow) and compute the number of killed mutants after amplification. Then, we compare the increase of mutation score obtained by using *A-Amplification* only and *I-Amplification* + *A-Amplification*.[5]

Research questions 3 and 4 focus on the mutation score to assess the value of amplified test methods. This experimental design choice is guided by our approach to select "focused" test methods, which are likely to be selected by the developers (described in subsubsection 2.4.5). Recall that the number of killed mutants by the amplified test is the key focus indicator. Hence, the more DSpot is able to improve the mutation score, the more likely we are to find good candidates for the developers.

4 Experimental Results

We first discuss how automated test improvements done by DSpot are received by developers of notable open-source projects (RQ1). Then, RQ2, RQ3 and RQ4 are based on a large scale quantitative experiments over 40 real-world test classes, whose main results are reported in Table 5. For the sake of open-science, all experimental results are made publicly available online: `https://github.com/STAMP-project/dspot-experiments/`.

4.1 Answer to RQ1

RQ1: Would developers be ready to permanently accept automatically improved test cases into the test repository?

4.1.1 Process

In this research question, our goal is to propose a new test to the lead developers of the open-source projects under consideration. The improved test is proposed through a "pull-request", which is a way to reach developers with patches on collaborative development platforms such as Github.

[5] Note that the relative contribution of *I-Amplification* cannot be evaluated alone, because as soon as we modify the inputs in a test case, it is also necessary to change and improve the oracle (which is the role of *A-Amplification*).

Table 2 Overall result of the opened pull request built from result of DSpot.

project	# opened	# merged	# closed	# under discussion
javapoet	4	4	0	0
mybatis-3	2	2	0	0
traccar	2	1	0	1
stream-lib	1	1	0	0
mustache	2	2	0	0
twilio	2	1	0	1
jsoup	2	0	1	1
prostostuff	2	2	0	0
logback	2	0	0	2
retrofit	0	0	0	0
total	19	13	1	5

In practice, short pull requests (*i.e.* with small test modifications) with clear purpose, *i.e.* what for it has been opened, have much more chance of being reviewed, discussed and eventually merged. So we aim at providing improved tests which are easy to review. As shown in subsubsection 2.4.1, DSpot generates several amplified test cases, and we cannot propose them all to the developers. To select the new test case to be proposed as a pull request, we look for an amplified test that kills mutants located in the same method. From the developer's viewpoint, it means that the intention of the test is clear: it specifies the behavior provided by a given method or block.

The selection of amplified test methods is done as described in subsubsection 2.4.5. For each selected method, we compute and minimize the diff between the original method and the amplified one and then we submit the diff as a pull request. A second point in the preparation of the pull request relates to the length of the amplified test: once a test method has been selected as a candidate pull request, we make the diff as concise as possible for the review to be fast and easy.

4.1.2 Overview

In total, we have created 19 pull requests, as shown in Table 2. In this table, the first column is the name of the project, the second is number of opened pull requests, *i.e.* the number of amplified test methods proposed to developers. The third column is the number of amplified test methods accepted by the developers and permanently integrated in their test suite. The fourth column is the number of amplified test methods rejected by the developers. The fifth column is the number of pull requests that are still being discussed, *i.e.* nor merged nor closed. (This number might change over time if pull-requests are merged or closed.)

Overall 13 over 19 have been merged. Only 1 has been rejected by developers. There are 5 under discussion. In the following, we perform a manual analysis of one pull-request per project. Table 4.1.2 contains the URLs of pull requests proposed in this experimentation.

Table 3 List of URLs to the pull-requests created in this experiment.

project	pull request urls
javapoet	https://github.com/square/javapoet/pull/669 https://github.com/square/javapoet/pull/668 https://github.com/square/javapoet/pull/667 https://github.com/square/javapoet/pull/544
mybatis-3	https://github.com/mybatis/mybatis-3/pull/1331 https://github.com/mybatis/mybatis-3/pull/912
traccar	https://github.com/traccar/traccar/pull/2897 https://github.com/traccar/traccar/pull/4012
stream-lib	https://github.com/addthis/stream-lib/pull/128
mustache	https://github.com/spullara/mustache.java/pull/210 https://github.com/spullara/mustache.java/pull/186
twilio	https://github.com/twilio/twilio-java/pull/437 https://github.com/twilio/twilio-java/pull/334
jsoup	https://github.com/jhy/jsoup/pull/1110 https://github.com/jhy/jsoup/pull/840
protostuff	https://github.com/protostuff/protostuff/pull/250 https://github.com/protostuff/protostuff/pull/212
logback	https://github.com/qos-ch/logback/pull/424 https://github.com/qos-ch/logback/pull/365

We now present one case study per project of our dataset.

4.1.3 javapoet

We have applied DSpot to amplify `TypeNameTest`. DSpot synthesizes a single assertion that kills 3 more mutants, all of them at line 197 of the equals method. A manual analysis reveals that this new assertion specifies a contract for the method `equals()` of objects of type `TypeName`: the method must return false when the input is null. This contract was not tested.

Consequently, we have proposed to the Javapoet developers the following one liner pull request [6]:

```
181        assertThat(a.hashCode()).isEqualTo(b.hashCode());
182  +     assertFalse(a.equals(null));
```

The title of the pull resuest is: "*Improve test on TypeName*" with the following short text: "*Hello, I open this pull request to specify the line 197 in the equals() method of com.squareup.javapoet.TypeName. if (o == null) return false;*" This test improvement synthesized by DSpot has been merged by of the lead developer of javapoet one hour after its proposal.

