



MASTER RESEARCH INTERNSHIP



INTERNSHIP REPORT

Adapting Amplified Unit Tests for Human Comprehension

Domain: Software Engineering — Artificial Intelligence

Author:
Simon BIHEL

Supervisor:
Benoit BAUDRY

KTH Royal Institute of Technology



Abstract:

TODO

Contents

Introduction	1
1 Background	1
1.1 Software Testing	1
1.1.1 Test Activities	1
1.1.2 Levels of Testing	2
1.1.3 Unit Testing	3
1.2 Elementary Metrics	4
1.3 Mutation Testing	5
1.4 The Need for Easy-to-use Tools	6
1.5 Cognitive Support Tool Development	6
2 Test Suite Amplification	6
2.1 Genetic Improvement	6
2.2 Test Amplification	6
2.3 DSpot	6
3 Problem Statement	7
3.1 The Need for Unit Test Cases Documentation	7
3.2 The Generated Random Noise	7
4 Related Works	7
4.1 Automatic Test Case Documentation	7
4.2 Cognitive Support for Unit Testing Review	8
5 Contribution	8
5.1 Identifying Amplifications	8
5.2 Minimisation	8
5.3 Replace or Keep	8
5.4 Focus	8
5.5 Slicing	8
5.6 Natural Language Description	8
5.7 Ranking	9
6 Evaluation	9
6.1 Threat to Validity	9
Conclusion	9
Acknowledgments	9

Introduction

TODO

1 Background

In this section, we present the landscape this thesis fits into. We define testing (Section 1.1) as well as go over its use in the industry. In particular, to understand the needs of practitioners, we present how they assess the quality of their code (Sections 1.2 and 1.3) and what it takes for a new tool to be incorporated in their workflow (Sections 1.4 and 4.2).

1.1 Software Testing

In this section we give a definition for the test activities and their actors, the different abstraction levels of tests, and our precise object of study, unit tests.

TODO: Why are we testing software and how do we do it

TODO: Say that the formal definitions aren't totally necessary

1.1.1 Test Activities

TODO: the text is too close to the oracle survey

Testing is about verifying that a system, for a given scenario, follows a certain behaviour that was previously defined. The *System-Under-Test* (SUT), in our domain a software system, has a set of components C . A scenario is sequence of stimuli that target a subset of C , and trigger responses: (i) feedbacks from the SUT; and (ii) changes in the state of its components. From [1], we have the following definition for to combination of stimuli and responses:

Definition 1.1 (Test Activities). For the SUT p , S is the set of stimuli that trigger or constrain p 's computation and R is the set of observable responses to a stimulus of p . S and R are disjoint. Test activities form the set $A = S \uplus R$. The disjoint union is used to label the elements of A .

The use of the terms “stimulus” and “observations” fits various test scenarios, functional and non-functional. As depicted in Figure 1, a stimulus can be either an explicit *input* from the tester, or an environmental factor that can affect the SUT. Stimuli include, among others, the platform, the language version, the resources available, interaction with an interface, etc. Observations encompass anything that can be discerned and measured like the content of a database, the visual responses on a computer screen, the time passed during the execution, etc.

Testing is about stimulation and observation so we talk about *test activity sequences* that are comprised of at least one stimulus and one observation. But testing is also about verifying that the observed responses, the behaviour, match to ones that were previously defined. This checking part is done by another actor:

Definition 1.2 (Test Oracle). A test oracle $D : T_A \mapsto \mathbb{B}$ is a partial function from a test activity sequence to true or false.

An oracle can be defined using different methods: a set of specifications an expected value for a variable after the execution of a particular test activity sequence, the expectation of a lack of crash, etc.

A test oracle is a partial function because it can leave certain elements of the SUT's behaviour unspecified. This part of uncertainty is fundamental in the real world because systems have grown to be so complex that humans cannot anticipate all the reactions of the SUT in all environments. It is useful nonetheless to have a name for a theoretical oracle that would have an answer for every possible question:

Definition 1.3 (Ground Truth). The ground truth oracle, G , is a total test oracle that always gives an answer.

