A Tool to Measure TDD Compliance: A Case Study with Professionals

Altieres de Matos[1], Reginaldo Ré[2](✉), and Marco Aurélio Graciotto Silva[1,2]

[1] Graduate Program in Informatics (PPGI), Federal University of Technology – Paraná (UTFPR), Cornélio Procópio, Paraná, Brazil
altitdb@gmail.com

[2] Department of Computing (DACOM), Federal University of Technology – Paraná (UTFPR), Campo Mourão, Paraná, Brazil
{reginaldo,magsilva}@utfpr.edu.br

Abstract. Context: There are several studies related to Test Driven Development (TDD), but many with divergences of results due to the short time to perform the experiments. Moreover, the environment where they are carried out is generally academic. On the other hand, the environment requires tools not used by practitioners or imposes many technical and training requirements for their application. **Goal:** The goal of this paper is to provide a tool that supports the evaluation of the TDD process in the software industry and academia settings. The tool focuses on analyzing the effects of verification, validation and test (VV&T). In addition, the compliance of TDD usage in software development was evaluated. **Method:** This study made use of the Goal Question Metric (GQM) paradigm to characterize a set of objectives using metrics towards TDD effects on software quality. A case study was conducted with IT professionals to evaluate the tool developed. **Results:** Considering the existing tools that perform TDD compliance assessment, the Butterfly tool was developed to enable the evaluation of the TDD lifecycle as the developer performs the coding of the software. With this tool it is possible to analyze the compliance of TDD usage during software development. **Conclusions:** The tool allows to measure the effects of TDD when developing software, which will support in the characterization of TDD contributions and interventions applied to software quality in future works.

Keywords: Test driven development · Agile software development · TDD conformance · TDD lifecycle · Software measurement

1 Introduction

Test Driven Development (TDD) provides developers with the ability to write small pieces of software based on software requirements, implementing test cases before production code [9,20]. Thus, each piece of code and its respective automatic test is written in a cycle [9]. This style of development enables developers to stay focused on requirements sets, ensuring that every piece of written production code is covered by automated tests [9,20].

TDD is considered an agile practice related to quality [6,13]. With the introduction of TDD in eXtreme Programming (XP) and its frequent use in conjunction with the Scrum agile method [12], TDD has gained popularity [2,6]. However, despite its adoption in software industry, several aspects regarding TDD, such as conformance and software testing activities, are not focused by the research community [16].

The motivation of this study lies in the fact that the software industry and academia are not strongly connected and do not have high collaboration between them [5]. In the software testing domain, one of the main problems is that researchers are not worried about solving problems of the software industry [10]. Several reasons regarding this have been discussed by software engineering researchers, from the difference of objectives between the two parties to challenges of scalability and applicability of problems [10].

One of the current discussions in the area of software testing is the compulsory testing of software in the scope of software development [5,18]. Another motivating factor is the increase in the interest of professionals in test automation [18]. In contrast, there are factors that limit the adoption of TDD in the software industry: (i) increased development time, (ii) insufficient experience and knowledge about TDD, (iii) lack of upfront design, (iv) insufficient developer testing skills, (v) lack of adherence to the TDD protocol, (vi) limitations regarding TDD implementation related to domain and tools, and (vii) legacy code [7]. In addition to these factors, it is observed that, currently, the adoption of agile methods requires that the responsibility for software quality be made beyond the quality team [8]. If, in agile methods, it is argued that teams have autonomy and that they are responsible for the software, it is sensible to give them greater responsibility for software quality, rather than delegating such a role integrally to a distinct part (such as a quality assurance team).

Although global software development industry and the software research academy have a large number of members, collaboration between the two is low [10]. In 2017, the state of the art software testing considered that performing manual or automatic testing became mandatory for the production of software products [5]. In this way, adopting TDD makes it possible to go further [20]. TDD provides the industry the possibility to improve adherence to proper software testing activities, minimizing the chances of skimping on the implementation of test cases after writing production code [20].