[6] https://github.com/square/javapoet/pull/544

4.1.4 mybatis-3

In project mybatis-3, We have applied DSpot to amplify a test for `MetaClass`. DSpot synthesizes a single assertion that kills 8 more mutants. All new mutants killed are located between lines 174 and 179, *i.e.* the `then` branch of an `if-statement` in method `buildProperty(String property, StringBuilder sb)` of `MetaClass`. This method builds a String that represents the property given as input. The `then` branch is responsible to build the String in case the `property` has a child, *e.g.* the input is "richType.richProperty". This behavior is not specified at all in the original test class.

We have proposed to the developers the following pull request entitled "*Improve test on MetaClass*" with the following short text: "*Hello, I open this pull request to specify the lines 174-179 in the buildProperty(String, StringBuilder) method of MetaClass.*" [7]:

```
68 +    assertEquals("richType.richProperty", meta.findProperty("richType.richProperty", false));
69 +
70      assertFalse(meta.hasGetter("[0]"));
```

The developer accepted the test improvement and merged the pull request the same day without a single objection.

4.1.5 traccar

We have applied DSpot to amplify `ObdDecoderTest`. It identifies a single assertion that kills 14 more mutants. All newly killed mutants are located between lines 60 to 80, *i.e.* in the method `decodesCodes()` of `ObdDecoder`, which is responsible to decode a `String`. In this case, the pull request consists of a new test method because the new assertions do not fit with the intent of existing tests. This new test method is proposed into `ObdDecoderTest`, which is the class under amplification. The PR was entitled "*Improve test cases on ObdDecoder*" with the following description: "*Hello, I open this pull request to specify the method decodeCodes of the ObdDecoder*". [8]

[7] https://github.com/mybatis/mybatis-3/pull/912/files
[8] https://github.com/tananaev/traccar/pull/2897

```
17        }
18
19  +     @Test
20  +     public void testDecodeCodes() throws Exception {
21  +         Assert.assertEquals("P0D14", ObdDecoder.decodeCodes("0D14").getValue());
22  +         Assert.assertEquals("dtcs", ObdDecoder.decodeCodes("0D14").getKey());
23  +     }
24  +
25    }
```

The developer of traccar thanked us for the proposed changes and merged it the same day.

4.1.6 stream-lib

We have applied DSpot to amplify `TestMurmurHash`. It identifies a new test input that kills 15 more mutants. All newly killed mutants are located in method `hash64()` of `MurmurHash` from lines 158 to 216. This method computes a hash for a given array of byte. The PR was entitled "*Test: Specify hash64*" with the following description: "*The proposed change specifies what the good hash code must be. With the current test, any change in "hash" would still make the test pass, incl. the changes that would result in an inefficient hash.*". [9]:

```
    -         long hashOfString = MurmurHash.hash64(input);
47  +         long hashOfString = -8896273065425798843L;
48            assertEquals("MurmurHash.hash64(byte[]) did not match MurmurHash.hash64(String)",
49                         hashOfString, MurmurHash.hash64(inputBytes));
```

Two days later, one developer mentioned the fact that the test is verifying the overload of the method and is not specifying the method hash itself. He closed the PR because it was not relevant to put changes there. He suggested to open an new pull request with a new test method instead of changing the existing test method. We proposed, 6 days later, a second pull request entitled "*add test for hash() and hash64() against hard coded values*" with no description, since we estimated that the developer was aware of our intention.[10]:

[9] https://github.com/addthis/stream-lib/pull/127/files
[10] https://github.com/addthis/stream-lib/pull/128/files

```
54      }
55  +
56  +   // test the returned valued of hash functions against the reference implementation: https://github
57  +
58  +   @Test
59  +   public void testHash64() throws Exception {
60  +       final long actualHash = MurmurHash.hash64("hashthis");
61  +       final long expectedHash = -8896273065425798843L;
62  +
63  +       assertEquals("MurmurHash.hash64(String) returns wrong hash value", expectedHash, actualHash);
64  +   }
65  +
66  +   @Test
67  +   public void testHash() throws Exception {
68  +       final long actualHash = MurmurHash.hash("hashthis");
69  +       final long expectedHash = -1974946086L;
70  +
71  +       assertEquals("MurmurHash.hash(String) returns wrong hash value", expectedHash, actualHash);
72  +   }
73  }⊘↵
```

The pull request has been merged by the same developer 20 days later.

4.1.7 mustache.java

We have applied DSpot to amplify `AbstractClassTest`. It identifies a try/-catch/fail block that kills 2 more mutants. This is an interesting new case, compared to the ones previously discussed, because it is about the specification of exceptions, *i.e.* of behavior under erroneous inputs. All newly killed mutants are located in method `compile()` on line 194. The test specifies that if a variable is improperly closed, the program must throw a `MustacheException`. In the Mustache template language, an improperly closed variable occurs when an opening brace "{" does not have its matching closing brace such as in the input of the proposed changes. We propose the pull request to the developers, entitled "*Add Test: improperly closed variable*" with the following description: "*Hello, I proposed this change to improve the test on MustacheParser. When a variable is improperly closed, a MustacheException is thrown.*".[11]

[11] `https://github.com/spullara/mustache.java/pull/186/files`

```
68    }
69  +
70  +  @Test
71  +  public void testImproperlyClosedVariable() throws IOException {
72  +      try {
73  +        new DefaultMustacheFactory().compile(new StringReader("{{{#containers}} {{/containers}}"), "example");
74  +        fail("Should have throw MustacheException");
75  +      } catch (MustacheException actual) {
76  +          assertEquals("Improperly closed variable in example:1 @[example:1]", actual.getMessage());
77  +      }
78  +  }
79  +
80    }
```

12 days later, a developer accepted the change, but noted that the test should be in another class. He closed the pull request and added the changes himself into the desired class.[12].

