Test Amplification in the Pharo Smalltalk Ecosystem

Mehrdad Abdi, Henrique Rocha, Serge Demeyer

▶ **To cite this version:**

Mehrdad Abdi, Henrique Rocha, Serge Demeyer. Test Amplification in the Pharo Smalltalk Ecosystem. International Workshop on Smalltalk Technologies, Aug 2019, Koln, Germany. hal-04053118

HAL Id: hal-04053118
https://hal.science/hal-04053118

Submitted on 31 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

Test Amplification in the Pharo Smalltalk Ecosystem

Mehrdad Abdi
Universiteit Antwerpen
Antwerpen, België

Henrique Rocha
Universiteit Antwerpen
Antwerpen, België

Serge Demeyer
Universiteit Antwerpen, België
Flanders Make, België

Abstract

Test amplification is the act of strengthening existing unit-tests to exercise the boundary conditions of the unit under test. It is an emerging research idea which has been demonstrated to work for Java, relying on the type system to safely transform the code under test.

In this paper we report on a feasibility study concerning test amplification in the context of the Smalltalk eco-system. We introduce a proof-of-concept test amplifier named Small-Amp, and discuss the advantages and challenges we encountered while incorporating the tool into the Pharo Smalltalk environment. We demonstrate that by building on top of the Refactoring Browser API, the MuTalk mutation tool, it is feasible to build a test amplifier in Pharo Smalltalk despite the absence of a type system.

CCS Concepts • **Software and its engineering → Software testing and debugging**; *Software evolution*;

Keywords test amplification, automatic testing, test optimization, dynamic languages

ACM Reference Format:
Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. 2019. Test Amplification in the Pharo Smalltalk Ecosystem. In *Proceedings of International Workshop on Smalltalk Technology (SUBMITTED TO IWST19 2019).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/xxx

1 Introduction

Test amplification is the act of automatically transforming a manually written unit test to exercise boundary conditions. Although test amplification may appear similar to test generation, there is a key difference between them: the input. Test amplification relies on test cases previously written by developers, while most test generation techniques ignore existing tests and just use the system under test to generate new tests.

Modern software projects contain a considerable amount of hand written test cases [7]. Such tests are a valuable asset for assuring that the code does not regress when the system under test evolves. Unfortunately, these test cases mainly exercise to the default usage scenarios and seldom cover corner cases. In test amplification, we extend these test cases to increase a specific measurable property, typically some form of code coverage (branch coverage, mutation coverage) [3].

SUBMITTED TO IWST19 2019, August 27-29th, 2019, Cologne, Germany
2019. ACM ISBN xxx…$15.00
https://doi.org/10.1145/xxx

DSpot [4] is a test amplification tool for Java projects combining two techniques: (i) evolutionary test case generation or *Input Amplification* [9], and (ii) regression oracle generation or *Assert Amplification* [11]. DSpot uses the manually written test cases as the initial population of the evolutionary algorithm. Then in each iteration, it creates extra test cases by changing the setup part of tests and regenerating the assertions, resulting in a new and larger set of test cases. The algorithm then selects tests which increase the mutation coverage, discarding others. This process is performed for a fixed number of steps which eventually results in a new test suite, with a better mutation coverage than the initial one, thus covering more corner cases.

In Pharo Smalltalk [8], there are many tools designed to help developers with their testing. For instance, MuTalk [10], and SMutant [6] are mutation testing frameworks. Hapao [2] is another testing tool focused on coverage. Moreover, DrTest [5] is a new plugin based testing platform meant to replace previous SUnit [8] UI. However, as far as we know, there is currently no tool for Pharo Smalltalk to support test amplification.

In this paper, we show that it is feasible to implement a test amplifier technique in the Pharo Smalltalk ecosystem. We introduce Small-Amp, a proof-of-concept test amplifier based on DSpot evolutionary approach, on top of the Smalltalk powerful Refactoring Browser library, and using MuTalk as the mutation testing framework. We use a simple example to show how Small-Amp can improve a test suite.

The rest of the paper is structured as follows. In Section 2, we elaborate on the concept of test amplification, taking DSpot as a concrete example. In Section 3, we describe Small-Amp, the proof-of-concept tool implemented in Pharo Smalltalk. In Section 4 we discuss the challenges and lessons learned regarding test amplification in Pharo Smalltalk. Finally, in Section 5 we present our final remarks and outline future work ideas.

