DARTS - Contextual Analysis

Andrea Cupito
Università degli Studi di Salerno
a.cupito@studenti.unisa.it

Giosuè Sulipano
Università degli Studi di Salerno
g.sulipano@studenti.unisa.it

Fabio Palomba
Università degli Studi di Salerno
f.palomba@unisa.it

ABSTRACT

Mining software repositories (MSR) is a software engineering field which consists of extracting useful and actionable information produced during the development process by mean of data mining techniques.

Types of software repositories include source control repositories, bug repositories, code repositories and even archived developer communications including mailing lists.

We focused our attention on *Test Smells* which represent sub-optimal design choices in the implementation of test code. As reported by recent studies, their presence might not only negatively affect the comprehension of test suites, but can also lead to test cases being less effective in finding bugs in production code.

In this paper, we investigate the relationship between the presence of test smells and the change- and defect-proneness of test code, as well as the defect-proneness of the production code being tested. To this aim we propose an improvement on Darts, an IntelliJ plug-in able to detect and automatically refactor three type of test smells, enabling it to analyze the history of production classes being tested with smelly test classes.

KEYWORDS

Test Smells, Data Mining, Software repositories, Commit

ACM Reference Format:
Andrea Cupito, Giosuè Sulipano, and Fabio Palomba. 2020. DARTS - Contextual Analysis. In *Proceedings of Course of Software Dependability 2020 (SD'20).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nevertheless, when developing test cases, programmers sometimes introduce sub-optimal implementation solutions that have the potential impact of reducing test code understandability, maintainability, and effectiveness: these are called *test smells.* Previous research has investigated these smells under different angles, highlighting that tests affected by them have reduced fault detection capabilities, other than requiring more effort to be maintained and/or enhanced. We can asses that their presence might not only negatively affect the comprehension of test suites, but can also lead to test cases being less effective in finding bugs in production code.

SD'20, February - July 2020, Fisciano, Italy,
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The earliest and most significant results advancing our empirical knowledge on the effects of test smells was presented by Bavota et al. [8]. They conducted the first controlled laboratory experiment to establish the impact of test smells on program comprehension during maintenance activities and found evidence of a negative impact of test smells on both comprehensibility and maintainability of test code [8]. A more recent study conducted and reported by Palomba *et al.* [3], strongly highlights the relationship between test smells and production code, in fact they definitively confirm that tests with smells are more change- and defect-prone than tests without smells and production code is more defect-prone when tested by smelly tests

In order to study the co-evolution of production and test code, we rely on the data that is stored in Version Control Systems (VCS's). Version Control Systems is a category of software tools that help a software team manage changes to artifacts, e.g. source code, over time. It keeps track of every modification in a kind of database. Using a VCS to study co-evolution implies the prerequisite that the code should be committed to the VCS.

Our work focuses on DARTS (Detection And Refactoring of Test Smells) [1], an open-source IntelliJ plug-in which implements a detection mechanism to detect instances of three test smell types, i.e., General Fixture, Eager Test, and Lack of Cohesion of Test Methods, at commit-level and enables their automated refactoring through the integrated APIs provided by IntelliJ. We are going to exploit the capability of DARTS to work during the commit phase and with the commits themselves, so we would implement a new feature that enables the plug-in to analyze the VCS change history of the project in order to asses if test smells detected are actually harmful or not and, at last, providing a brief summary in which we expose the result of the analysis.

In this paper, we focus on how production code evolve with regard to the related test code, and how bad design choices made during the writing of test classes, e.g. test smells, could impact on the behavior of production code. In order to evaluate the defect- and change-proneness of the target class we count the number of committed bug fixing activity made on the class under analysis.

The *goal* of our study is to assess whether and to what extent test methods affected by test smells are associated with the defect-proneness of the production code they test and, also we want to improve the developers' awareness about the danger related to test smells.

2 STATE OF THE ART

Writing tests, however, is as challenging as writing production code and test code should be maintained with the same care given to production code [4]. Nevertheless, several studies found that developers perceive and treat production code as more important than test code, thus generating quality problems in the tests [5], [6].

