Evaluating the efficiency of continuous testing during test-driven development

Evaluating the Efficiency of Continuous Testing during Test-Driven Development

Serge Demeyer
Inria Lille - Nord Europe, France
Universiteit Antwerpen
Flanders Make vzw (Belgium)

Benoît Verhaeghe, Anne Etien,
Nicolas Anquetil, Stéphane Ducasse
Université de Lille, CNRS, Inria,
Centrale Lille, UMR 9189 – CRIStAL, France

***Abstract*—Continuous testing is a novel feature within modern programming environments, where unit tests constantly run in the background providing early feedback about breaking changes. One of the more challenging aspects of such a continuous testing tool is choosing the heuristic which selects the tests to run based on the changes recently applied. To help tool builders select the most appropriate test selection heuristic, we assess their efficiency in a continuous testing context. We observe on two small but representative cases that a continuous testing tool generates significant reductions in number of tests that need to be executed. Nevertheless, these heuristics sometimes result in false negatives, thus in rare occasions discard pertinent tests.**

***Index Terms*—test-driven development; test selection; continuous testing**

I. Introduction

A recent and widely discussed trend within the software testing community is named *shift left*: beginning testing as early as practical in the lifecycle — earlier than ever before [1], [2]. Test-driven development (sometimes referred to as *test-first* programming) is a frequently cited example as it is adopted in many agile development approaches [3]. The ultimate tool support for test-driven development is dubbed *continuous testing*: running the tests within the integrated development environment while changes are being made [4], [5]. Today, there exist several continuous testing plug-ins for programming environments like Eclipse, IntelliJ and VisualStudio: infinitest, NCrunch, dotCover to name but a few.

One detrimental side effect of the *shift left* trend, is the amount of resources spent during test execution. Runeson, for example reported that some unit test suites take hours to run [6]. To avoid executing the complete test suite over and over again, a lot of research has been performed on *test selection* heuristics [7], [8]. These heuristics identify the subset of tests that may be affected by a given change and only run that subset. Unfortunately, most of the existing work is in the context of reducing *regression test suites*, hence are too heavyweight for use within an integrated development environment. Of the lightweight heuristics available, the dominant approaches are *static analysis* and *dynamic analyis* [7]. Both of these construct a test dependency graph containing traceability links between the base code and the test code and use that as the basis for deciding which tests need to be rerun.

In this paper, we aim to help tool builders selecting the most appropriate test selection heuristic in a continuous testing context. To assess the efficiency of such a heuristic, we focus on three research criteria.

1) *Reduction:* What is the percentage of the complete test suite that needs to be rerun? In other words, compared to a brute-force *retest-all* approach, how many unnecessary test executions can be avoided?
2) *False Negatives:* How many tests are discarded inappropriately? Depending on the approach used (*i.e.,* static vs. dynamic) the construction of the dependency graph will forsake some traceability links, potentially discarding tests covering the latest changes.
3) *Static v.s Dynamic:* In which cases do these approaches select different subsets of tests? Does this result in a better reduction without creating too many false negatives?

To perform the evaluation, we created a prototype continuous testing tool named SmartTest, which features both static and dynamic construction of the test dependency graph. The prototype is implemented in the Pharo integrated development environment for the Smalltalk programming language [9]. This environment is well suited for such an evaluation because it records fine-grained change events and provides for subsequent playback [10]. Moreoever, the SUnit framework is tightly integrated into the environment, which allows for seamless test execution [11]. Finally, the meta-object protocol allows to intercept messages at run-time, which offers a convenient way to monitor test executions [9, Chapter 13].

We conducted a pilot study applying SmartTest on two small (a few hundreds lines) projects following a test-driven development approach. We discovered that a continuous testing tool may achieve significant reductions, in number of tests that need to be executed (between 70 and 80 %). We also observed that overall the static test-selection approach performed slightly better than the dynamic one however not always. Last but not least, both the static and the dynamic sometimes result in false negatives, thus discard pertinent tests.

The remainder of this paper is organised as follows. Section II gives an overview of the state of the art in continuous testing and test selection. Section III describes the case study set-up, which naturally leads to Section IV reporting the results. Finally, we summarise our findings and lessons learned in Section V.