4.1.8 twilio-java

We have applied DSpot to amplify `RequestTest`. It identifies two new assertions that kill 4 more mutants. All mutants were created between lines 260 and 265 in the method `equals()` of `Request`. The change specifies that an object `Request` is not equal to null nor an object of different type, *i.e.* `Object` here. The pull request was entitled "*add test equals() on request*", accompanied with the short description "*Hi, I propose this change to specify the equals() method of com.twilio.http.Request, against object and null value*" [13]:

```
168
169  +      @Test
170  +      public void testEquals() {
171  +          Request request = new Request(HttpMethod.DELETE, "/uri");
172  +          request.setAuth("username", "password");
173  +          assertFalse(request.equals(new Object()));
174  +          assertFalse(request.equals(null));
175  +      }
176  +
177     }
```

A developer merged the change 4 days later.

[12] the diff is same:https://github.com/spullara/mustache.java/commit/9efa19d595f893527ff218683e70db2ae4d8fb2d

[13] https://github.com/twilio/twilio-java/pull/334/files

4.1.9 jsoup

We have applied DSpot to amplify `AttributeTest`. It identifies one assertion that kills 13 more mutants. All mutants are in the method `hashcode` of Attribute. The pull request was entitled "*add test case for hashcode in attribute*" with the following short description "*Hello, I propose this change to specify the hashCode of the object org.jsoup.nodes.Attribute.*"[14]:

```
19        }
20  +
21  +     @Test
22  +     public void testHashCode() {
23  +         String s = new String(Character.toChars(135361));
24  +         Attribute attr = new Attribute(s, (("A" + s) + "B"));
25  +         assertEquals(111849895, attr.hashCode());
26  +     }
27    ↴
```

One developer highlighted the point that the `hashCode` method is an implementation detail, and it is not a relevant element of the API. Consequently, he did not accept our test improvement.

At this point, we have made two pull requests targeting `hashCode` methods. One accepted and one rejected. `hashCode` methods could require a different testing approach to validate the number of potential collisions in a collection of objects rather than checking or comparing the values of a few objects created for one explicit test case. The different responses we obtained reflect the fact that developer teams and policies ultimately decide how to test the hash code protocol and the outcome could be different from different projects.

4.1.10 protostuff

We have applied DSpot to amplify `TailDelimiterTest`. It identifies a single assertion that kills 3 more mutants. All new mutants killed are in the method `writeTo` of `ProtostuffIOUtil` on lines 285 and 286, which is responsible to write a buffer into a given scheme. We proposed a pull request entitled "*assert the returned value of writeList*", with the following short description "*Hi, I propose the following changes to specify the line 285-286 of io.protostuff.ProtostuffIOUtil.*"[15], shown earlier in Figure 1

[14] `https://github.com/jhy/jsoup/pull/840`
[15] `https://github.com/protostuff/protostuff/pull/212/files`

```
146         ByteArrayOutputStream out = new ByteArrayOutputStream();
    -       writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
147 +       final int bytesWritten = writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
148 +       assertEquals(0, bytesWritten);
149         byte[] data = out.toByteArray();
150
151         ByteArrayInputStream in = new ByteArrayInputStream(data);
```

A developer accepted the proposed changes the same day.

4.1.11 logback

We have applied DSpot to amplify `FileNamePattern`. It identifies a single assertion that kills 5 more mutant. Newly killed mutants were located at lines 94, 96 and 97 of the `equals` method of the `FileNamePattern` class. The proposed pull request was entitle "*test: add test on equals of FileNamePattern against null value*" with the following short description: "*Hello, I propose this change to specify the equals() method ofFileNamePattern against null value*".[16]:

```
192         }
193 +
194 +       @Test
195 +       public void testNotEqualsNull() {
196 +           FileNamePattern pp = new FileNamePattern("t", context);
197 +           assertFalse(pp.equals(null));
198 +       }
199 +
200     }
```

Even if the test asserts the contract that the `FileNamePattern` is not equals to null, and kills 5 more mutants, the lead developer does not get the point to test this behavior. The pull request has not been accepted.

4.1.12 retrofit

We did not manage to create a pull request based on the amplification of the test suite of retrofit. According to the result, the newly killed mutants are spread over all the code, and thus the amplified methods did not identify a missing contract specification. This could be explained by two facts: 1) the original test suite of retrofit is strong: there is no test class with low mutation

[16] `https://github.com/qos-ch/logback/pull/365/files`

Table 4 Contributions of *A-Amplification* and *I-Amplification* on the amplified test method used to create a pull request.

Project	*#A-Amplification*	*#I-Amplification*
javapoet	2	2
mybatis-3	3	3
traccar	10	7
stream-lib	2	2
mustache	4	3
twilio	3	4
jsoup	34	0
protostuff	1	1
logback	2	2

score and a lot of them are very high mutation score, *i.e.* 90% and more; 2) the original test suite of retrofit uses complex test mechanism such as mock and fluent assertions of the form the `assertThat().isSomething()`. For the former point, it means that DSpot has been able to improve, even a bit, the mutation score of a very strong test suite, but not in targeted way that makes sense in a pull request. For the latter point, this puts in evidence the technical challenge of amplifying fluent assertions and mocking mechanisms.