This finding is in line with the experience reported by van Deursen *et al.*[7], who described how the quality of test code was "*not as high as the production code [because] test code was not refactored as mercilessly as our production code*".

Since the inception of the concept of test smells, introduced by van Deursen *et al.*[7], this has gained significant traction both among practitioners and the software engineering research community [8], [9].

The earliest and most significant results advancing our empirical knowledge on the effects of test smells was presented by Bavota et al. [8]. They conducted the first controlled laboratory experiment to establish the impact of test smells on program comprehension during maintenance activities and found evidence of a negative impact of test smells on both comprehensibility and maintainability of test code [8].

In conclusion, testing is nowadays considered to be an essential process for improving the quality of software systems; unfortunately, in literature there are so many studies showed that test code can often be of low quality and may contain design flaws.

3 SUBJECT OF THE STUDY

In our study we have to select two types of subjects: software systems and test smells.

Software System: In our study we have selected jpacman-framework project, that is a project used for teaching software testing. The selection is driven by two main factors: firstly, since we have to run static analysis tools to detect test smells and extract dynamic metrics, we focus on projects whose source code is publicly available; secondly, we analyze a system that have the Maven structure in order to use DARTS.

Test Smells: as subject test smells for our study, we consider those detected by DARTS:

- *General Fixture*: there is a General Fixture when the fixture (initialization of setup variables) of a test suite is too general and the test methods in the suite only use a part of it.
- *Eager Test*: it occurs when a test method exercises more than one method of the production class under test.
- *Lack of Cohesion of Test Methods*: there is a Lack of Cohesion of Test Methods when there is low cohesion between the test methods in a test suite.

4 DATA EXTRACTION

To achieve our main goal we extract data about the change- and defect- proneness of the production code exercised by the test code. The extracted data used to analyze the results are available in a final summary or in a CSV file created during the analysis.

Detecting Test Smells. We adopt the test smell detector by Lambiase *et al.* [1], which is able to reliably identify the three smells considered in our study by exploiting on textual and structural rules.

Defining the defect-proneness of production code. To compute the change- and defect-proneness of each production class, we mine the change history information of the subject system using RepoDriller [15], a Java framework that allows the extraction of information such as commits, modifications, diffs, and source code; after we follow a procedure able to calculate the number of buggy commits in the project repository. In particular, we first determine whether a commit fixed a defect employing a textual based technique, which is based on the analysis of commit messages. If a commit message contains keywords such as 'bug', 'fix', or 'defect', we consider it as a bug fixing activity. A similar approach has been extensively used in the past to determine bug fixing changes [11], [10] and it has an accuracy close to 80% [12], [13]. Once we have detected all the bug fixing commits involving a a production class we use a simple HashMap to collect and count the number of bug fixing activities that involve the selected class. It's important to specify the fact that we "limited" the analysis for commits that only involve .java classes, in order to avoid any waste of time.

For what concern the effectiveness of the test cases, at the end of the analysis, we will provide a short summary which exposes some dynamic metrics, which can be useful to delineate whether and which tests would deserve maintenance operations.

Calculating dynamic metrics: We adopt VITRuM by Pecorelli *et al.* [14], which is able to calculate different types of metrics. Specifically, we calculate the so called "dynamic metrics", which provide information about the effectiveness of tests. The list of extracted metrics is above:

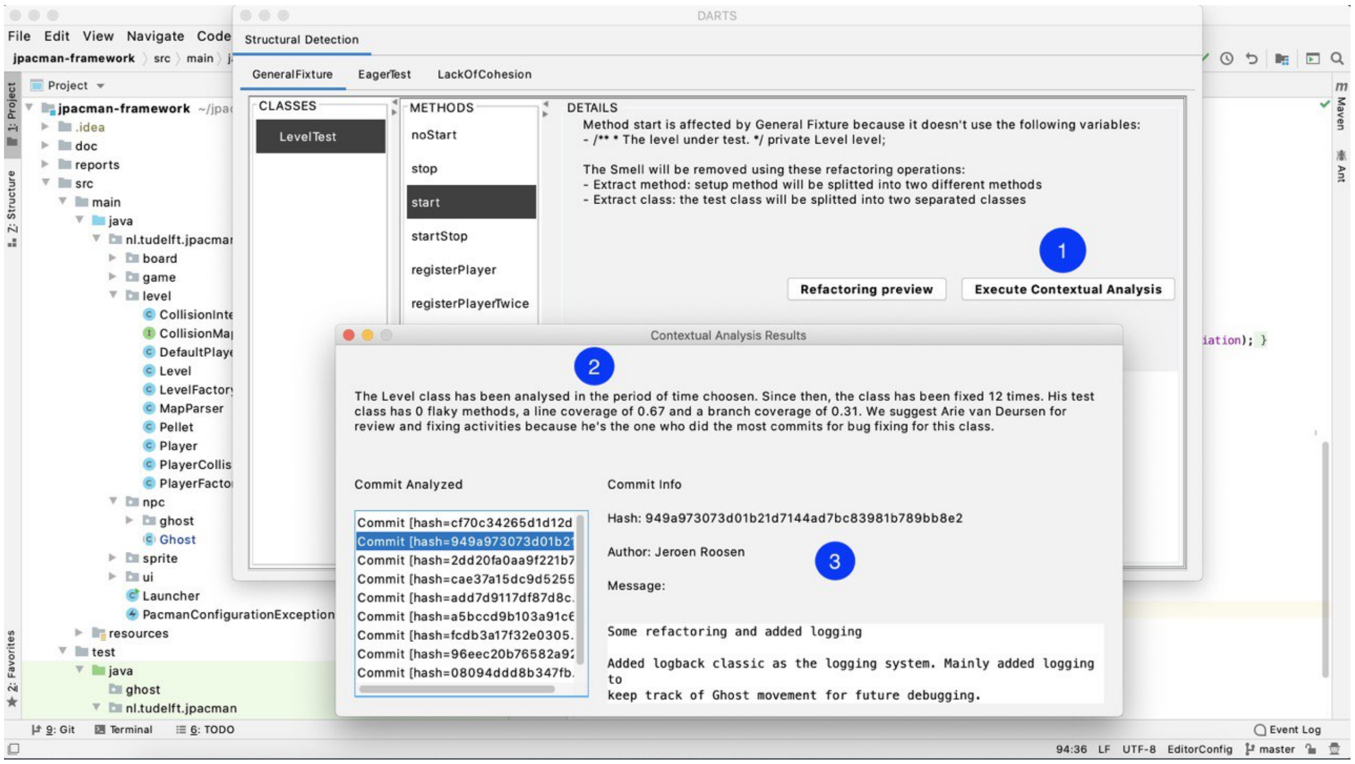
- **Branch & Line Coverage**:, calculated as the percentage of lines/branches exercised by the test with respect to the total number of lines/branches in the production class.
- **Flakyness**: Flaky tests are tests exhibiting a non-deterministic behavior (i.e., they can pass or fail with the same input). In order to detect the presence of flaky tests, we execute each test 10 times: if the output is different in at least one case than the test is flaky.

5 USE CASE SCENARIO

In this section we report a use case scenario of how to execute the contextual analysis with Darts.

The Figure 1 depicts the report shown after the computation of the analysis on the example project JPacman framework.

In particular, the plugin provides the developer with three main tabs, one for each supported test smell type. For each of them, Darts shows three pieces of information. Under the 'Classes' tab, it indicates which test suite contains at least one instance of test smells, while under the 'Methods' tab the developer can see the list of test methods involved in a smell. More importantly, the right-hand side of the window allows the developer to directly analyze the smelly test method as well as gather tips on how to refactor it. The template used to give tips is automatically adapted based on the type of test smell affecting the test under review. On the one hand, the explanation of the smell comes directly from the analysis of how it is detected. At this point users can select the button "Execute Contextual Analysis" (step 1 in the figure) to define the details about the data mining process and execute it. At its completion Darts shows the window, named "Contextual Analysis Results", in which



Method start is affected by General Fixture because it doesn't use the following variables:
- /** * The level under test. */ private Level level;