2 Background

In this section we elaborate on the concept of test amplification (Section 2.1), taking DSpot as a concrete example (Section 2.2).

2.1 Test Amplification

Test amplification is an emerging topic in software engineering. Danglot et al. conducted a literature survey, classifying test amplification works in four main groups [3].

AMP_{add}: Test amplification technique AMP_{add} consists of creating new tests from existing ones so as to improve an engineering goal. The most commonly used engineering goal is to improve coverage according to a coverage criterion.

AMP_{change}: Test amplification technique AMP_{change} consists of adding new tests to the current test suite, by creating new tests that cover and/or observe the effects of a change in the application code.

AMP_{exec}: Test amplification technique AMP_{exec} consists of modifying the test execution process or the test harness in order to maximize the knowledge gained from the testing process.

AMP_{mod}: Test amplification technique AMP_{mod} refers to modifying the body of existing test methods. The goal here is to make the scope of each test cases more precise or to improve the ability of test cases at assessing correctness (with better oracles). Differently from AMP_{add}, it is not about adding new test methods or new tests classes.

Based on their snowballing survey, they found 48 different tools where only 18 of them was publicly available. Most works aim at improving the code coverage and the most targeted systems are Java programs.

2.2 DSpot

DSpot [1, 4] is a test amplification or missing assertion generator tool on Java for JUnit tests. It reads a Java project and its existing test suites and writes new test cases based on existing ones.

It is built upon the algorithms of Tonella[9] and Xie [11]. The transformation of existing tests to generate new test cases is based on Tonella's evolutionary technique (called Input-Amplification), and inserting new assertion statements to test body using Xie's technique (called Assertion-Amplification). DSpot uses mutation coverage as the measurable property which is going to be enhanced. More specifically, DSpot keeps new tests that are able to kill mutants that were not killed by the original test suite.

DSpot main loop is presented in Algorithm 1. DSpot takes the program under test P and its related test suite (TS) as the input. The other input is the number of iterations N which is a parameter related to the evolutionary part of the algorithm. As the result, DSpot returns a new amplified test suite (ATS). DSpot generates new versions for each test case in *amplifyAssertion* step, then it insert new assertions to them in *amplifyAssertion* step and finally selects the tests which are better in killing mutant in *applySelection* step. DSpot applies these steps on the generated test cases N times.

Input amplification is the process of generating new versions of a test case by mutating some parts of the original test. DSpot removes all existing assertion statements before generating new versions of the test cases. There are three categories of mutation operations in Input amplification:

- Mutating literals
- Adding new method calls
- Create new objects (if needed) as the parameter of newly added method calls

```
input  : program P
input  : existing test suite TS
input  : number of iterations N
output : new amplified test suite ATS
1  ATS ← {};
2  for each t ∈ TS do
3  |   U ← amplifyAssertions(t);
4  |   ATS ← applySelection(U, P);
5  |   TMP ← ATS;
6  |   for i ← 0 to N do
7  |   |   V ← amplifyInputs(TMP);
8  |   |   V ← amplifyAssertions(V);
9  |   |   ATS ← ATS ∪ applySelection(V, P);
10 |   |   TMP ← V;
11 |   end
12 end
13 return ATS
```

Algorithm 1: DSpot main loop

Assertion amplification is the process of automatically creating new assertions. This process consists of three steps:

1. Creating observation points
2. Running the program to collect the state of objects
3. Create new assertions based on collected data

To select test methods, DSpot uses a heuristic which sorts amplified test methods based on (1) the ratio of killing new mutants (2) the number of modifications in the code (smaller changes are more preferred) (3) maximum numbers of mutants killed in the same method.

DSpot is implemented in Java and it is publicly available in Github[1]. It consists of 8,800+ logical lines of code and it is implemented on top of Spoon and Pitest.

3 Small-Amp

In this section, we describe our test amplification technique. Small-Amp is a replication of DSpot in the Pharo Smalltalk ecosystem. With this replication, we aimed to investigate the feasibility of implementing the most important aspects of DSpot. The project is open source and is publicly available in Github[2].