This allows us to define two properties for all oracles:

Definition 1.4 (Soundness). The test oracle D is sound for test activity a iff $D(a) \rightarrow G(a)$

Definition 1.5 (Completeness). The test oracle D is complete for test activity a iff $G(a) \rightarrow D(a)$

A test oracle cannot, in general, be complete as it would require handling every possible test activity sequence. It is not rare for an oracle to be unsound as humans can make errors writing specifications.

In practice, following pattern designs of modularity, the act of testing is composed of multiple test activity sequences that target specific elements of the SUT's behaviour:

Definition 1.6 (Test Case). A test case T_C is a test activities sequence that contains at least one stimulus and one observation with a test oracle that verifies each observation.

Definition 1.7 (Test Suite). A test suite is a collection of tests cases.

Another concept that we will use when talking about software evolution is regression testing [2]:

Definition 1.8 (Regression Testing). Regression testing is performed between two different versions of the SUT in order to provide confidence that newly introduced features do not interfere with the existing features.

It is a way to avoid having a human write formal specification but the problem remains of trying every possible scenario and observing any useful response.

TODO: give some clues to each concept to tell why it is a field of research?

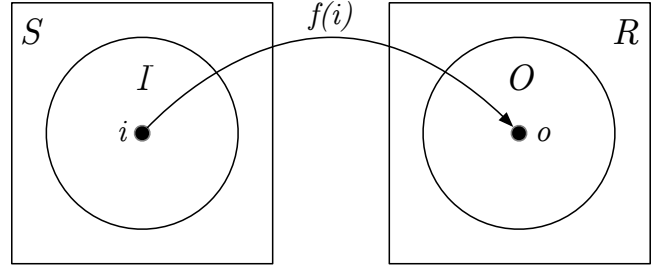


Figure 1: Stimuli and observations: S is anything that can change the observable behaviour of the SUT f ; R is anything that can be observed about the system's behaviour; I includes f 's explicit inputs; O is its explicit outputs; everything not in $S \cup R$ neither affects nor is affected by f .

1.1.2 Levels of Testing

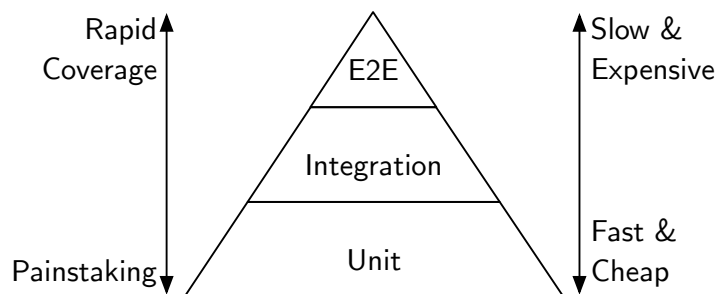


Figure 2: A view of the test pyramid.

In the case of software testing, the SUT can be different elements of a program, i.e. it can belong to different levels of abstraction. One can decide to test a whole program, by manually giving inputs and verifying that the output is correct. Or one could test individual functions, having a more control over the behaviour of each elements.

These levels of test abstraction can be visualised with a test pyramid, as shown in Figure 2. Tests are generally separated into three distinct categories. The top category is for the End-to-End tests, which means testing the program as a whole. It can mean interacting with the UI and evaluating the visual responses. Whilst it allows for directly evaluating the well behaviour of the program, each test is costly in time and resources because the whole program has to be run. Not only will each component be run, whether or not it requires testing, but time will also be spent on call to libraries that are already tested. At the other end of the test spectrum are the unit tests. The goal is to test each and every component of the program (e.g. every function), as an individual and isolated from the rest of the program. Running a single component is faster than running the whole program, but require much more work from the person writing tests, as every component needs its set of tests. In between the two testing practices are the integration tests, which aim at ensuring the well collaboration of components, like verifying that gears rotate in the right way. Examples of integration tests include API testing, or simply the fact of calling a function using the result from another function call. This last example gives an intuition on how fuzzy the distinctions between all kinds of tests are, with definitions often relying on personal opinions¹. More details about the loose definition of unit tests are given Section 1.1.3.