This study aims to provide a tool to support the evaluation of the TDD process in the academia and software industry. A fundamental aspect of the study is the application in the software industry encompassed in the agile context along with the use of the iterative model. The tool allows the real-time analysis of coding performed by developers, classifying their actions regarding editing test cases and code and, considering such actions, classify the development cycle/iteration in: (i) test-addition, (ii) test-first, (iii) test-last, (iv) test driven development, (v) refactoring e (vi) unknown. Thus, conformance to TDD can be evaluated through the classifications and other measurements regarding testing activities, such as test coverage and quantity of implemented test cases.

Some studies that evaluate TDD usage in the software industry and TDD conformance are described in Sect. 2. Considering their findings and the limitations regarding tool support for TDD adoption, the tool, named Butterfly, is described in Sect. 3. Afterwards, in Sect. 4, it was provided details of the case study used to evaluate the developed tool, describing the results in Sect. 5 and discussing how the tool can influence the community in Sect. 6. Conclusions and next steps regarding the investigation on the integration of TDD and software testing within the industry setting are presented in Sect. 7.

2 Related Work

Test Driven Development (TDD) [3] is an iterative software development technique [9,19]. In the TDD process, each new iteration consists of the implementation of a feature [9]. Three phases make up the TDD process: (i) writing the unit test, (ii) implementing the production code, and (iii) refactoring [9,19]. The iteration begins with writing the unit test, followed by the implementation phase of the production code, and finalizing itself in the refactoring phase [9,19]. The iteration is terminated when all phases of the process are executed and the unit tests are successfully executed [9]. The main rule of TDD is: *"If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding"* [11].

TDD and some of its effects have been extensively studied [9]. Considering the objective of this study, it was focused on related work regarding TDD conformance, helping developers with the use of TDD and improving software design, so that it was possible to obtain the best solution for the desired scenario [17].

In the study by Kou et al. [15], the authors presented a tool that allows automated recognition of TDD. The so called Zorro system allows the operational definition of TDD practices to be verified. The automated recognition of TDD can bring several benefits to the community, either to support TDD practices or to assist in empirical studies on the effectiveness of TDD. The study described how the analysis can be performed with the Zorro system, in addition to two empirical assessments. The first controlled experiment aimed to ensure that the collected and analyzed information was appropriate and effective. In the second controlled experiment conducted by the authors, they aimed to obtain better data about the strengths and limitations of Zorro for TDD inference. A third controlled experiment was conducted in order to address a validity threats. Thus, the authors did not use students as subjects, as in the two previous controlled experiments, besides not using the classroom environment. Instead, the authors run the controlled experiment with professionals of the software industry. The study fostered the possibility of tool evolution, in order to improve its ability to recognize TDD processes, in addition to providing information in a clear and objective way. It also made it possible to evaluate the effects of TDD in the medium and long term with respect to software quality.

In the study by Becker et al. [4], the authors presented the Besouro tool, which is an improved version of their previous TDD automatic recognition tools and studies. The following tools were used: (i) TestFirstGauge, (ii) TDDGuide, (iii) Zorro and (iv) SEEKE. The authors compared them with other existing tools and commented on the new features existing in the Besouro, some of them being built with private (closed-source) components. The Besouro tool shares several concepts used in the Zorro tool of the study by Kou et al. [15]. To verify the effectiveness of the tool, the authors performed a controlled experiment that was defined through the GQM model (goal, question and metrics). Given this model, the authors defined the following objective: "Analyze the variations of an operational definition of TDD to evaluate with respect to TDD compliance criteria from the perspective of the developers in the context of programming activities". The authors considered the Besouro tool as a potential system for conducting quantitative TDD studies.

In the study by Fucci et al. [9], the authors presented an extensive study on TDD processes. As a goal, the authors sought to find out the impact of the effects that the TDD process characteristics can have on the external quality of the software and the productivity of the developers. The authors identified four characteristics in the TDD process, detailed in Table 1: (i) granularity, (ii) uniformity, (iii) sequence and (iv) refactoring effort.