The Smell will be removed using these refactoring operations:
- Extract method: setup method will be splitted into two different methods
- Extract class: the test class will be splitted into two separated classes

The Level class has been analysed in the period of time choosen. Since then, the class has been fixed 12 times. His test class has 0 flaky methods, a line coverage of 0.67 and a branch coverage of 0.31. We suggest Arie van Deursen for review and fixing activities because he's the one who did the most commits for bug fixing for this class.

```
Commit [hash=cf70c34265d1d12d
Commit [hash=949a973073d01b21
Commit [hash=2dd20fa0aa9f221b7
Commit [hash=cae37a15dc9d5255
Commit [hash=add7d9117df87d8c.
Commit [hash=a5bccd9b103a91c6
Commit [hash=fcdb3a17f32e0305.
Commit [hash=96eec20b76582a92
Commit [hash=08094ddd8b347fb.
```

Hash: 949a973073d01b21d7144ad7bc83981b789bb8e2

Author: Jeroen Roosen

Message:

```
Some refactoring and added logging

Added logback classic as the logging system. Mainly added logging
to
keep track of Ghost movement for future debugging.
```

Figure 1: DARTS - Main window showing the results of Contextual Analysis process.

the user can get familiar with all the metrics calculated (step 2 in the figure). It's important to note that the template used to give the results is automatically adapted based on the production class under review, our strategy for text generation is to define templates of natural language sentences, and use the output from the contextual analysis to fill these templates; in the example shown in the Figure 2, the *Level* class has been fixed 12 times and its test class expose a line coverage of 0.67 and a branch coverage of 0.31.

For this task we rely on the study made by McBurney *et al.* [16], in which they have presented SWUM "Software Word Usage Model" a technique for representing program statements as sets of nouns, verbs, and prepositional phrases. SWUM works by making assumptions about different Java naming conventions, and using these assumptions to interpret different programs statements and extract noun phrases. One strategy for using SWUM for text generation is to define templates of natural language sentences, and use the output from SWUM to fill these templates. Be aware of the fact that we just exploit this technique but we haven't used SWUM in our work. On the other hand users can also see the details of every single commit analyzed (step 3 in the figure). All information reported to users improve the knowledge about their software and mainly the effectiveness of the test suites; in order to make a reasonable decision about the execution of the automatic refactoring, proposed by DARTS, of the smelly classes.

6 ARCHITECTURE OF THE TOOL

Figure 2 overviews the architecture of DARTS, which has been developed through the INTELLIJ and JAVA 8 APIs. It consists of two main layers which are described in the paper by Labiase *et al.* [1]. In the following we are describing the main changes made on the architecture in order to accomplish our objectives:

- **Interface Layer**. The interactions with the user as well as the logic pertaining to the creation of the contents to be displayed are managed in this layer. The *windowCommitConstruction* package is the one responsible to create and manage all users' interfaces, in here we define *contextualAnalysisPanel* package responsible for the definition of the UI components that contains the information resulting from the classes' analysis.
- **Application Layer**. The whole logic of the plugin is placed in this layer. Here we implemented *ContextualAnalysis* package which contains the data extraction logic. This subsystem heavily rely on RepoDriller API, allowing the plug-in to clone the remote repository in order to manipulate the repository history and collect useful data about it. The core of this subsystem is the *DataMiner* class which implements the *Study* interface that provide methods able to execute the analysis. From a practical perspective, this interface expose the "execute" method in which we have configured our study, the project to analyze, metrics to be executed, and output files. It's important to note that the classes in this package are