II. Related Work

Automated test suites grow along with the software systems under tests, often making it too costly to execute entire test suites. Runeson, for example reported that some unit test suites take hours to run [6]. Consequently, the last decade has seen an increasing interest in *test selection*, seeking to identify those test cases that are relevant for the most recent changes. Since 2008 five literature surveys have been published on the topic, illustrating the vast amount of knowledge [12], [13], [14], [7], [8].

During test selection, a software analysis tool constructs a test dependency graph, mapping the relevant portions of the code under test to the corresponding test cases. Note however that the test selection heuristics are designed to reduce large regression test suites which run infrequently on a test execution environment with plenty of resources. As a consequence, most of them are rather heavyweight, relying on among others control dependence graphs, program slicing, cluster identification, ... [7].

For continuous testing lightweight heuristics are more appropriate [4], [5]. On the other hand, since continuous testing tools are part of an integrated development environment, they have access to the internal data structures within. Currently, there are two dominant approaches in the state-of-the-practice: *static* (where the test dependency graph is constructed from the source code or some representation of it) and *dynamic* (where the test dependency graph is constructed from execution traces recorded during actual test runs).

Note that the programming language supported by the integrated development environment is a complicating factor. In particular, whether the language is statically typed (Java, C++, C#, ...) or dynamically typed (Python, Javascript, Smalltalk, ...). Indeed, during static analysis, the test dependency graph is constructed from the call graph; a data structure containing all the methods within the system and the direct calls between them. Once the call graph is available, the test dependencies are derived by inverting the calling relationship from the method changed to the tests that cover it. The presence of a type system[1] obviously helps to construct a reasonably accurate dependency graph. Nevertheless, because of language features like exceptions, type-casts, reflection, dynamic link libraries it is impossible to guarantee a complete call graph [16].

During dynamic analysis, the test dependency graph is created by instrumenting the code and monitoring the calling relationship as they unfold during execution. Assuming that the instrumentation adds hooks on all relevant methods, the dependency graph is complete by construction. However that is only to the point of the last test-run: changes applied thereafter are neglected. Thus, even with the frequent edit-compile-run cycles inherent in modern development environments there is always a small window of changes for which the calling relationships are not yet incorporated in the call graph.

To illustrate the diversity in the current state-of-the-practice, Table I gives an overview of the continuous testing tools currently available. They are ordered by the programming language they operate on (classified as either statically typed or dynamically typed) and the test selection heuristic used. As can be seen, statically typed languages adopt a static test selection heuristic, while dynamically typed languages rely on a dynamic one. dotCover is the exception that proves the rule.

[1]For dynamically typed languages there exist type inference heuristics [15].

TABLE I
Overview Continuous Testing Plug-ins

Plug-in	Language	Test Selection
STATICALLY TYPED		
infinitest	java	static
NCrunch	C#, VB.NET	static
dotCover	C#, VB.NET	dynamic
Mighty Moose	C#	static
DYNAMICALLY TYPED		
pytest-testmon	Python	dynamic
wallabyjs	Javascript	dynamic
jest	Javasript	dynamic
Karma-Runner	Javascript	dynamic

The most recent surveys by Biswas et al. [7] and Kazmi et al [8] confirm that today it is unknown how good static and dynamic test selection heuristics perform during continuous testing. This lack of knowledge is the main motivation for our research.

III. Case Study Set Up

To determine the efficiency of available test selection heuristics from the perspective of a tool builder we conducted a case study applying SmartTest on two small single developer projects. The following subsections describe the set-up of our case study.

A. Cases Under Investigation

TABLE II
Descriptive statistics (in lines of code)

Case	Size (LOC)	Test Size (LOC)	Code Churn (LOC)	Test Churn (LOC)
LAN Simulation	1135	560 (49%)	837	464
Forward Chainer	589	173 (29%)	1111	571

The cases under investigating are two small single developer projects, created separately as educational hobby projects by two of the authors (Serge Demeyer and Stéphane Ducasse).

- LAN Simulation: A simulation of a local area network, written in a procedural style and then rewritten into an object-oriented style by means of a series of refactoring steps [17].
- Forward Chainer: A rudimentary inference engine, which repeatedly applies a series of rules (the *modus ponens*) to deduce a logical decision.

