TestKnight: An Interactive Assistant to Stimulate Test Engineering

Cristian-Alexandru Botocan*
Piyush Deshmukh
Pavlos Makridis
botocan.christian@gmail.com
p.a.deshmukh@student.tudelft.nl
p.makridis@student.tudelft.nl
Delft University of Technology
The Netherlands

Jorge Romeu Huidobro
Mathanrajan Sundarrajan
Maurício Aniche
Andy Zaidman
j.romeuhuidobro@student.tudelft.nl
m.sundarrajan@student.tudelft.nl
M.FinavaroAniche@tudelft.nl
a.e.zaidman@tudelft.nl
Delft University of Technology
The Netherlands

ABSTRACT

Software testing is one of the most important aspects of modern software development. To ensure the quality of the software, developers should ideally write and execute automated tests regularly as their code-base evolves. TestKnight, a plugin for the IntelliJ IDEA integrated development environment (IDE), aims to help Java developers improve the testing process through support for creating and maintaining high-quality test suites.

Github repo: https://github.com/SERG-Delft/testknight
Jetbrains Marketplace: https://plugins.jetbrains.com/plugin/17072-testknight
YouTube video: https://www.youtube.com/watch?v=BSaL-K7ug6M

KEYWORDS

Software Testing, Developer Assistance, IDE plug-in

ACM Reference Format:
Cristian-Alexandru Botocan, Piyush Deshmukh, Pavlos Makridis, Jorge Romeu Huidobro, Mathanrajan Sundarrajan, Maurício Aniche, and Andy Zaidman. 2022. TestKnight: An Interactive Assistant to Stimulate Test Engineering. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3517052

1 INTRODUCTION

In order to ensure that software systems under development are of high quality, software engineers should write and maintain automated tests that enforce the quality of their code-base [1, 2, 4]. However, software developers generally view software testing as an arduous and dull process [5, 6, 14]. As a consequence, testing is often neglected [5, 7, 13, 17–19] and the resulting software systems are potentially riddled with bugs and defects. As software systems get increasingly integrated into all aspects of modern life, problematic software can have catastrophic consequences [12]. Therefore, it is of the utmost importance that developers actively ensure the quality of their product.

Ergo, the question arises; how can we encourage software engineers to incorporate testing into their daily workflow? *TestKnight*, a plugin for the IntelliJ IDEA IDE ensures a better experience for developers during the testing process. The plugin supports developers in three dimensions, namely: (1) coming up with test cases, (2) implementing tests, and (3) evaluating existing test suites. Concretely, the tool allows the user to quickly duplicate and adapt automated test cases, inspect how the test coverage evolves, and find assertions they can use when creating new test cases.

*The first five authors have contributed equally to this paper.



This Work is Licensed under a Creative Commons Attribution International 4.0 License.
ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3517052

2 TESTKNIGHT

We hypothesize that using TestKnight's workflow enables quick and effective test case engineering. Let's consider the following scenario that involves Alice.

Alice was recently given a piece of code developed by one of her colleagues to test. She does not understand the code, so it is difficult for her to figure out where to start. To come up with tests, Alice uses the testing checklist generation and assertion suggestion features of TestKnight. Then she proceeds to write the tests. To avoid writing a lot of boilerplate code, she uses TestKnight's test method generation feature. At the same time, to quickly create new test cases by adapting existing ones, she uses TestKnight's test duplication feature. As she adds new test cases, she uses the diff coverage feature to evaluate how effective the new tests are. During this process, she notices that some of the tests fail. To pinpoint where issues might arise, she invokes TestKnight's test traceability feature.

In the following sections, we will present the above options in more detail. Furthermore, we will also be showcasing the customizability options TestKnight offers for each of its features so that users can tailor the plugin to their preferences.