TODO: why we use a pyramid Practitioners often recommend putting an emphasis on unit test, with a distribution of 70% unit tests, 20% integration tests, and 10% end-to-end tests for example².

TODO: how tests are automated

1.1.3 Unit Testing

A unit test is test for a precise component of the SUT, but defining the level of granularity is not trivial, even for experienced programmers³⁴ [3]. **TODO**

¹<https://testing.googleblog.com/2009/07/software-testing-categorization.html>

²<https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>

³<https://martinfowler.com/bliki/TestPyramid.html>

⁴<https://martinfowler.com/bliki/UnitTest.html>

```

1 testIterationOrder() {
2     TreeList tl = new TreeList(10);
3     for (int i = 0; i < 10; i++) {
4         tl.add(i);
5     }
6     ListIterator it = tl.listIterator();

7     int i = 0;
8     while (it.hasNext()) {
9         Integer val = it.next();
10        assertEquals(i++, val.intValue());
11    }
12 }

```

Listing 1: Example of an object-oriented unit test (taken from the Apache Commons Collections, in the class `TreeListTest`, line 270): it consists of test inputs (lines 2–6) that manipulate the SUT; and assertions (line number 10). **TODO: what about lines 8 & 9**

An example of unit test is given in Listing 1.

TODO: why we sometimes speak about test methods

TODO: can we link blog post from google and martin fowler to say that it's a difficult engineering

TODO: evoke 'software artifact'

TODO: problems with OO, private methods, composition design patterns [4]

TODO: it's a program (and thus it carries pros and cons on comprehension and ease of use) (a

TODO: having small focused tests also reduce the likelihood of having a flaky test⁵

TODO: dependencies⁶

1.2 Elementary Metrics

Metrics have been developed to help developers automatically assess the quality of their code. Each metric measures one specific characteristic, such as the need for refactoring, the size of the program, or — and we are particularly interested in this one — the likelihood of bugs.

TODO: give examples for other metrics?

The most basic metrics to measure how thoroughly tested a system is, are coverage based metrics. For example, during the test suite execution, one can keep track of all statements that were executed. At the end, you have a percentage of executed statements for the total number of statements. Instead of statements, one can also keep track of control flow branches explored.

It is generally acknowledged that a system with high coverage means that the system is less likely to have bugs — but it is not foolproof.

TODO: What are their limits

- doesn't mean corner cases are avoided
- can't have 100% branch coverage so it's confusing

1.3 Mutation Testing

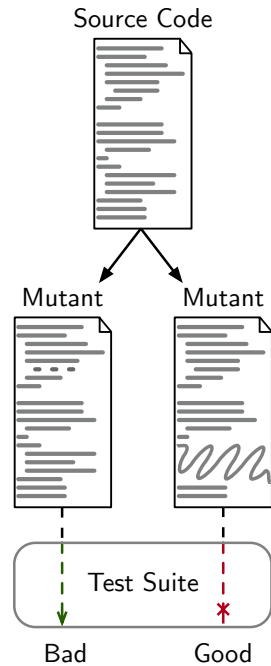


Figure 3: Mutation testing process.

Mutation testing is a method used to assess the quality of a test suite, giving it a grade called *mutation score*. The idea is to insert bugs in the software and see if these faulty versions still pass the tests or not. We call the derived versions *mutants*, hence the name mutation testing. When a mutant goes through the test suite without making a single test fail, it is considered *live*, and the opposite case it is considered *dead*. The global process is depicted in Figure 4. The main appeal is that, a test suite with a good mutation score, in addition to ensure the well behaviour of the SUT in useful scenarios, it can detect bad behaviours — which is particularly useful for regression testing. **TODO: rework that sentence** There is a correlation between mutation score and coverage score as test suites with high mutation score tend to have good coverage [6]. It has been found that correlations between mutation scores and real fault detection are weak [7].