Table 1. Characteristics corresponding to TDD processes [9].

Characteristic	Definition
(i) Granularity	Characterized by a short development process, where each cycle typically lasts between 5 and 10 min
(ii) Uniformity	Characterized by development cycles that last approximately the same time
(iii) Sequence	Indicates the prevalence of the test-first (TF) sequence during the development process
(iv) Refactoring effort	Indicates the prevalence of refactoring activity in the development process

The authors conducted a quasi-experiment in the context of the software industry. For the production of the data, the authors performed four workshops with themes on unit tests, TDD, TF, TL and iterative process of unit tests. Each workshop lasted five days. To obtain the data of the development cycles, the authors used the tool Besouro [4]. The data generated by the tool were used to calculate metrics that represent the TDD characteristics described in Table 1. Within their study, it was possible to conclude that the benefits of TDD are not only provided by the dynamics of test first (TF). TDD as a process encourages developers to follow fine and steady steps by improving the focus and flow of development [9].

Even with a number of tools designed to recognize the TDD lifecycle processes, no completely open source tools for this purpose in the community was found. There is also the need for effective analysis and summarization of data generated by the tools. Building an open-source tool free of private components is a differential against existing tools in the community, providing an increase in the maintainability and easing the evolution of the tool.

3 Butterfly Tool

To develop the **Butterfly** tool it was necessary to evolve heuristics used to classify the actions performed by developers during the development cycle. The heuristics were based on those presented by Fucci et al. [9] and Kou et al. [15].

3.1 Actions

To define each heuristic, it was necessary to classify the actions that are often executed by developers. In Table 2 each action and its respective interpretation are presented. Five essential actions were considered to produce the necessary heuristics to classify each scenario used in the software development cycle. The Test Creation action comprises creating an automated test case, either before or after writing the production code. The Test Pass action refers to the execution of one or more automated test cases successfully. The Test Failure action, contrary to the Test Pass action, is related to the execution of one or more failed automated test cases. The Test Editing action covers the inclusion, change, or removal of source code from existing automated test cases. The Code Editing action, similarly to the Test Editing action, corresponds to the inclusion, modification, and removal of production source code.

Table 2. Development lifecycle actions.

Action	Definition
Test Creation	Characterized by the creation of automatic test cases
Test Pass	Characterized by the execution of test cases that result in success
Test Failure	Characterized by running test cases that result in failure
Test Editing	Characterized by adding, changing or removing code from test cases
Code Editing	Characterized by adding, changing or removing production source code

3.2 Categories

For the development of the Butterfly tool, the heuristics underwent changes, as presented in Table 3. To define the heuristics it was necessary to evaluate in detail the life cycle of each development. The new heuristic model consists of 6 categories and 16 types of episodes. For this new model, the Production category was removed and a new category was added, called Test Driven Development. In this new category you can evaluate the entire red-green-refactoring life cycle of TDD. Like the other tools, Butterfly also considers the end of the development cycle as the Test Pass action. The tool includes the following categories: Test Addition (TA), Test-first (TF), Test-last (TL), Refactoring (RF), Test Driven Development (TDD) and Unknown (UK).

Table 3. Heuristics used to infer the classification of the development cycle.

Type	Definition
Test Addition	TA1. Test Creation → Test Pass
	TA2. Test Creation → Test Failure → Test Editing → Test Pass
Test-first	TF1. Test Creation → Code Editing → Test Pass
	TF2. Test Creation → Test Failure → Code Editing → Test Pass
	TF3. Test Creation → Code Editing → Test Failure → Code Editing → Test Pass
Test-last	TL1. Code Editing → Test Creation → Test Pass
	TL2. Code Editing → Test Creation → Test Failure → Test Editing → Test Pass
Refactoring	RF1. Code Editing → Test Pass
	RF2. Code Editing → Test Failure → Code Editing → Test Pass
	RF3. Test Editing → Test Pass
	RF4. Test Editing → Test Failure → Test Editing → Test Pass
Test Driven Development	TDD1. Test Creation → Test Failure → Code Editing → Test Pass
	TDD2. Test Creation → Test Failure → Code Editing → Test Pass → Test Editing → Test Pass
	TDD3. Test Creation → Test Failure → Code Editing → Test Pass → Test Editing → Test Failure → Test Editing → Test Pass
	TDD4. Test Creation → Test Failure → Code Editing → Test Pass → Code Editing → Test Failure → Code Editing → Test Pass
Unknown	UK1. None of the above → Test pass