2.1 Assertion Suggestion

The assertion suggestion feature can be used to help find effective and meaningful assertions for your test cases based on the method under test (MUT). It works based on two pillars. Firstly, it analyzes

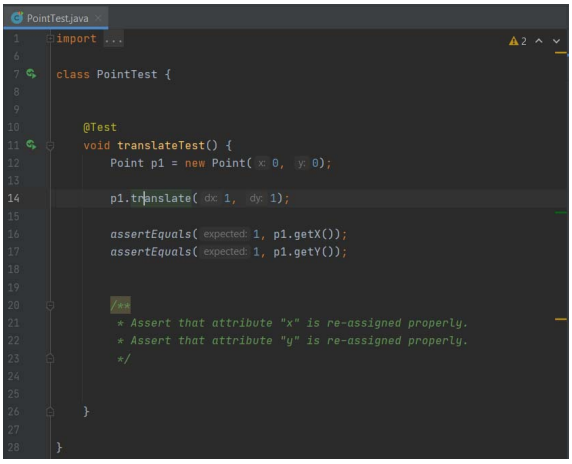


Figure 1: Example of assertion suggestion applied on p1.translate.

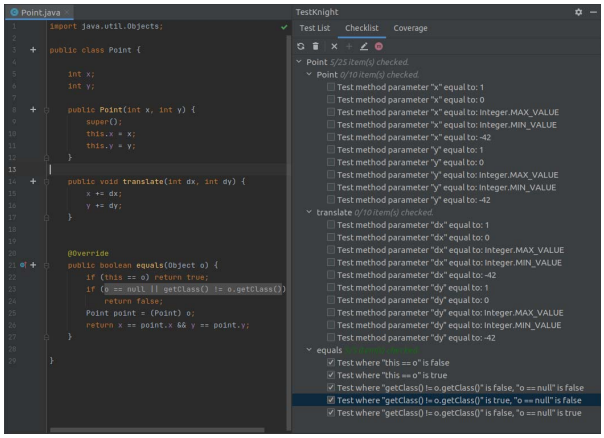


Figure 2: Example of checklist generated for class *Point*

the MUT to determine the type of output and reminds the user to assert for it. For example, if the MUT is supposed to return an integer, then the user is reminded to assert that the correct integer was returned. Secondly, it generates assertions based on the MUT's side effects, where a side effect is defined as a state change that can be observed by the code that invoked the MUT [15].

Although there are many categories of side effects [16], TestKnight focuses on only two. Specifically, it looks for class field and method argument side effects. Class field side effects occur when a method mutates at least one of the fields of the class it belongs to. A typical case of such a method is a setter, but another example method is represented in Figure 1. Similarly, argument mutation side effects occur when a method mutates one of its arguments.

2.2 Checklist

TestKnight offers the ability to generate testing checklists for methods/classes. These checklists contain the test cases which should be implemented to adequately cover the class/method according to several structural and black box testing criteria.

To generate the checklist for these test cases, TestKnight traverses the unit under test and generates different checklist items for each code construct encountered as seen in Figure 2. For example, when testing methods, the plugin suggests testing for different values of the input parameters. For `for` and `while` loops, the tool suggests implementing test cases needed to fulfill the loop boundary adequacy criterion [10]. For `if` blocks and other conditionals, the tool generates a checklist item for each MC/DC independence pair in the condition. Doing this ensures that if the test cases for each checklist item were implemented, the unit under test would have 100% MC/DC coverage.

The checklist generation is highly customizable, e.g., the user can change the parameter suggestions for methods, change the coverage criterion from MC/DC to Branch coverage and disable generating checklists for any code construct.

Once the checklist has been generated, TestKnight can autogenerate typical JUnit boilerplate code for a test case given a checklist item, thus potentially speeding up the writing of test cases.

2.3 Coverage

Figure 3 shows how TestKnight keeps track of changes in code coverage between editing actions. Currently, it only provides information about changes in line coverage. The change in coverage information can be accessed from either the gutter or TestKnight's toolwindow. New colors are added to coverage highlighters to indicate newly covered and uncovered lines. The coverage tab provides numerical information on the coverage change. A detailed *coverage diff* can be requested through the coverage table. However, this information can only be obtained if the respective source file has not been modified since the last two test runs with coverage.