TODO: a mutation covered line is stronger than a regular covered line [foundational papers]

Mutants are obtained by applying *mutators* on the original version. These operators can be varied: change a condition (e.g. substituting \leq with $>$), delete the body of a method **[descartes]**, etc. Most of the time, we generate 1-order mutants, that means that a mutant is the result of a mutator applied once. A mutator will usually generate several mutants. For example, a mutator that replaces a **while** statement into a **do-while** statement will generate a mutant for each **while** in the artifact. In the case of k -order mutants, which can be thought of as the result of k successive 1-order mutants[8], the number of derived programs starts to blow up.

Mutation testing suffers from drawbacks that have limited its use outside of the research world even though it has been an active research domain for many years [9]. The first obvious flaw is that it is slow. For compiled languages, all mutants have to be compiled, and that process often takes a lot of time for large programs. Then the whole test suite has to be executed for each mutant, which again is a long process. However, given certain trade-offs in terms of quality of the mutation analysis process (e.g. not generating all possible mutants), the computational complexity can be dodged [10, 11]. Optimized techniques for regression testing also exist [2], including minimisation, selection and prioritisation, which can reduce the execution time of the test suite. The mutant generation is also full of traps. The major pitfall that cannot be avoided is the equivalent mutant problem. Some mutants, although they have a different program than the original, have the same semantics. Which means that they will always pass the test suite and will not bring any insight on the SUT. The generation of mutants that are simply syntactically valid is also a non-trivial problem that requires pattern replacement instead of simple text replacement [12].

Another difficulty for mutation testing to take on in the industry is the lack of understanding by practitioners. **TODO: rework that sentence** The process of mutation and elimination can be confusing. But also, when a mutant is live, it is not always clear what actions should be taken in order to kill it. **TODO: maybe add more**

PIT⁸ [13] is a mutation testing tool for Java.

TODO: what is great about it

TODO: maybe quick comparison with other tools

TODO: it is deterministic

1.4 The Need for Easy-to-use Tools

TODO: easy to understand | TODO: useful for surveys [14]

TODO: tests are written by regular developers, even though there are engineers specialised in

TODO: hard to move the industry forward | TODO: we share similar experience with static analy

TODO: avoid false positives, angry developers, must feel they benefit from the tool, enjoy using i

TODO: *effective false positives* [17]

1.5 Cognitive Support Tool Development

TODO [18] [19]

TODO: case studies [20]

2 Test Suite Amplification

In this Section, we provide more context for this thesis' contribution. TODO

2.1 Genetic Improvement

TODO [21] [fundational papers]

TODO: part of SBSE [22]

[the surprising creativity of digital evolution?]

2.2 Test Amplification

TODO [23] [A Systematic Literature Review on Test Amplification] [24, 25, 26, 27]

TODO: avoid overfitting

TODO: often only partial coverage¹⁰

2.3 DSpot

TODO ¹¹ [Genetic-Improvement based Unit Test Amplification for Java] [28, 29, 30]
[31]

TODO: only produces 1-order mutants TODO: that's confusing to say mutants for amplified t

TODO: finding bugs is easy [32]

⁸<http://pitest.org/>

¹¹<https://github.com/STAMP-project/dspot>

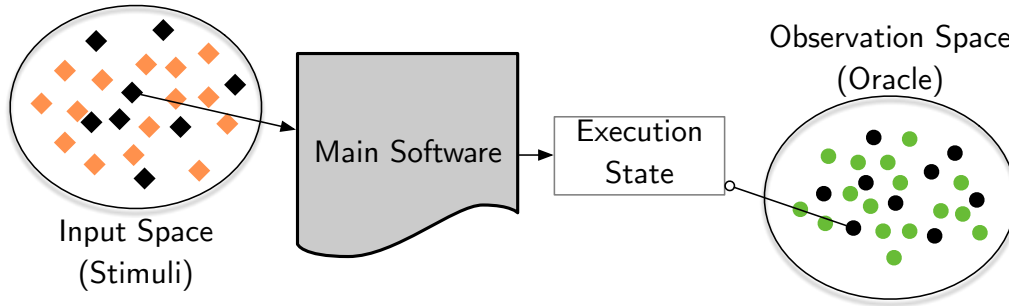


Figure 4: On the left, the testing input space is composed by specified input points (orange diamonds) and unspecified input points (black diamonds). On the right, the observation space over a given program state depicts the assertions of tests. The green circles are values that are already asserted in the existing test suite, the newly added assertions are shown as black circles.