The prime reason to select these cases was our self-confidence in adopting a test-driven development style combined with deep knowledge about the changes applied and the effect these may have on the tests. As can be seen from the descriptive statistics in Table II, the projects are indeed quite small (1135 and 589 lines of code respectively) however

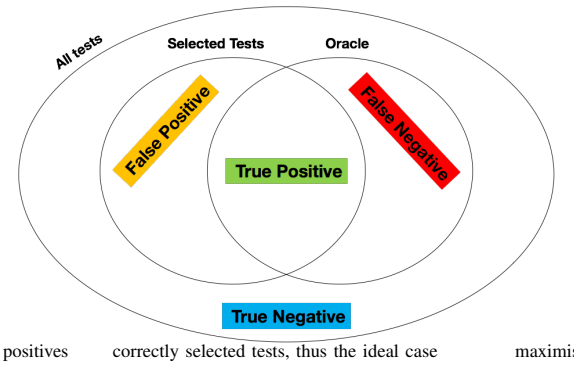
underwent quite a lot of changes (a code churn of 873 and 1111). The tests themselves where SUnit tests between 29 % and 49% of the size of the base code and co-evolved gracefully (a test code churn of 464 and 571).

B. Case Study Protocol

To collect the evaluation measurements for both the static and dynamic test selection we use the following protocol. We start with a base-line installation (an *image* in Pharo parlance) with all the necessary plug-ins installed (in particular SmartTest). Then we replay the changes twice; once for the static analysis and once for the dynamic analysis. For each change: (a) we calculate the code churn of both the base code and the test code, using the lineCount on the difference obtained by means of the utility TextDiffBuilder; (b) we apply the change; (c) run the test code selection; (d) reset the test dependency graph; (e) run the complete test suite to obtain the oracle *and* fill the test dependency graph again; (f) collect the necessary measurements (see Section III-C) and store them in an in-memory table. During this whole process we keep a detailed log of all actions for later introspection. At the end of the run, we export the data to a `.csv` file for post-processing. We use the visualisation library Roassal [18] and Excel spreadsheet to analyse the results.

C. Evaluation Metrics

To evaluate the accuracy of test selection heuristic, researchers typically compare the set of selected tests for a given change against an oracle with the set of actual tests affected by that change [7], [8]. For our evaluation, we count the number of true positives, false positives, false negatives, true negatives (see Figure 1). A good continuous testing tool (and the test selection heuristic adopted within the tool), would maximise the number of true positives and true negatives, would minimise the number of false positives (but a few can be tolerated), yet false negatives should be avoided at all cost.



true positives	correctly selected tests, thus the ideal case	maximise
false positives	tests which shouldn't have been selected, thus a waste of resources	minimise
true negatives	tests discarded correctly, thus the actual savings	maximise
false negatives	a test inappropriately discarded, thus false sense of confidence	*avoid*

Fig. 1. Evaluating Test Selection Algorithms

D. Creation of the Oracle

This is step (e) in Section III-B. To create the Oracle we rely on the TestCoverage facility which in turn is based on the Pharo meta-object protocol [9, Chapter 13]. For each individual test case, we obtain a complete trace of all methods executed by the test. We search these traces for the occurrences of the methods just changed and as such know precisely which tests should have been affected.

E. Dependency Graph: Static

The Pharo environment doesn't maintain an internal call-graph structure. Instead the environment allows to inspect the byte-code and ask for all message names send from a given method. The environment subsequently allows to query for all methods with a given name and obtain their byte-code. A transitive application of these steps allows for an on-the-fly construction of the call graph. Note that this approach is slightly less precise than for a statically typed language because one cannot rely on the type of the receiver, thus all methods with the same name are candidate targets. Experience has shown that, apart for very common method names defined high in the object hierarchy (e.g., (equals, printOn:, assert:)) this is rarely a problem. However, once you target one of such methods, the call graph easily covers the whole code base. To circumvent this, we cut the transitive process once we target a method outside of the package containing the change. Consequently, our evaluation should assess the impact of this cut-off on the test selection algorithm.