2.4 Test Duplication

A recent observational study has shown that developers often engineer test cases by copying and pasting a previous test method as a way to start writing a new test method [3]. Following this, refactoring test code is also common behaviour. TestKnight offers support for the aforementioned behaviour. Specifically, when a test is selected to be duplicated through TestKnight, the selected test will be duplicated in the source code and certain elements of the method that are likely to be changed in the duplicate are highlighted.

In order to determine which sections of the method need to be highlighted, a variety of highlight-resolution strategies are implemented, which each analyze the method contents and a collection of elements that should be highlighted. In particular, there are

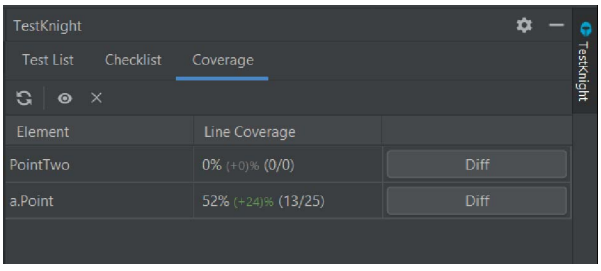


Figure 3: Coverage user interface found in the tool window.

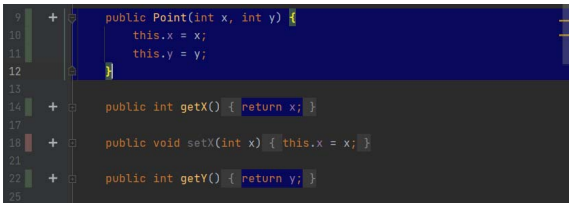


Figure 4: Example of traced code being highlighted.

strategies for finding: *Constructor arguments*, *Assertion arguments* and *Literals*. By default, all three are highlighted in addition to the method name, but the individual strategies can be enabled/disabled in the settings panel.

Lastly, we need an algorithm to resolve conflicts when several highlight resolution strategies have overlapping sections. For example, if there is a literal inside of a constructor argument there will be two overlapping highlighted elements. To resolve this, we use the *interval partitioning algorithm* to find the largest possible subset of non-overlapping highlights [11, pages 116–120].

2.5 Customizability

TestKnight aims to provide as much customizability as possible to ensure that the user is not limited by the restrictions of the plugin. These options can be found under TestKnight's settings panel.

The user can choose which sections of the test method are highlighted when it is being duplicated. Similarly, the user can choose the strategies used to generate the checklist (e.g., checklist for If Statements). Furthermore, the user can also modify the parameter suggestions for data types.

2.6 Traceability

This feature allows the user to immediately see the lines of production code covered by an individual test case, as it is shown in Figure 4. In order to get this data, whenever tests are run with coverage, the tool keeps track of the lines of code exercised by each test case [9]. Once this data is available the tool can highlight the lines specific of code which are evaluated in a given test case.

3 IMPLEMENTATION AND CHALLENGES

3.1 Coverage

A central aspect of TestKnight is differential coverage, which enables the user to see the difference between two consecutive test runs. Specifically, it enables to see in an intuitive way how the test code modification affected code coverage. This boils down to making the coverage run information stateful, which drew in some challenges. The most obvious and naïve solution would probably be persisting every coverage report and displaying two consecutive ones side by side. However, there are multiple obvious problems with this approach: a) a lot of unncecessary information is required to be persisted as we store 2 reports (previous and current) for a run even if this feature is never used, b) just showing two reports side by side is not very useful for the users to navigate between the differences lines, especially for large classes, and c) showing actual line coverage difference for better user experience would be difficult as it would involve writing a script to traverse through all lines in the source code to highlight the coverage difference alongside trying to relate a line in the previous run to the current run.