3 Problem Statement

This thesis aim at helping developers understand amplified tests. There are two sides for this problem: supporting the developer understand the test (Section 3.1) and cleaning the tests from the noise injected during the amplification process (Section 3.2).

TODO: understanding the test is essential [15, 16]

3.1 The Need for Unit Test Cases Documentation

TODO Several works have highlighted the need for test documentation [33, 34, 35, 36, 37, 38].

TODO: tests are complex

TODO: lack of why information

TODO: the code the test interacts with; what the code does; what is expected

TODO: programmers like short textual description

TODO: explain what li2016automatically has done

TODO: talk about the field of software maintenance [39]

TODO: lack of work on generating human friendly descriptions for mutation testing

3.2 The Generated Random Noise

TODO

4 Related Works

TODO

4.1 Automatic Test Case Documentation

TODO

TODO: paraphrasing the code; lack of why information

TODO: stereotypes are for general purpose code

[40, 41, 36, 42, 43, 44]

TODO: there are also works on documenting code changes (i.e. commits) [45, 46,

47, 48, 49, 50, 51]

TODO: works on describing code written by humans [52, 53, 54, 55, 56, 57, 58, 59, 60, 61] (e.g.

TODO: Automatic summarization of natural language documents has been attempted by resear

4.2 Cognitive Support for Unit Testing Review

TODO

5 Contribution

TODO

5.1 Identifying Amplifications

TODO

5.2 Minimisation

TODO TODO: put slicing before?

TODO: removing useless assertions

TODO: cannot use general purpose techniques[63, 64] because we want the original part intact

5.3 Replace or Keep

TODO

5.4 Focus

TODO [65]

5.5 Slicing

TODO [66]¹²

5.6 Natural Language Description

TODO

TODO: Focus on mutation testing

TODO: avoid talking about mutants TODO: How to describe a mutant

TODO: SWUM [67] [68, 69, 70]

List of possible information:

¹²<http://wala.sourceforge.net/>

- control flow branches
- name of mutant
- code under test [71]
- traceability link
- label/stereotype

TODO: on the usefulness of nl

5.7 Ranking

TODO

6 Evaluation

TODO

6.1 Threat to Validity

TODO

TODO: DSpot is not yet established and recognized in the community. It's is difficult to have

Conclusion

TODO

Acknowledgments

Thanks to Benoit Baudry and Martin Monperrus for their guidance. Thanks to Benjamin Danglot for his collaboration and all his work on DSpot. Thanks to Zimin Chen, Nicolas Harrand, He Ye and Long Zhang for making daily life enjoyable. This internship was supported by the Fondation Rennes 1 and its patrons. Many thanks to KTH for hosting me.

References

- [1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.
- [2] 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.
- [3] P. Runeson, “A survey of unit testing practices,” *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

- [4] P. Wolfgang, “Design patterns for object-oriented software development,” *Reading Mass*, p. 15, 1994.
- [5] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.
- [6] B. Assylbekov, E. Gaspar, N. Uddin, and P. Egan, “Investigating the correlation between mutation score and coverage score,” in *Computer Modelling and Simulation (UKSim), 2013 UKSim 15th International Conference on*, pp. 347–352, IEEE, 2013.
- [7] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, “Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults,” in *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, 2018.
- [8] K. Wah, “A theoretical study of fault coupling,” *Software testing, verification and reliability*, vol. 10, no. 1, pp. 3–45, 2000.
- [9] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.
- [10] A. J. Offutt, G. Rothermel, and C. Zapf, “An experimental evaluation of selective mutation,” in *Proceedings of the 15th international conference on Software Engineering*, pp. 100–107, IEEE Computer Society Press, 1993.
- [11] J. Mořucha and B. Rossi, “Is mutation testing ready to be adopted industry-wide?,” in *International Conference on Product-Focused Software Process Improvement*, pp. 217–232, Springer, 2016.
- [12] A. Simão, J. C. Maldonado, and R. da Silva Bigonha, “A transformational language for mutant description,” *Computer Languages, Systems & Structures*, vol. 35, no. 3, pp. 322–339, 2009.
- [13] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “Pit: a practical mutation testing tool for java,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452, ACM, 2016.
- [14] M. Delahaye and L. Bousquet, “Selecting a software engineering tool: lessons learnt from mutation analysis,” *Software: Practice and Experience*, vol. 45, no. 7, pp. 875–891, 2015.
- [15] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world,” *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
- [16] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.
- [17] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, “Tricorder: Building a program analysis ecosystem,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 598–608, IEEE Press, 2015.