4.1.13 Contributions of A-Amplification *and* I-Amplification *to the Pull-requests*

In Table 4, we summarize the contribution of *A-Amplification* and *I-Amplification*, where a contribution means an source code modification added during the main amplification loop. In 8 cases over the 9 pull-requests, both *A-Amplification* and *I-Amplification* were necessary. Only the pull request on jsoup was found using only *A-Amplification*. This means that for all the other pull-requests, the new inputs were required to be able: 1) to kill new mutants and 2) to obtain amplified test methods that have values for the developers.

Note that this does not contradict with the fact that the pull-requests are one-liners. Most one-liner pull-requests contain both a new assertion and a new input. Consider the following Javapoet's one liner `assertFalse(x.equals(null))` (javapoet). In this example, although there is a single line starting with "assert", there is indeed a new input, the value "null".

RQ1: Would developers be ready to permanently accept improved test cases into the test repository?
Answer: We have proposed 19 test improvements to developers of notable open-source projects. 13/19 have been considered valuable and have been

merged into the main test suite. The developers' feedback has confirmed the relevance, and also the challenges of automated test improvement.

In the area of automatic test improvement, this experiment is the first to put real developers in the loop, by asking them about the quality of automatically improved test cases. To our knowledge, this is the first public report of automatically improved tests accepted by unbiased developers and merged in the master branch of open-source repositories.

4.2 Answer to RQ2

RQ2 To what extent are improved test methods considered as focused?

Table 5 presents the results for RQ2, RQ3 and RQ4.It is structured as follows. The first column is a numeric identifier that eases reference from the text. The second column is the name of test class to be amplified. The third column is the number of test methods in the original test class. The fourth column is the mutation score of the original test class. The fifth is the number of test methods generated by DSpot. The sixth is the number of amplified test methods that met the criteria explained in subsubsection 2.4.5. The seventh, eight and ninth are respectively the number of killed mutants of the original test class, the number of killed mutants of its amplified version and the absolute increase obtained with amplification, which is represented with a pictogram indicating the presence of improvement. The tenth and eleventh columns concern the number of killed mutants when only A-amplification is used. The twelfth is the time consumed by DSpot to amplify the considered test class. The upper part of the table is dedicated to test classes that have a high mutation score and the lower for the test classes that have low mutation score.

For RQ2, the considered results are in the sixth column of Table 5. Our selection technique produces candidates that are focused in 25/26 test classes for which there are improved tests. For instance, considering test class TypeNameTest (#8), there are 19 improved test methods, and among them, 8 are focused per our definition and are worth considering to be integrated in the codebase. On the contrary, for test class ConcurrencyTest (#29), the technique cannot find any improved test method that matches the focus criteria presented in subsubsection 2.4.5. In this case, that improved test methods kill additional mutants in 27 different locations. Consequently, the intent of the new amplified tests can hardly be considered as clear.

Interestingly, for 4 test classes, even if there are more than one improved test methods, the selection technique only returns one focus candidate (#23, #24, #25, #40). In those cases, there are two possible different reasons: 1) there are several focused improved tests, yet they all specify the same application method (this is the case for #40 2) there is only one improved test method that is focused (this is the case for #23, #24, and #25)

Table 5 The effectiveness of test amplification with DSpot on 40 test classes: 24 well-tested (upper part) and 16 average-tested (lower part) real test classes from notable open-source Java projects.