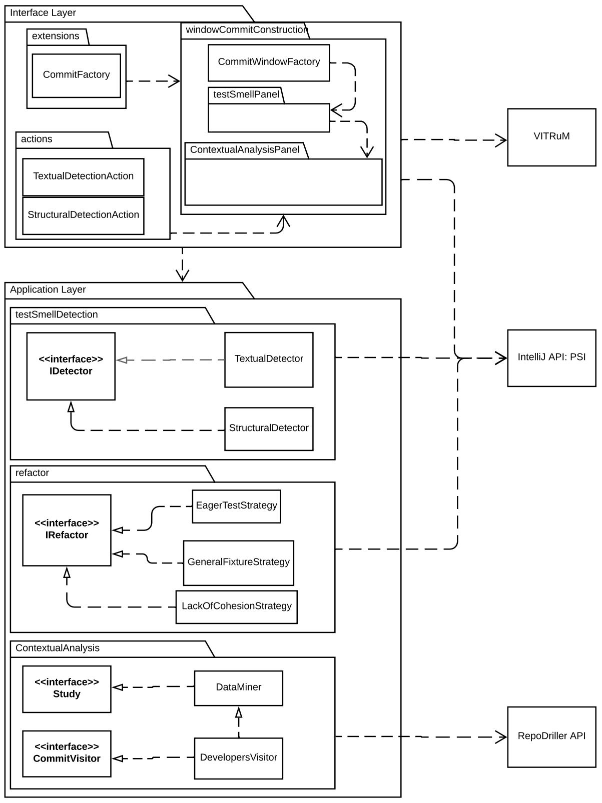


Figure 2: Architecture of DARTS

called once the detection phase is completed, because we need to assess if there are smelly test classes in the project in order to perpetrate the contextual analysis.

7 CONCLUSIONS AND FUTURE WORKS

In this paper, we presented a Darts' improvement able to exploit data stored in project's repositories in order to improve the knowledge of the users about the real effectiveness of their test suites, giving some useful information disposable by users to better decide the following development steps.

We made the tool publicly available and open-source with the aim of encouraging the research community in further improving our tool with additional test smells, detectors, and/or refactoring recommendations. Our future research agenda includes (1) the integration of other test smells' metrics, e.g., Assertion Roulette and (2) the evaluation of how Darts is used in practice: to this aim, we plan to release the tool in the Intellij plugin store and make it downloadable by developers.

REFERENCES

[1] Steafano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia and Fabio Palomba. *Just-In-Time Test Smell Detection and Refactoring: The DARTS Project.*. In Proceedings of Seoul '20: ICPC International Conference on Program Comprehension (Seoul '20)

[2] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen *Mining Software Repositories to Study Co-Evolution of Production & Test Code.*

[3] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink and Alberto Bacchelli *On The Relation of Test Smells to Software Code Quality.*

[4] S. Berner, R. Weber, and R. K. Keller *Observations and lessons learned from automated testing* In Proceedings of the International Conference on Software Engineering (ICSE), pages 571–579. ACM, 2005.

[5] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman. *Developer testing in the IDE: Patterns, beliefs, and behavior* IEEE Transactions on Software Engineering.

[6] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. *When, how,and why developers (do not) test in their IDEs.* In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pages 179–190. ACM, 2015.

[7] A. van Deursen, L. Moonen, A. Bergh, and G. Kok. *Refactoring test code* In Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP), pages 92–95, 2001.

[8] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. *Are test smells really harmful? An empirical study.* Empirical Software Engineering, 20(4):1052–1094, aug 2015.

[9] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey. *Strategies for avoiding text fixture smells during software evolution* In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), pages 387–396. IEEE, 2013.

[10] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha,and N. Ubayashi. *A large-scale empirical study of just-in-time quality assurance* A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 39(6):757–773June 2013.

[11] S. Kim, E. J. Whitehead, and Y. Zhang. *Classifying software changes: Clean or buggy?* IEEE Transactions on Software Engineering, 34(2):181–196, 2008.

[12] M. Fischer, M. Pinzger, and H. Gall. *Populating a release history database from version control and bug tracking systems* In Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, pages 23–32. IEEE, 2003.

[13] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. *The scent of a smell: An extensive comparison between textual and structural smells* An extensive comparison between textual and structural smells. IEEE Transactions on Software Engineering, 2017

[14] Fabiano Pecorelli, Gianluca di Lillo, Fabio Palomba, and Andrea De Lucia. *2020. VITRuM - A Plug-In for the Visualization of Test-Related Metrics*

[15] M. Aniche. Repodriller. https://github.com/mauricioaniche/repodriller, 2012.

[16] Paul W. McBurney and Collin McMillan *Automatic Source Code Summarization of Context for Java Methods*