3.1 Main Loop

The main loop of Small-Amp is similar to Algorithm 1. First, we reject all failing test-cases and consider the passing tests as the initial population. The population enters to an evolutionary loop which iterates N times. In each iteration the

[1] https://github.com/STAMP-project/dspot, verified 2019-06-12.
[2] https://github.com/mabdi/small-amp, verified 2019-06-12.

following steps will be applied on the population respectively:

1. Input-Amplification
2. Assert-Amplification
3. Selection

At the end, Small-Amp makes a new test-class using the latest population. As an illustrative example, we use a sample class **SmallBank** and its related test class **SmallBankTest** indicated respectively in Listing 1 and Listing 2.

```
Object subclass: #SmallBank
instanceVariableNames: 'balance'
classVariableNames: ''
poolDictionaries: ''
category: 'Amplification−TestsClasses'

initialize
  balance := 0

balance
  ^ balance

withdraw: amount
  (balance >= amount)
    ifTrue: [ balance := balance − amount]

deposit: amount
  balance := balance + amount
```

Listing 1. SmallBank

```
TestCase subclass: #SmallBankTest
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Amplification−TestsClasses'

testInit
  | b |
  b := SmallBank new.
  self assert: b balance equals: 0

testDeposit
  | b |
  b := SmallBank new.
  b deposit: 10.
  self assert: b balance equals: 10.
  b deposit: 100.
  self assert: b balance equals: 110

testWithdraw
  | b |
  b := SmallBank new.
  b deposit: 100.
  self assert: b balance equals: 100.
  b withdraw: 30.
  self assert: b balance equals: 70
```

Listing 2. SmallBankTest

```
testWithdraw
  | b |
  b := SmallBank new.
  b deposit: 100.
  b balance.
  b withdraw: 30.
  b balance
```

Listing 3. Removing Assertions

```
testWithdraw_0
  | b |
  b := SmallBank new.
  b deposit: 50.  "100 −> 50"
  b balance.
  b withdraw: 30.
  b balance
```

Listing 4. A sample method of mutated versions

3.2 Input Amplification

Input amplification consists of the following steps:

1. Remove assertions.
2. Mutate the stripped test method.

Small-Amp finds all message calls appearing as a parameter in assertion statements and copies them before the assertion. After extracting the message calls it removes the assertions from the test method. Small-Amp keeps the existing message calls because some message calls may change the internal state of the receiver.

As an example, let's take the **testWithdraw** method in Listing 2. This function after applying the step 1 will be similar to Listing 3.

In the second step, Small-Amp starts to mutate the modified test method. Currently just literal mutations are implemented, according to the ones listed in Table 1. Applying mutators on a method generates a lot of versions of that method. Listing 4 shows one example, where the constant value of 100 is replaced by 50.

Integers	±1 multiply and divide by 2 Int.MIN, 0, Int.MAX
Booleans	not
Strings	empty string, random string, add new character to random position remove a character randomly change a character randomly
Character	±1, null, cr, space random character

Table 1. Literal mutation operators

```
testWithdraw_0
  | b tmp_Jzu0JrwJ7V1 tmp_TyOf5CP59Q2 tmp_fPcdrWheM03
      tmp_xjg7sDGICl4 |
  b := SmallBank new.
  tmp_Jzu0JrwJ7V1 := b deposit: 50.
  tmp_TyOf5CP59Q2 := b balance.
  tmp_fPcdrWheM03 := b withdraw: 30.
  tmp_xjg7sDGICl4 := b balance
```

Listing 5. testWithdraw after adding temporary variables

```
testWithdraw_0
  | b tmp_Jzu0JrwJ7V1 tmp_TyOf5CP59Q2 tmp_fPcdrWheM03
      tmp_xjg7sDGICl4 |
  b := SmallBank new.
  self observe: anObject name: #b id: 0.
  tmp_Jzu0JrwJ7V1 := b deposit: 50.
  self observe: tmp_Jzu0JrwJ7V1 name: #tmp_Jzu0JrwJ7V1 id: 1.
  self observe: b name: #b id: 2.
  tmp_TyOf5CP59Q2 := b balance.
  self observe: tmp_TyOf5CP59Q2 name: #tmp_TyOf5CP59Q2 id: 3.
  self observe: b name: #b id: 4.
  tmp_fPcdrWheM03 := b withdraw: 30.
  self observe: tmp_fPcdrWheM03 name: #tmp_fPcdrWheM03 id: 5.
  self observe: b name: #b id: 6.
  tmp_xjg7sDGICl4 := b balance
  self observe: tmp_xjg7sDGICl4 name: #tmp_xjg7sDGICl4 id: 7.
  self observe: b name: #b id: 8.
```