Hence, a different approach was required. We made use of the CoverageDataManager, which is an internal IntelliJ service. It maps lines of source code to the tests that cover it, which is a good representation for this cause. We attached listeners to the coverage run events which were charged with manipulating and maintaining the correct state information (e.g., switching the current coverage information to the previous coverage information just before running the test suite). And when the suite coverage data was maintained side by side with a hashmap for all classes, it is trivial and computationally cheap to produce a *diff view* for the class requested by parsing the lines only in that class and coloring them according to the scheme decided. The default one being — dark green for lines that were recently covered in the new run but not in the old test run, dark red for lines lost in the new run that were covered before, light green for lines covered in both runs, and light red for lines not covered in both. We tackled the aforementioned challenge of changes in source code by assuming no changes in between consecutive test runs. However, this may not be intuitive for some users, so we had to develop better guarantees to detect such a change in the source code file to inform the user that it has changed so the diff view would not be accurate. To do this, modification timestamps were used. These are stored alongside the coverage suite information when listeners are triggered as mentioned before. So if the timestamps do not match for a requested file, an informative warning is thrown to the user. A future version of our approach can investigate the use of Clone Region Descriptors [8].

3.2 Testing Checklist Generation

When it came to generating the testing checklist, we have encountered a few challenges related to its implementation. Firstly, the checklist generation should be efficient. Secondly, customizability to allow the user to turn on and off generating checklist items for certain language constructs.

As mentioned above, the first challenge when it came to the testing checklist generation was to ensure its efficiency. To do that, we designed the system in such a way that each source code file is only traversed once. This is achieved by using the Visitor design pattern to traverse the PSI tree generated by IntelliJ.

The second challenge we faced was keeping the checklist generation system configurable and customizable at runtime. To do this efficiently we used an adaptation of the Strategy design pattern that allows the Visitor to call upon the Strategy classes which implement the checklist generation for their corresponding code construct. With this structure in place, it was then easy for the Visitor to check for every node in the program tree that it visited whether the Strategy for it was enabled, by a simple lookup in the IDE's settings. If that is the case the Visitor performs the call to the Strategy. Otherwise, it simply moves on to the next node.

Again the Visitor pattern was useful here because it allows checking for every node in the PSI that is visited, whether the user wants checklist items for it. The latter part was achieved by looking up the settings using the IntelliJ API.

3.3 MC/DC checklist generation

To generate testing checklist items that would fully cover the unit under test with MC/DC coverage the first step is parsing the boolean expression in the condition into a syntax tree representation. From this tree, a simplified propositional expression with n propositions can be obtained by compressing all subtrees with a precedence lower than the || operator into a single proposition.

Each of these propositions is a term whose Boolean value affects the value of the overall expression. Now the expression is evaluated with all possible combinations of truth values yielding the truth table of the expression. Lastly, the truth table is traversed to find $n - 1$ independence pairs which fully cover the condition.

3.4 Side Effect Detection

Side effect detection was one of the most challenging parts to implement. Most of the traditional approaches heavily rely on the heavy use of call-graphs. This structure allows for detecting both immediate and transitive side effects. Immediate side effects are those that are immediately obvious from a single source code file (e.g., a method changes the value of some field in its class), whereas transitive side effects are side effects that occur when the MUT calls upon another method which either performs an immediate side effect or has other transitive side effects as well.

Although call-graph based approaches are highly accurate, they were deemed inappropriate for TestKnight since they become computationally expensive when analyzing large codebases. Without them, we were able to implement accurate immediate side effect detection. However for transitive side effect detection we opted for a heuristic-based approach. The heuristic we came up with was "method calls on reference types are considered side effects". With this heuristic, the tool can detect transitive side effects, although admittedly it results in some false positives.

4 USER STUDY

To assess the strengths, weaknesses and usability of TestKnight a preliminary user survey was conducted. The survey was conducted among 14 experienced programmers. We asked the participants to use the plugin on a sample project in any way they seemed fit and then fill in a feedback form. Users were also given a document showcasing the different features of the plugin.