ID	Class	# Orig. test methods	Mutation Score	# New test methods Candidates for pull request	# Killed mutants orig.	# Killed mutants ampl.	Increase killed	# Killed mutants only A-ampl	Increase killed only A-ampl	Time (minutes)
	High mutation score									
1	TypeNameTest	12	50%	19	8	599	715	19% ↗	599	0.0% →	11.11
2	NameAllocatorTest	11	87%	0	0	79	79	0.0% →	79	0.0% →	4.76
3	MetaClassTest	7	58%	108	10	455	534	17% ↗	455	0.0% →	235.71
4	ParameterExpressionTest	14	91%	2	2	162	164	1% ↗	162	0.0% →	25.93
5	ObdDecoderTest	1	80%	9	2	51	54	5% ↗	51	0.0% →	2.20
6	MiscFormatterTest	1	72%	5	5	42	47	11% ↗	42	0.0% →	1.21
7	TestLookup3Hash	2	95%	0	0	464	464	0.0% →	464	0.0% →	6.76
8	TestDoublyLinkedList	7	92%	1	1	104	105	0.97% ↗	104	0.0% →	3.03
9	ArraysIndexesTest	1	53%	15	4	576	647	12% ↗	586	1% ↗	10.58
10	ClasspathResolverTest	10	67%	0	0	50	50	0.0% →	50	0.0% →	4.18
11	RequestTest	17	81%	4	3	141	156	10% ↗	141	0.0% →	60.55
12	PrefixedCollapsibleMapTest	4	96%	0	0	54	54	0.0% →	54	0.0% →	13.28
13	TokenQueueTest	6	69%	18	6	152	165	8% ↗	152	0.0% →	15.61
14	CharacterReaderTest	19	79%	71	9	309	336	8% ↗	309	0.0% →	57.06
15	TailDelimiterTest	10	71%	1	1	381	384	0.79% ↗	381	0.0% →	12.90
16	LinkBufferTest	3	48%	12	7	66	90	36% ↗	66	0.0% →	3.24
17	FileNamePatternTest	12	58%	27	9	573	686	19% ↗	573	0.0% →	25.08
18	SyslogAppenderBaseTest	1	95%	1	1	143	148	3% ↗	143	0.0% →	7.88
19	RequestBuilderAndroidTest	2	99%	0	0	513	513	0.0% →	513	0.0% →	0.04
20	CallAdapterTest	4	94%	0	0	55	55	0.0% →	55	0.0% →	7.30
	Low mutation score									
21	FieldSpecTest	2	31%	12	4	223	316	41% ↗	223	0.0% →	4.44
22	ParameterSpecTest	2	32%	11	5	214	293	36% ↗	214	0.0% →	3.66
23	WrongNamespacesTest	2	8%	6	1	78	249	219% ↗	249	219% ↗	29.70
24	WrongMapperTest	1	8%	3	1	97	325	235% ↗	325	235% ↗	7.13
25	ProgressProtocolDecoderTest	1	16%	2	1	18	27	50% ↗	23	27% ↗	1.30
26	IgnitionEventHandlerTest	1	22%	0	0	13	13	0.0% →	13	0.0% →	0.77
27	TestICardinality	2	7%	0	0	19	19	0.0% →	19	0.0% →	2.13
28	TestMurmurHash	2	17%	40	2	52	275	428% ↗	174	234% ↗	2.18
29	ConcurrencyTest	2	28%	2	0	210	342	62% ↗	210	0.0% →	315.56
30	AbstractClassTest	2	34%	28	4	383	475	24% ↗	405	5% ↗	12.67
31	AllTimeTest	3	42%	0	0	163	163	0.0% →	163	0.0% →	0.02
32	DailyTest	3	42%	0	0	163	163	0.0% →	163	0.0% →	0.02
33	AttributeTest	2	36%	33	11	178	225	26% ↗	180	1% ↗	10.76
34	AttributesTest	5	52%	9	6	316	322	1% ↗	316	0.0% →	6.21
35	CodedDataInputTest	1	1%	0	0	5	5	0.0% →	5	0.0% →	3.58
36	CodedInputTest	1	27%	29	28	108	166	53% ↗	108	0.0% →	0.88
37	FileAppenderResilience_AS_ROOT_Test	1	4%	0	0	4	4	0.0% →	4	0.0% →	0.65
38	Basic	1	10%	0	0	6	6	0.0% →	6	0.0% →	0.89
39	ExecutorCallAdapterFactoryTest	7	62%	0	0	119	119	0.0% →	119	0.0% →	0.09
40	CallTest	35	69%	3	1	642	644	0.32% ↗	642	0.0% →	52.84

To conclude, according to this benchmark, DSpot proposes at least one and focused improved test in all but one cases. From the developer viewpoint, DSpot is not overwhelming it proposes a small set of suggested test changes, which are ordered, so that even with a small time budget to improve the tests, the developer is pointed to the most interesting case.

RQ2: To what extent are improved test methods considered as focused?
Answer: In 25/26 cases, the improvement is successful at producing at least one focused test method, which is important to save valuable developer time in analyzing the suggested test improvements.

4.3 Answer to RQ3

RQ3: To what extent do improved test classed kill more mutants than developer-written test classes?

In 26 out of 40 cases, DSpot is able to amplify existing test cases and improves the mutation score (MS) of the original test class. For example, let us consider the first row, corresponding to `TypeNameTest`. This test class originally includes 12 test methods that kill 599 mutants. The improved, amplified version of this test class kills 715 mutants, *i.e.* 116 new mutants are killed. This corresponds to an increase of 19% in the number of killed mutants.

We first discuss the amplification of the test classes that can be considered as being already good tests since they originally have a high mutation score: those good test classes are the 24 tests in Table 5. There is a positive increase of killed mutants for 17 cases. This means that even when human developers write good test cases, DSpot is able to improve the quality of these test cases by increasing the number of mutants killed. In addition, in 15 cases, when the amplified tests kill more mutants, this goes along with an increase of the number of expressions covered with respect to the original test class.

For those 24 well-test classes, the increase in killed mutants varies from 0,3%, up to 53%. A remarkable aspect of these results is that DSpot is able to improve test classes that are initially extremely strong, with an original mutation score of 92% (ID:8) or even 99% (ID:20 and ID:21). The improvements in these cases clearly come from the double capacity of DSpot at exploring more behaviors than the original test classes and at synthesizing new assertions.

Still looking to the upper part of Table 5 (the well-tested classes), we now focus on the relative increase in killed mutants (column "Increase killed"). The two extreme cases are `CallTest` (ID:24) with a small increase of 0.3% and `CodeInputTest` (ID:18) with an increase of 53%. `CallTest` (ID:24) initially includes 35 test methods that kill 69% of 920 covered mutants. Here, DSpot runs for 53 minutes and succeeds in generating only 3 new test cases that kill 2 more mutants than the original test class, and the increase in mutation score is only minimal. The reason is that input amplification does not trigger any new behavior and assertion amplification fails to observe new parts of the program state. Meanwhile, DSpot succeeds in increasing the number of mutants killed by `CodeInputTest` (ID:18) by 53%. Considering that the original test class

is very strong, with an initial mutation score of 60%, this is a very good achievement for test amplification. In this case, the *I-Amplification* applied easily finds new behaviors based on the original test code. It is also important to notice that the amplification and the improvement of the test class goes very fast here (only 52 seconds).

One can notice 4 cases (IDs:3, 13, 15, 24) where the number of new test cases is greater than the number of newly killed mutants. This happens because DSpot amplifies test cases with different operators in parallel. While we keep only test cases that kill new mutants, it happens that the same mutant is newly killed by two different amplified tests generated in parallel threads. In this case, DSpot keeps both test cases.