Listing 6. testWithdraw after adding observation statements

3.3 Assertion Amplification

Assertion amplification consists of the following steps:

1. Add temporary variables.
2. Add observers.
3. Dynamic observation.
4. Insert assertions.

At the beginning of the assertion amplification, Small-Amp adds some temporary variables to the source code. These variables are used to capture the result of all message calls (and nested calls) that are not enclosed by an assignment. Small-Amp uses random names for these temporary variables. Listing 5 is an example of **testWithdraw_0** after adding temporary variables.

Then, observation statements are added after each message call. Small-Amp adds two observation statements after each assignment, one for observing the state of the receiver and the other for the result of the message call. We do not observe state of the classes so after a class side message call we have only one observation statement observing the result of the call. The observation statement is in the form of *self observe: anObject name: anString id: anInt.* The **observe** method is a port to one of the Small-Amp core classes which serialize an object and keep it in an incremental list. We call this list the memory. Listing 6 is an example of **testWithdraw_0** after adding observation statements.

After inserting the observation points, Small-Amp builds a test class based on the original test class without its test methods. It adds the generated test method and the `observe:name:id:` method to this class. Then Small-Amp runs the tests by calling *suite run* on the test class. During test execution, each observation statement serializes the object under observation and maintains it in the memory. The serialization process is a nested operation. We have chosen 5 as the max depth of the process, thus Small-Amp will ignore the objects in depth more than 5.

Small-Amp serialization is performed dynamically as follows:

1. If the type of object at the runtime is kind of Number or String or other primitive types, Small-Amp will capture the string value of the object.
2. Else if the type of object at the runtime is kind of Collections like Set or OrderedCollection, Small-Amp iterates over the items and serialize each of them and stores them as an OrderedCollection.
3. Otherwise, If the type of object is not primitive nor collection, it's considered as an object. Small-Amp will grab all methods in accessing protocol of the object and will serialize the result and store it in a dictionary object.

When the test suite is executed, Small-Amp collects the memory variables related to this run and analyses each record and based on the changes on the object state, it generates assertion statements and insert them in the place of the related observation point. During the assert generation process the object state is compared to its last state and if they are not equal:

- If it is the serialization of a primitive: we assert if the object's value *asString* is equal to the value in memory.
- If it is the serialization of another object: we assert the value of each accessor which is not equal to the previous one.
- If the object is the serialization of a collection: if the size of the collection is changed, we assert the new value of the collection size. And if all elements except for one are equal, we assert the changed value. Also, if the size just increased by 1 and all the items in the new collection are the same except for one, we assert the value of the newly added item.

Listing 7 is an example of **testWithdraw_0** after adding assertion calls.

3.4 Selection

We implemented the same heuristic suggested by DSpot in Small-Amp. As we explained in Section 2.2, DSpot selection is based on killed mutants. Therefore, we need to employ mutation testing for the selection phase in Small-Amp. In this version of Small-Amp, we used MuTalk [10] as the framework for all computations regarding mutation testing.

```
testWithdraw_0
  | b tmp_Jzu0JrwJ7V1 tmp_TyOf5CP59Q2 tmp_fPcdrWheM03
      tmp_xjg7sDGICl4 |
  b := SmallBank new.
  self assert: b balance equals: 0.
  tmp_Jzu0JrwJ7V1 := b deposit: 50.
  self assert: tmp_Jzu0JrwJ7V1 balance equals: 50.
  self assert: b balance equals: 50.
  tmp_TyOf5CP59Q2 := b balance.
  self assert: tmp_TyOf5CP59Q2 asString equals: '50'.
  tmp_fPcdrWheM03 := b withdraw: 30.
  self assert: tmp_fPcdrWheM03 balance equals: 20.
  self assert: b balance equals: 20.
  tmp_xjg7sDGICl4 := b balance.
  self assert: tmp_xjg7sDGICl4 asString equals: '20'
```