F. Dependency Graph: Dynamic

Just as with the creation of the oracle (see Section III-D) we create the dependency graph by executing the tests and analysing the resulting trace. The result is stored in a test dependency graph that must be refreshed regularly because changes may invalidate certain dependencies. As described in Section III-B, step (d), we reset the test dependency graph after each change and we fill it again with a new run of all tests, step (e). This minimises the number of false positives because then our dependency graph is always up to date. Yet allows for a few false negatives because the very last changes are not incorporated. Here as well, our evaluation should assess how often such misses occur.

IV. Evaluation

We evaluate efficiency of the available test selection heuristics within the context of a continuous testing tool, using the following three research questions.

RQ1: *REDUCTION: What is the percentage of the complete test suite that needs to be rerun?*
Approach. From Figure 1 we deduce that the reduction corresponds to the amount of *true negatives*. Nevertheless, the amount of false positives plays a role as well, as these correspond with a waste of resources. Consequently, if we see a significant amount of false positives, we investigate why the test selection makes an overestimation.

Results. Figure 2 and Figure 3 visualise the results for the two cases under investigation. The X-axis corresponds to the evolution over time, and the y-axis counts the number of true positives (green), false positives (yellow), false negatives (red) and true negatives (blue). The blue areas dominate the graphs, hence we conclude that the savings are significant. Nevertheless, there are some yellow areas as well, and that is where too many tests are selected.

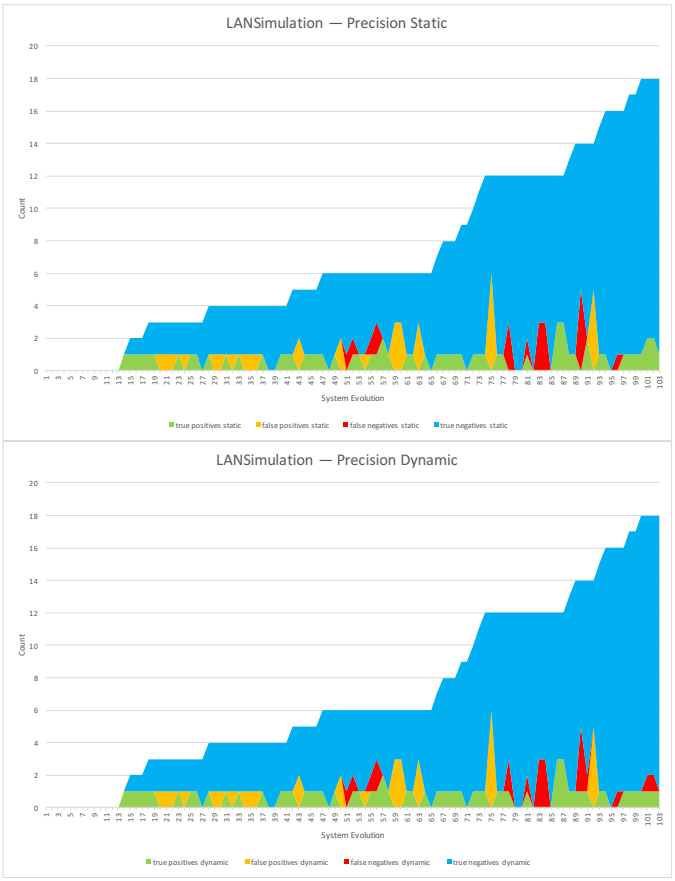


Fig. 2. Evaluation of Test Selection Algorithm on LAN Simulation

These observations are confirmed in Table III. There we see that the savings vary from 84% (LANSimulation) to 70% (ForwardChainer), however that the amount of false positives is considerable, especially for the ForwardChainer case (22% - 23%).