There are 7 cases with high mutation score for which DSpot does not improve the number of killed mutants. In 5 of these cases, the original mutation score is greater than 87% (IDs: 2, 7, 12, 21, 22), and DSpot does not manage to synthesize improved inputs to cover new mutants and eventually kill them. In some cases DSpot cannot improve the test class because they rely on an external resource (a jar file), or use utility methods that are not considered as test methods by DSpot and hence are not modified by our tool.

Now we consider the tests in the lower part of Table 5. Those tests are weaker because they have a lower mutation score. When amplifying weak test classes, DSpot improves the number of killed mutants in 9 out of 16 cases. On a per test class basis, this does not differ much from the well tested classes. However, there is a major difference when one considers the increase itself: the increases in number of killed mutants range from 24% to 428%. Also, we observe a very strong distinction between test classes that are greatly improved and test classes that are not improved at all (9 test classes are much improved, 7 test classes cannot be improved at all, the increase is 0%). In the former case, we find test classes that provide a good seed for amplification. In the latter case, we have test classes that are designed in a way that prevents amplification because they use external processes, or depend on administration permission, shell commands and external data sources; or extensively use mocks or factories; or simply very small test cases that do not provide a good potential to DSpot to perform effective amplification.

RQ3: To what extent do improved test classes kill more mutants than manual test classes?
Answer: In our novel quantitative experiment on automatic test improvement, DSpot significantly improves the capacity of test classes at killing mutants in 26 out 40 of test classes, even in cases where the original test class is already very strong. Automatic test improvement works particularly well for weakly tested classes (lower part of Table 5): the mutation score of three classes is increased by more than 200%.

The most notable point of this experiment is that we have considered tests

that are already really strong (Table 5), with mutation score in average of 78%, with the surprising case of a test class with 99% mutation score that DSpot is able to improve.

4.4 Answer to RQ4

What is the contribution of *I-Amplification* and *A-Amplification* to the effectiveness of automated test improvement?

The relevant results are reported in the tenth and eleventh column of Table 5. They give the number of killed mutants and the relative increase of the number of killed mutants when only using *A-Amplification*.

For instance, for `TypeNameTest` (first row, id #1), using only *A-Amplification* kills 599 mutants, which is exactly the same number of the original test class. In this case, both the absolute and relative increase are obviously zero. On the contrary, for `WrongNamespacesTest` (id #27), using only *A-Amplification* is very effective, it enables DSpot to kill 249 mutants, which, compared to the 78 originally killed mutants, represents an improvement of 219%.

Now, if we aggregate over all test classes, our results indicate that *A-Amplification* only is able to increase the number of mutants killed in 7 / 40 test classes. Increments range from 0.31% to 13%. Recall that when DSpot runs both *I-Amplification* and *A-Amplification*, it increases the number of mutants killed in 26 / 40 test classes, which shows that it is indeed the combination of *A-Amplification* and *I-Amplification* which is effective.

We note that *A-Amplification* performs as well as *I-Amplification* + *A-Amplification* in only 2/40 cases (ID:27 and ID:28). In this case, all the improvement comes from the addition of new assertions, and this improvement is dramatic (relative increase of 219% and 235%).

The limited impact of *A-Amplification* alone has several causes. First, many assertions in the original test cases are already good and precisely specify the expected behavior for the test case. Second, it might be due to the limited observability of the program under test (*i.e.*, there is a limited number of points where assertions over the program state can be expressed). Third, it happens when one test case covers global properties across many methods: test #28 `WrongMapperTest` specifies global properties, but is not well suited to observe fine grained behavior with additional assertions. This latter case is common among the weak test classes of the lower part of Table 5.

RQ4: What is the contribution of I-Amplification *and* A-Amplification *to the effectiveness of test amplification?*
Answer: The conjunct run of *I-Amplification* and *A-Amplification* is the best strategy for DSpot to improve manually-written test classes. This experiment has shown that *A-Amplification* is ineffective, in particular on tests that are already strong.

To the best of our knowledge, this experiment is the first to evaluate the relative contribution of *I-Amplification* and *A-Amplification* to the effectiveness of automatic test improvement.

5 Threats to Validity

RQ1 The major threat to RQ1 is that there is a potential bias in the acceptance of the proposed pull requests. For instance, if we propose pull requests to colleagues, they are more likely to merge them. However, this is not the case here. In this evaluation, the pull requests are submitted by the first author, who is unknown to all considered projects. The developers who study the DSpot pull requests are independent from our group and social network. Since the first author is unknown for the pull request reviewer, this is not a specific bias towards acceptance or rejection of the pull request.

RQ2 The technique used to select focused candidates is based on the proportion of mutant killed and the absolute number of modification done by the amplification. However, it may happen that some improvements that are not focused per our definition would still be considered as valuable by developers. Having such false negative is a potential threat to validity.

RQ3 A threat to RQ3 relates to external validity: if the considered projects and tests are written by amateurs, our findings would not hold for serious software projects. However, we only consider real-world applications, maintained by professional and esteemed open-source developers. This means we tried to automatically improve tests that are arguably among the best of the open-source world, aiming at as strong construct validity as possible.

RQ4. The main threat to RQ4 relates to internal validity: since our results are of computational nature, a bug in our implementation or experimental scripts may threaten our findings. We have put all our code publicly-available for other researchers to reproduce our experiment and spot the bugs, if any.

Oracle. DSpot generates new assertions based on the current behavior of the program. If the program contains a bug, the resulting amplified test methods would enforce this bug. This is an inherent threat, inherited from [27], which is unavoidable when no additional oracle is available, but only the current version of the program. To that extent, the best usage of DSpot is to improve the test suite of a supposedly almost correct version of the program.