Listing 7. testWithdraw after adding assertions

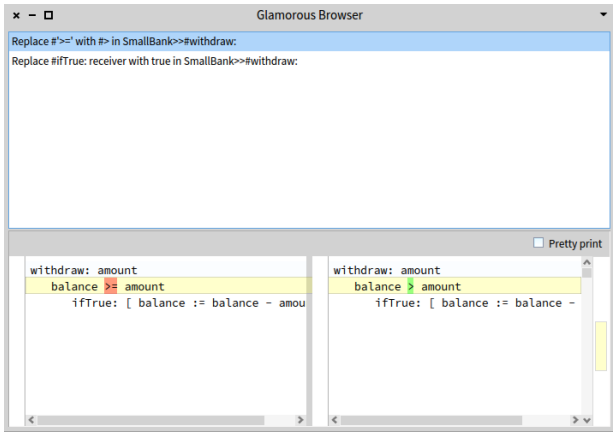

Figure 1. MuTalk results showing the surviving mutations before amplification

3.5 Illustrative Example

We use the *SmallBankTest* (Listing 2) example to assess whether Small-Amp is capable of enhancing a test suite. Since our goal is to show a proof-of-concept, we believe this example is enough to demonstrate the feasibility of test amplification in Pharo. A case study using a real application is outside the scope of this paper.

First, we execute MuTalk on the test suite and take note of how many mutants survive before the amplification. Figure 1 shows the result of MuTalk, where there are still two mutants surviving after the mutation test. Therefore, if we want to improve the mutation score of this test suite, we need to kill those surviving mutants.

Second, we used Small-Amp on the same test suite. We selected *three* as the number of iterations (i.e., *N* detailed in Algorithm 1). The amplification performed in about two minutes.

Third, we executed MuTalk again on the amplified test suite. We expect to see an improvement in the mutation score (i.e., fewer mutants surviving). After the amplification, **zero** mutants survived. Consequently, this result shows that Small-Amp can indeed enhance a test suite. In this case, the mutation score improved after the test amplification.

4 Lessons Learned

In this section we talk about the lessons learned from implementation and pre evaluating the the proof-of-concept Small-Amp. These lessons include the challenges of test amplification in smalltalk, observations from preliminary running and future works.

Smalltalk is a dynamic language and type information is not accessible until runtime. Type information is required in two situations for test amplification using Small-Amp:

- Serializing objects in assert amplification. Fortunately this action is a dynamic step and type information is accessible.
- Mutation analysis. Mutation operators need type information to generate different versions of the application. Existing mutation analysis, SMutant[6] and Mutalk [10], have overcamed this problem.

One main difference between Smalltalk and C like programming languages is the message based mechanisms. Most structural syntax statements (e.g., if, while) are implemented by sending messages to objects. This actually makes it easier to implement the amplification. Furthermore, cascade statements or nested message calls can compress a sequence of calls into a single statement. Currently, we did not break them to smaller statements. We can add a normalization step before Input amplification to break down nested statements to simpler ones. Another difference is the existence of blocks. Currently, in Small-Amp, we completely ignore the codes inside of the blocks. In this case, we may skip assertions inside blocks.

Another challenge is how we can serialize the objects. We used methods in accessing protocol to query the state of an object. But there's no guaranty that developers have implemented all their accessors methods in this protocol. We can use reflection mechanisms to capture the internal state of the objects, but in this case, we miss the encapsulation of objects. Moreover, another possible issue is nondeterministic accessors, like *atRandom*. DSpot uses a nondeterministic method detection mechanism. Such detection executes each method three times and if the results are not equal, it marks the accessor as nondeterministic and ignores it.

The increasing computational cost and execution time in each generation is another challenge. This cost forces us to use small numbers for iterations (Figure 2). Using a small number of iterations may hinder the benefits of the evolutionary algorithm. Possibly, reducing the computation complexity in each generation could allow us to use a higher number of iterations and take better gain of the genetic algorithm advantages. Moreover, we found out that the Assertion amplification step is computationally expensive.