The next part of the survey involved rating the usefulness, usability and intuitiveness of TestKnight's features. Specifically, participants were asked to rate the above qualities on a scale from 1 to 5 for all the major features of TestKnight. The boxplots in Figures 6 through 8 show respectively the participants' opinion on usability, usefulness, and intuitiveness of TestKnight's features. In summary, users generally found the features useful and easy to use. The *testing checklist* and *test list* features were rated to be the most useful, usable and intuitive overall, scoring mostly 4 and 5 in all aspects. The *tracability* feature was deemed very useful, but not as usable or intuitive to use. As Figure 5 shows, the testing checklist was also the one most liked by the participants. The assertion suggestion feature on the other hand was found to be the least intuitive, with most users giving it a 3. Additionally, the assertion suggestion was the feature least liked.

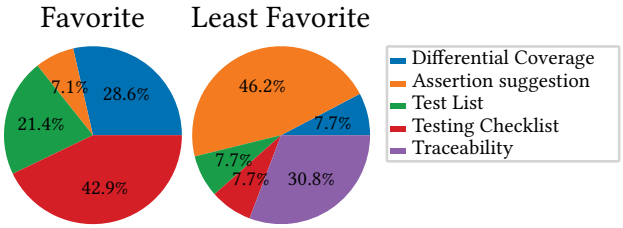


Figure 5: The results of the user survey questions "Which feature did you like the most?" and "Which feature did you like the least?", respectively

Three metrics were used to evaluate the ratings of each of the features incorporated into TestKnight: usability, intuitiveness, and usefulness. Usefulness could be explained as a measure of effectiveness to remedy the root problem the individual feature is aimed at if it would be implemented perfectly. Intuitiveness is a measure of how easy it is for a new user to navigate the UI and use the functionality. Usability refers to the practical ease of application in regular development scenarios. As can be seen from the plots in Figures 6, 7 and 8, the testing checklist received consistently high scores for all three metrics, probably making it to be the most liked feature. Assertion suggestion on the other hand received lower scores for intuitiveness, which indicates why it was relatively low-ranked among other features. Also, the usefulness of this feature was commonly remarked about in the reviews because it did not highlight rather non-trivial side effects. The responses mention go-to as a simple and effective feature to save time and was liked as a "shortcut". One of the respondents mentioned differential coverage to be their favorite feature because it helped them achieve standard testing requirements that are often enforced. In contrast, however, another developer mentioned that differential coverage was not the most useful for their use case because they rigorously practiced testing the more vulnerable aspects of the code and did not mind lower coverage statistics. They did agree, however, that the coverage would be useful in development scenarios after rigorous testing to help additionally achieve more coverage so that even the less "vulnerable" code blocks are minimally tested. Test traceability received mixed reviews in terms of intuitiveness, because of the procedure required to run it (some settings need to be in place beforehand), however, the usefulness score was quite high.

At the end of the survey, the participants were asked to give an overall rating of TestKnight. On average the plugin scored 4.35 out of 5. The exact distribution of the scores given is showcased in Figure 9. The average score indicates that TestKnight fulfils developer needs to a good extent.

ACKNOWLEDGMENTS

The authors would like to thank Pouria Derakhshanfar and Mark Swillus for their guidance during development, as well as the participants of our user study. This work was partially sponsored by the Dutch science foundation NWO through the Vici "TestShift" project (No. VI.C.182.032), and the Swiss National Science Foundation through SNF Project No. 200021M 205146.