TABLE III
EVALUATION OF TEST SELECTION ALGORITHM

Case	true positives	false positives	false negatives	true negatives
LANSimulation				
Static	63 (9%)	35 (5%)	21 (3%)	605 (84%)
Dynamic	62 (9%)	35 (5%)	22 (3%)	605 (84%)
ForwardChainer				
Static	127 (6%)	476 (22%)	39 (2%)	1.526 (70%)
Dynamic	118 (5%)	491 (23%)	48 (2%)	1.509 (70%)

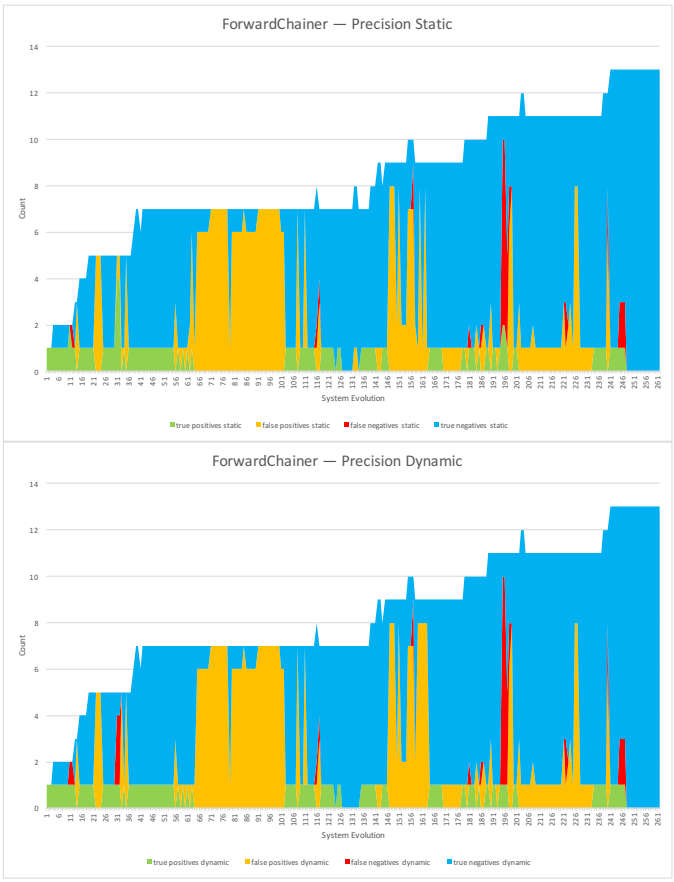


Fig. 3. Evaluation of Test Selection Algorithm on Forward Chainer

RQ2: *FALSE NEGATIVES: How many tests are discarded inappropriately?*

Approach. The false negatives should be avoided and if they occur we should investigate why the algorithm choose to discard this test case.

Results. The red areas in Figure 2 and Figure 3 are few and far between, thus we can say that both the static an dynamic algorithm perform rather well. Nevertheless, according to Table III, in 2%-3% of the cases the test selection algorithm discard pertinent tests. We investigated the reason why the test selection sometimes fails to select the appropriate tests. For the static analysis, this was due to modifications of methods with common names (printOn:, assert:). As mentioned in Section III-E, the construction of the call graph is terminated once we target methods which are defined outside the package under change, which is precisely what happens here. For the dynamic analysis, this was due to the test-driven development approach. There tests are written first (and do not pass) and afterwards the code is changed until the test pass. Thus when adding a new test, this logic is not yet included in the trace of the previous test run. Also, while modifying the code which should make the test pass again, the new execution paths will not yet be included in the trace of the previous test run.

We conclude that in rare occasions, both the static and the dynamic sometimes result in false negatives, thus discard pertinent tests. These can be attributed to way the call graphs

are constructed; which inevitably forsake some traceability links.

RQ3: *STATIC vs. DYNAMIC: Where and why do the static and dynamic analysis differ?*

Approach. To address, this question we count the number of changes where the static and dynamic analysis result in the same amount of false positives and false negatives. Afterwards, we make an in depth investigation of those changes where the number of false positives and false negatives differ.

Results. Our results show that for both cases the static and dynamic analysis select the same tests. We were surprised that most differences are in favour of the static analysis; for the LAN Simulation these are changes 101 and 102 where there is one false negative for dynamic not found with the static analysis. For Forward Chainer, the same happens with changes 11, 31, 32, and 33. The static analysis also gives more false positives (less critical) for changes 31, 63, 132, and 133; the dynamic one gives more false positive for changes 161 and 162. Further research is needed to verify how the two approaches compare in other evolution scenarios: refactoring, redesigning, adding features, ...