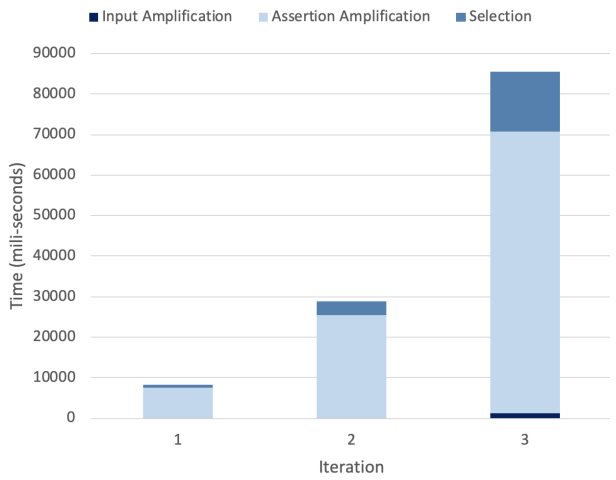

Figure 2. Run time performance per iteration

The number of temporary variables increases at each generation quickly which causes a problem in the compiler at higher generations. Furthermore, this makes the generated test methods unreadable and hard to maintain. We need a cleanup step after each generation to remove unnecessary temporary variables or to store them in a collection.

5 Conclusions

Test amplification is an emerging topic where manually written tests are used as a basis to create new test cases. Therefore, test amplification could potentially enhance a test suite by improving one of its characteristics (e.g., coverage, mutation score).

In this paper, we showed that it is feasible to introduce test amplification techniques into the Pharo ecosystem. We presented Small-Amp as a proof-of-concept tool. We applied our tool on an illustrative example which improved the test suite mutation score. We also discussed lessons learned from this implementation, challenges, and solutions to further develop test amplification in the Pharo environment.

As future work, we plan to add a cleanup step at the end of each generation to make the created test more readable. We are also going to enhance the mutation testing framework to tag the assertion statements that killed a mutant. Therefore, we can distinguish useless assertions and safely erase them. Moreover, we intend to add Small-Amp as a plugin for the DrTest framework.

Another line of future work is to define a fitness function, assessing the understandability quality of tests and use that to discard tests with low readability before the assertion amplification phase. We also are going to conduct a case study applying an improved version of Small-Amp on a test suite from a real software.

Acknowledgments

This work is supported by (a) the Fonds de la Recherche Scientifique-FNRS and the Fonds Wetenschappelijk Onderzoek - Vlaanderen (FWO) under EOS Project 30446992 SECO-ASSIST (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

[1] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. 2015. DSpot: Test Amplification for Automatic Assessment of Computational Diversity. *CoRR* abs/1503.05807 (2015). arXiv:1503.05807 http://arxiv.org/abs/1503.05807

[2] Alexandre Bergel and Vanessa Peña. 2014. Increasing test coverage with Hapao. *Science of Computer Programming* 79 (2014), 86–100. https://doi.org/10.1016/j.scico.2012.04.006 Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[3] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2018. A Snowballing Literature Study on Test Amplification. *arXiv paper 1705.10692v2* (2018).

[4] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2019. Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects. *Empirical Software Engineering, Springer Verlag* (2019).

[5] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P Black, and Anne Etien. 2019. Rotten Green Tests. In *41th International Conference on Software Engineering (ICSE '19)*. IEEE, 500–511. https://hal.inria.fr/hal-02002346

[6] Milos Gligoric, Sandro Badame, and Ralph Johnson. 2011. SMutant: a tool for type-sensitive mutation testing in a dynamic language. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 424–427.

[7] L. Madeyski. 2009. *Test-Driven Development: An Empirical Evaluation of Agile Practice.* Springer Berlin Heidelberg.

[8] Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. 2010. *Pharo by Example.* Square Bracket Associates, c/o Oscar Nierstrasz.

[9] Paolo Tonella. 2004. Evolutionary testing of classes. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04* (2004). https://doi.org/10.1145/1007512.1007528

[10] Hernán Wilkinson, Nicolás Chillo, and Gabriel Brunstein. 2009. Mutation Testing. European Smalltalk User Group (ESUG 09). Brest, France. http://www.esug.org/data/ESUG2009/Friday/Mutation_Testing.pdf.

[11] Tao Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. *Lecture Notes in Computer Science* (2006), 380–403. https://doi.org/10.1007/11785477_23