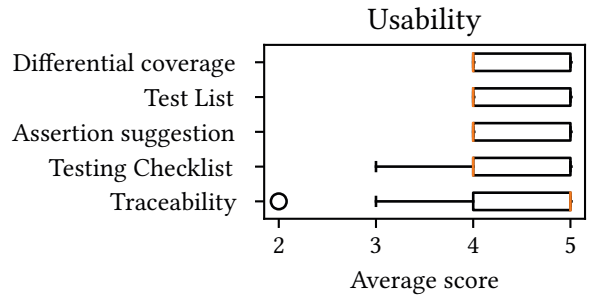


Figure 6: Boxplot of the usability of TestKnight's main features

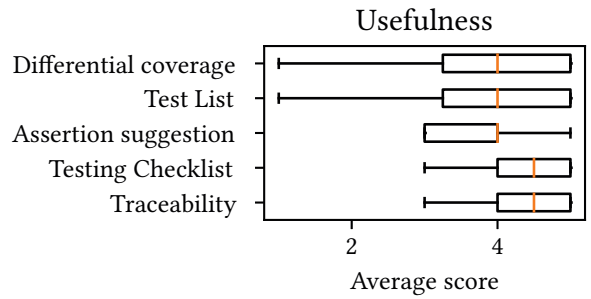


Figure 7: Boxplot of the usefulness of TestKnight's main features

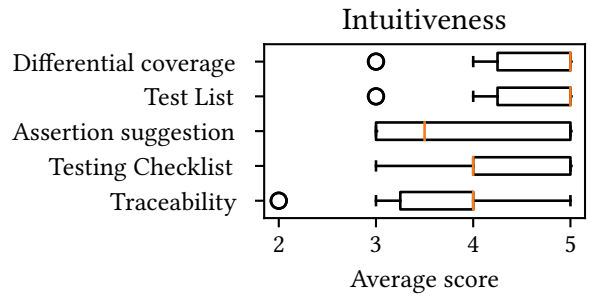


Figure 8: Boxplot of the intuitiveness of TestKnight's main features

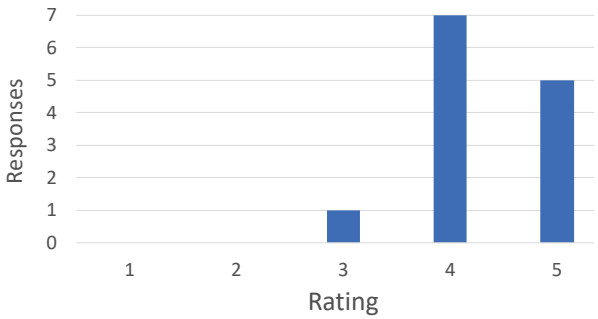


Figure 9: The results of the user survey question "How would you rate TestKnight?"

REFERENCES

[1] Maurício Aniche. [n.d.]. *Why software testing?* https://sttp.site/chapters/getting-started/why-software-testing.html
[2] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Manning.
[3] Maurício Aniche, Christoph Treude, and Andy Zaidman. [n.d.]. How Developers Engineer Test Cases: An Observational Study. *IEEE Trans. on Softw. Engineering* ([n. d.]). *To Appear.*
[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Trans. Software Eng.* 40, 11 (2014), 1100–1125.
[5] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.
[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* ACM, 179–190.
[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE Volume 2)*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE, 559–562.
[8] Ekwa Duala-Ekoko and Martin P. Robillard. 2010. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.* 20, 1 (2010), 3:1–3:31.
[9] Victor Hurdugaci and Andy Zaidman. 2012. Aiding Software Developers to Maintain Developer Tests. In *16th European Conference on Software Maintenance and Reengineering (CSMR).* IEEE, 11–20.
[10] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proc. Int'l Conf. on Software Engineering (ICSE).* IEEE, 191–200.
[11] Jon Kleinberg and Tardos Éva. 2005. *Algorithm design.* Pearson.
[12] Amy J Ko, Bryan Dosono, and Neeraja Duriseti. 2014. Thirty years of software problems in the news. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering.* ACM, 32–39.
[13] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. 2008. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In *Software Evolution.* Springer, 173–202.
[14] Marc Rettig. 1991. Practical programmer. *Commun. ACM* 34, 5 (1991), 25–29.
[15] Atanas Rountev. 2004. Precise identification of side-effect-free methods in Java. In *Proc. Int'l Conf. on Softw. Maintenance (ICSM).* 82–91.
[16] David A Spuler and A Sayed Muhammed Sajeev. 1994. Compiler detection of function call side effects. *Informatica* 18, 2 (1994), 219–227.
[17] Chak Shun Yu, Christoph Treude, and Maurício Finavaro Aniche. 2019. Comprehending Test Code: An Empirical Study. In *International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 501–512.
[18] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. 2008. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *First International Conference on Software Testing, Verification, and Validation (ICST) 2008.* IEEE, 220–229.
[19] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empir. Softw. Eng.* 16, 3 (2011), 325–364.