6 Related Work

This work on test amplification contributes to the field of genetic improvement (GI) [21]. The key novelty is to consider a test suite as the object to be improved, while previous GI works improve the application code. (Yet, they use the test suite as a fitness function while assessing the degree of improvement.)

The work of Arcuri and Yao [3] and Wilkerson *et al.* [26] are good examples of such work that use the test suite as fitness, while improving the program for automatic bug fixing. Both work follow a similar approach: evolve the input program into new versions that pass the regression test suite and that also pass the bug revealing test case (that fails on the original program). In this paper, we do not evolve the application code but the test code.

Evosuite is a state of the art tool to generate test cases for Java program [12]. Evosuite and DSpot have different goals. Evosuite generates new tests, while DSpot improves existing developer-written tests. The interaction between developers and synthesized tests is key here: in 2016, an empirical study demonstrated that developers who are asked to add oracles in test cases generated by Evosuite, produce test suites that are not better than manually written test suites at detecting bugs [14]. On the contrary, DSpot is designed to improve manually written test suites to detect more bugs, and the relevance study of RQ1 demonstrates that the outcome of DSpot is considered as valuable by developers in order to improve existing test suites.

Our work is related to previous work that aim at automatically generating test cases to improve the mutation score of a test suite. Liu *et al.* [17] aim at generating small test cases, by targeting a path that covers multiple mutants to create test inputs. They evaluate their approach on five small projects. Fraser and Arcuri [13] propose a search-based approach to generate test suites that maximize the mutation score. However their work is different from ours since they generate new test cases from scratch, while DSpot always starts from developper-written tests. Baudry *et al.* [5] improve the mutation score of test suites using a bacteriological algorithm. They run experiments on a small dataset and confirm that their approach is able to increase the mutation score of tests. However, the scope of the study is limited to small programs, and they do not consider the synthesis of assertions.

Other works aim at increasing fault detection capacities of test suites. Zhang *et al.* [33], propose the Isomorphic Regression Testing system and its implementation in ISON. It considers two versions of a program P and P'(for instance P' is the updated version of P, on which we want to detect any regression). First, ISON identifies isomorphims, that is to say, code fragments that have the same behavior. Then, they run the test suite on P and P' to identify which of the branches are uncovered in the isomorphic part, and they collect the output. In order to cover all branches, they compute a branch condition to execute the uncovered code. They compare ISON to Evosuite, and conclude that Evosuite achieves a better branch coverage, while ISON is able to detect faults that Evosuite does not.

Harder *et al.* [18] start from an existing test suite. They evaluate the quality of this initial test suite with respect to operational abstractions, i.e., an abstract description of the behavior covered by the test suite. Their work is about selecting new valuable tests, while ours is about synthesizing new valuable tests.

Then, they generate novel test cases and keep only the ones that change the operational abstraction. The new test cases are generated by mining invariants

using Daikon. They evaluate their approach on 8 C programs, and show that it generates test cases with good fault-detection capabilities.

Milani *et al.* [19] propose an approach which combines the advantages of manually written tests and automatic test generation. They exploit the knowledge of existing tests and then combine it with the power of automated crawling. It has been shown that the approach can effectively improve the fault detection rate of the original test suite. Test amplification, as considered in this work, is different, as it aims at enhancing the fault detection power of manually written test suites.

Yoo *et al.* [31] propose Test Data Regeneration(TDR), which is a kind of test amplification. They use hill climbing on existing test data (set of input) that meets a test objective (*e.g.* cover all branch of a function). The algorithm is based on *neighborhood* and a *fitness* functions as the classical hill climbing algorithm. The goal is to create new test data inputs, that have the same behavior as the original one (*e.g.* cover same branches). The key novelties of our work with respect to the work of Yoo *et al.* [31] are as follow: they mutate only literals in existing test cases, while DSpot's *I-Amplification* also amplifies method calls and can synthesize new objects when needed, *A-Amplification* makes the synthesis of assertions an integral part of our test suite improvement process and we evaluate the relevance of the synthesized test cases by proposing them to the developers.

Xie [27] proposes a technique to add assertions into existing test methods. His approach is similar to what we propose with *A-Amplification*. However, this work does not consider the synthesis of new test inputs (*I-Amplification*) and hence cannot cover new execution paths. This is the novelty of DSpot and our experiments showed that this is an essential mechanism to improve the test suite.

We now discuss a group of papers together. Pezzè *et al.* [22] synthesize integration test cases from unit test cases. The idea is to combine unit test cases, which test simple functionalities on specific objects, to create new integration test cases supported by the fact that unit test cases are early developed, and integration test cases require more effort to do so. Röβler et al. [23] aim to isolate failure causes. They propose BugEx, a system that starts from a single failing test as input and generates test cases. It extracts the differences in path execution between failing and passing tests. They evaluate BugEx on 7 failures and show that it is able to lead to the failure root causes in 6 cases. Yu *et al.* [32] augment test suites to enhance fault localization. They use test input transformations to generate new test cases in existing test suites. They transform iteratively some existing failing tests to derive new test cases potentially useful to localize the specific encountered fault, similarly at *I-Amplification*. Their tool is designed to target GUI applications. To reproduce a crash occurred in production, Xuan *et al.* [30] propose to transform existing test cases. The approach first selects relevant test cases based on the stack trace in the crash, followed by the elimination of assertions in selected test cases, and finally uses a set of predefined transformations to produce new test cases that can help to reproduce the crash. None of those works have evaluated whether

the technique scales on object-oriented applications of the size considered here, and whether the synthesized tests are considered valuable by senior developers.