- [18] S. Oviatt, “Human-centered design meets cognitive load theory: designing interfaces that help people think,” in *Proceedings of the 14th ACM international conference on Multimedia*, pp. 871–880, ACM, 2006.
- [19] K.-J. Stol, P. Ralph, and B. Fitzgerald, “Grounded theory in software engineering research: a critical review and guidelines,” in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 120–131, IEEE, 2016.
- [20] B. Flyvbjerg, “Five misunderstandings about case-study research,” *Qualitative inquiry*, vol. 12, no. 2, pp. 219–245, 2006.
- [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] P. McMinn, “Search-based software testing: Past, present and future,” in *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pp. 153–163, IEEE, 2011.
- [23] 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.
- [24] S. Yoo and M. Harman, “Test data regeneration: generating new test data from existing test data,” *Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, 2012.
- [25] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, “B-refactoring: Automatic test code refactoring to improve dynamic analysis,” *Information and Software Technology*, vol. 76, pp. 65–80, 2016.
- [26] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, “Dynamic analysis can be improved with automatic test suite refactoring,” *arXiv preprint arXiv:1506.01883*, 2015.
- [27] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, “Automatic test case optimization: A bacteriologic algorithm,” *ieee Software*, vol. 22, no. 2, pp. 76–82, 2005.
- [28] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, “Automatic software diversity in the light of test suites,” *arXiv preprint arXiv:1509.00144*, 2015.
- [29] B. Baudry, S. Allier, and M. Monperrus, “Tailored source code transformations to synthesize computationally diverse program variants,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 149–159, ACM, 2014.
- [30] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, “Dspot: Test amplification for automatic assessment of computational diversity,” *arXiv preprint arXiv:1503.05807*, 2015.
- [31] 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*, vol. 46, pp. 1155–1179, 2015.
- [32] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

- [33] M. P. Prado, E. Verbeek, M.-A. Storey, and A. M. Vincenzi, “Wap: Cognitive aspects in unit testing: The hunting game and the hunter’s perspective,” in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pp. 387–392, IEEE, 2015.
- [34] M. P. Prado and A. M. Vincenzi, “Advances in the characterization of cognitive support for unit testing: The bug-hunting game and the visualization arsenal,” in *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*, pp. 213–220, IEEE, 2016.
- [35] M. P. Prado and A. M. R. Vincenzi, “Towards cognitive support for unit testing: a qualitative study with practitioners,” *Journal of Systems and Software*, 2018.
- [36] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, “Automatically documenting unit test cases,” in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pp. 341–352, IEEE, 2016.
- [37] E. Daka and G. Fraser, “A survey on unit testing practices and problems,” in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pp. 201–211, IEEE, 2014.
- [38] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, “The impact of test case summaries on bug fixing performance: An empirical investigation,” in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 547–558, IEEE, 2016.
- [39] E. B. Swanson, “The dimensions of maintenance,” in *Proceedings of the 2nd international conference on Software engineering*, pp. 492–497, IEEE Computer Society Press, 1976.
- [40] G. Neubig, “Survey of methods to generate natural language from source code,” 2016.
- [41] N. Nazar, Y. Hu, and H. Jiang, “Summarizing software artifacts: A literature review,” *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.
- [42] B. Li, *Automatically Documenting Software Artifacts*. PhD thesis, College of William & Mary, 2018.
- [43] M. Kamimura and G. C. Murphy, “Towards generating human-oriented summaries of unit test cases,” in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 215–218, IEEE, 2013.
- [44] M. Ghafari, C. Ghezzi, and K. Rubinov, “Automatically identifying focal methods under test in unit test cases,” in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pp. 61–70, IEEE, 2015.
- [45] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, “On automatically generating commit messages via summarization of source code changes,” in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 275–284, IEEE, 2014.
- [46] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, “Changscribe: A tool for automatically generating commit messages,” in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2, pp. 709–712, IEEE, 2015.