V. Conclusion

In this paper, we help tool builders select the most appropriate test selection heuristic in a continuous testing context. For this purpose, we created a prototype tool named SmartTest, which features both static and dynamic construction of the test dependency graph. We conducted a pilot study applying SmartTest on two small single developer projects adopting a test-driven development approach. We discovered that a continuous testing tool achieves significant reductions (between 70 and 80 %) in number of tests that need to be executed. We also observed that —due to the test-driven development approach— the static test-selection approach performed slightly better than the dynamic one, yet that there are situations where the dynamic analysis outperforms the static analysis. Nevertheless, both the static and the dynamic heuristic sometimes generate false negatives, thus in rare occasions discard pertinent tests.

Obviously, we cannot make strong conclusions on two small single developer projects. Nevertheless, based on this pilot study we conclude that further research is warranted. In the near future, we intend to investigate further using larger projects (i.e. larger in terms of code size, number of contributors and code churn) with different evolution scenarios (refactoring, adding features) and tackle additional research questions. In particular, we are interested in the size of the test dependency graph (can it be maintained in memory?), and the savings in test execution time (does the continuous testing tool provide instantaneous feedback?). We welcome feedback and suggestions on related work we might have missed and other research questions to explore.

VI. Acknowledgments

This work is sponsored (a) by the Conseil Régional Hauts-De-France; Nord Pas de Calais — Picardie; (b) via the TESTOMAT Project (ITEA project number 16032; VLAIO project number HBC.2017.0287; (c) via the Flanders Make vzw; the strategic research centre for the manufacturing industry.

References

[1] L. Smith, "Shift-left testing," *Dr. Dobb's J.*, vol. 26, no. 9, pp. 56–ff, Sep. 2001, last Accessed: December 13th, 2017. [Online]. Available: http://www.drdobbs.com/shift-left-testing/184404768

[2] D. Firesmith, "Four types of shift left testing," https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html, Mar. 2015, software Engineering Institute Insights; Last Accessed: December 13th, 2017.

[3] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.

[4] D. Saff and M. D. Ernst, "Continuous testing in eclipse," in *Proceedings ICSE 2005 (27th International Conference on Software Engineering)*. New York, NY, USA: ACM, 2005, pp. 668–669.

[5] Q. D. Soetens and S. Demeyer, "Cheopsj: Change-based test optimization," in *Proceedings CSMR 2012 (16th Conference on Software Maintenance and Reengineering)*, Apr. 2012, pp. 535 – 538.

[6] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[7] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica (03505596)*, vol. 35, no. 3, 2011.

[8] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys*, vol. 50, no. 2, pp. 29:1–29:32, May 2017.

[9] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Kehrsatz, Switzerland: Square Bracket Associates, 2009. [Online]. Available: http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf

[10] M. Dias, D. Cassou, and S. Ducasse, "Representing code history with development environment events," in *Proceedings IWST '11 (International Workshop on Smalltalk Technologies)*, 2011.

[11] K. Beck, "Simple Smalltalk testing," in *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*. Cambridge University Press, 1997, pp. 277—288.

[12] E. Engström, M. Skoglund, and P. Runeson, "Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 22–31.

[13] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[14] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.430

[15] N. Milojković, M. Ghafari, and O. Nierstrasz, "Exploiting type hints in method argument names to improve lightweight type inference," in *Proceedings ICPC 2017 (25th International Conference on Program Comprehension)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 77–87. [Online]. Available: https://doi.org/10.1109/ICPC.2017.33

[16] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse, "Test case selection in industry: An analysis of issues related to static approaches," *Software Quality Journal*, pp. 1–35, 2016. [Online]. Available: http://rmod.inria.fr/archives/papers/Blon16a-Software_Quality_Journal-Test_Case_Selection_in_Industry-An_Analysis_of_Issues_Related_to_Static_Approaches.pdf

[17] S. Demeyer, F. Van Rysselberghe, T. Gîrba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, S. D. Dirk Janssens, M. Lanza, H. G. Matthias Rieger, and M. El-Ramly, "The LAN-simulation: A refactoring teaching example," in *Proceedings IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*. IEEE Press, 2005, pp. 123–131, acceptance ratio: (13 + 13) / 54 = 48%.

[18] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval, "Agile visualization with Roassal," in *Deep Into Pharo*. Square Bracket Associates, Sep. 2013, pp. 209–239.