It can be noted that several test generation techniques start from a seed and evolve it to produce a good test suite. This is the case for techniques such as concolic test generation [15], search-based test generation [11], or random test generation [16]. The main difference between all these works and DSpot lies in the nature of the seed: previous work use input values in the form of numerical or String values, vectors or files, and do not consider any form of oracle. On the contrary, we consider as a seed a real test case. It means the seed is a complete program, which creates objects, manipulates the state of these objects, calls methods on these objects and asserts properties on their behavior. This is the contribution of DSpot: using real and complex object-oriented tests as seed.

Almasi *et al.* [2] investigate the efficiency and effectiveness of automated test generation on a production ready application named *LifeCalc*. They use 25 real faults from *LifeCalc* to evaluate two state-of-the-art tools, Evosuite and Randoop, by asking feedback from the developers about the generated test methods. The result are as follows: overall the tools found 19 over 25 real faults; The developers state that the assertions and the readability of generated test methods must be improved. The developers also suggest that such tools should be implemented in continuous integration. The reason of the 7 faults that remain undetected is that they either require complex test data input or specific assertions. Our experiment is larger in scope, we evaluate DSpot on 10 notable open-source software from GitHub, by proposing amplified test methods in pull requests.

Allamanis *et al.* [1] devised a technique to rename elements in code and evaluate their approach through five pull requests where four of them have been accepted. Their work and ours both rely on independent evaluation through pull-requests. One important difference is that, in the description of the pull request, they say that the improvements are generated by a tool, while in our case, we did not say anything about the research project underlying our pull requests.

7 Conclusion

We have presented DSpot, a novel approach to automatically improve existing developer-written test classes. We have shown that DSpot is able to strengthen real unit test classes in Java from 10 real-world projects. Our experiment with real developers indicates that they are ready to merge test cases improved by DSpot into their test suite. The road ahead for automatic synthesis of test case improvements is exciting.

First, there is a need to study how to generate meaningful natural language explanations of the suggested test improvements: generation of well named tests, generation of text accompanying the pull request, we dream of using natural-language deep-learning for this task.

Second, we aim at automating even more the process of integrating the amplification result in a ready-to-use pull request. This requires two major steps: first, one needs to identify which parts of the amplified test methods are "valuable". Second, we need to choose between modifying an existing test method or create a new one that is derived from an existing one, even if the new method is by construction an extension of an existing one. Such a decision procedure must be made based on the intention of the existing test methods and the potentially new intention of the amplified test. If we find an existing test method that carries the same intention, *i.e.* it tests the same portion of code as the amplification, one would preferably add changes there rather than creating a new test methods. This challenging vision of mining and comparing test purposes is the main area of our future work.

Third, and finally, we envision to integrate DSpot in a continuous integration service (CI) where test classes would be amplified on-the-fly for each commit. This would greatly improve the direct industrial applicability of this software engineering research.

References

1. M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.
2. M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, May 2017.
3. A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 162–168. IEEE, 2008.
4. B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus. DSpot: Test Amplification for Automatic Assessment of Computational Diversity. ArXiv paper 1503.05807, 2015.
5. B. Baudry, F. Fleurey, J.-M. Jézéquel, and L. Yves. From genetic to bacteriological algorithms for mutation-based testing. *Software, Testing, Verification & Reliability journal (STVR)*, 15(2):73–96, June 2005.
6. K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
7. M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.
8. B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry. The emerging field of test amplification: A survey. *arXiv preprint arXiv:1705.10692*, 2017.
9. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
10. B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.
11. G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 121–130. IEEE, 2012.
12. G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.
13. G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

14. G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):23, 2015.
15. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
16. A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th international conference on Software Engineering*, pages 621–631. IEEE Computer Society, 2007.
17. M. h. Liu, Y. f. Gao, J. h. Shan, J. h. Liu, L. Zhang, and J. s. Sun. An approach to test data generation for killing multiple mutants. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 113–122, Sept 2006.
18. M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proc. of the Int. Conf. on Software Engineering (ICSE)*, pages 60–71, 2003.
19. A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78. ACM, 2014.
20. R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46:1155–1179, 2015.
21. J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward, et al. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.
22. M. Pezz, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 11–20, Washington, DC, USA, 2013. IEEE Computer Society.
23. J. Röβler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 309–319. ACM, 2012.
24. J. Roche. Adopting devops practices in quality assurance. *Commun. ACM*, 56, 2013.
25. P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.
26. J. L. Wilkerson and D. Tauritz. Coevolutionary automated software correction. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1391–1392. ACM, 2010.
27. T. Xie. Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 380–403, 2006.
28. T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In D. Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 380–403, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
29. J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.
30. J. Xuan, X. Xie, and M. Monperrus. Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 910–913, New York, NY, USA, 2015. ACM.
31. S. Yoo and M. Harman. Test data regeneration: Generating new test data from existing test data. *Softw. Test. Verif. Reliab.*, 22(3):171–201, May 2012.
32. Z. Yu, C. Bai, and K.-Y. Cai. Mutation-oriented Test Data Augmentation for GUI Software Fault Localization. *Inf. Softw. Technol.*, 55(12):2076–2098, Dec. 2013.
33. J. Zhang, Y. Lou, L. Zhang, D. Hao, L. Zhang, and H. Mei. Isomorphic regression testing: Executing uncovered branches without test augmentation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 883–894, New York, NY, USA, 2016. ACM.