- [47] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, “Using stereotypes to help characterize commits,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 520–523, IEEE, 2011.
- [48] S. Jiang and C. McMillan, “Towards automatic generation of short summaries of commits,” in *Proceedings of the 25th International Conference on Program Comprehension*, pp. 320–323, IEEE Press, 2017.
- [49] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, IEEE Press, 2017.
- [50] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, “On automatic summarization of what and why information in source code changes,” in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 1, pp. 103–112, IEEE, 2016.
- [51] R. P. Buse and W. R. Weimer, “Automatically documenting program changes,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 33–42, ACM, 2010.
- [52] X. Wang, Y. Peng, and B. Zhang, “Comment generation for source code: State of the art, challenges and opportunities,” *arXiv preprint arXiv:1802.02971*, 2018.
- [53] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse engineering method stereotypes,” in *Software Maintenance, 2006. ICSM’06. 22nd IEEE International Conference on*, pp. 24–34, IEEE, 2006.
- [54] N. Dragan, “Emergent laws of method and class stereotypes in object oriented software,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 550–555, IEEE, 2011.
- [55] L. Moreno and A. Marcus, “Jstereocode: automatically identifying method and class stereotypes in java code,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 358–361, ACM, 2012.
- [56] R. P. Buse and W. R. Weimer, “Automatic documentation inference for exceptions,” in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 273–282, ACM, 2008.
- [57] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for java methods,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52, ACM, 2010.
- [58] K. Herbert, J. Goldhamer, and D. Ilvento, “Swummary: Self-documenting code,” 2016.
- [59] P. W. McBurney and C. McMillan, “Automatic source code summarization of context for java methods,” *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.
- [60] G. Sridhara, L. Pollock, and K. Vijay-Shanker, “Automatically detecting and describing high level actions within methods,” in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 101–110, ACM, 2011.

- [61] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, “Automatic generation of natural language summaries for java classes,” in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 23–32, IEEE, 2013.
- [62] K. S. Jones, “Automatic summarising: The state of the art,” *Information Processing & Management*, vol. 43, no. 6, pp. 1449–1481, 2007.
- [63] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, “Efficient unit test case minimization,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 417–420, ACM, 2007.
- [64] A. Zeller, “Yesterday, my program worked. today, it does not. why?,” in *ACM SIGSOFT Software engineering notes*, vol. 24, pp. 253–267, Springer-Verlag, 1999.
- [65] 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 *Software Maintenance, 2006. ICSM’06. 22nd IEEE International Conference on*, pp. 113–122, IEEE, 2006.
- [66] J. Dolby, S. J. Fink, and M. Sridharan, “Tj watson libraries for analysis (wala),” *URL <http://wala.sf.net>*, 2015.
- [67] E. Hill, *Integrating natural language and program structure information to improve software search and exploration*. University of Delaware, 2010.
- [68] A. Alali, H. Kagdi, and J. I. Maletic, “What’s a typical commit? a characterization of open source software repositories,” in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 182–191, IEEE, 2008.
- [69] L. P. Hattori and M. Lanza, “On the nature of commits,” in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. III–63, IEEE Press, 2008.
- [70] S. Letovsky, “Cognitive processes in program comprehension,” *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.
- [71] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, “Scotch: Test-to-code traceability using slicing and conceptual coupling,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 63–72, IEEE, 2011.