Test Addition. It is understood by the addition of new test cases. In this category there is no change in production source code, only in test source code. The possible flows are seen in the Fig. 1. The first flow corresponds to a test case which was added and that did not fail when executed. The second flow considers that, after adding the test case, it failed and had to be edited until eventually passing.

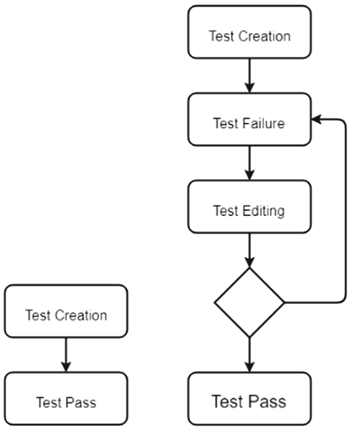

Fig. 1. Test addition flows.

Test-First. In this category, the test case must be created before the corresponding production code. The possible flows are demonstrated in Fig. 2. The first scenario considers that, after test case and code creation, the test case passed. In the second scenario, the test case is created and executed (probably due to the absence of the corresponding production code), then the production code is edited until the test case finally pass. The third flow is similar to the second, but without the execution of the test case just after its creation.

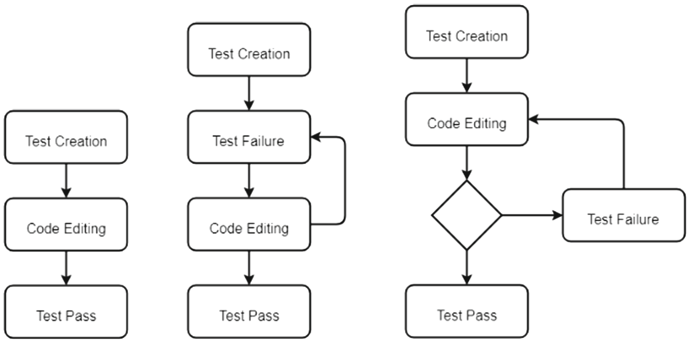

Fig. 2. Test-first flows.

Test-Last. In this category, the test case must be created after the production code is created. The possible flows are demonstrated in Fig. 3. In the first flow, production code and test code are created and the test case pass. In the second flow, the test case fails, which requires further modification of the tests until it eventually pass.

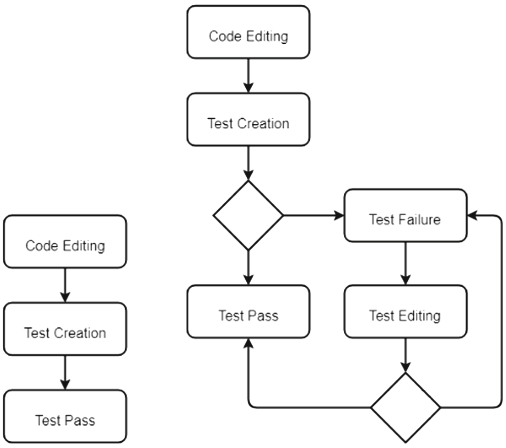

Fig. 3. Test-last flows.

Refactoring. In this category, it can be performed refactoring for production or test source code. It also comprehends activities associated with the improvement of the source code, whether it is carried out in the production code or in test cases. The possible flows are presented in Fig. 4. Two flows are associated with production code improvement, where test cases can pass after code editing or, in

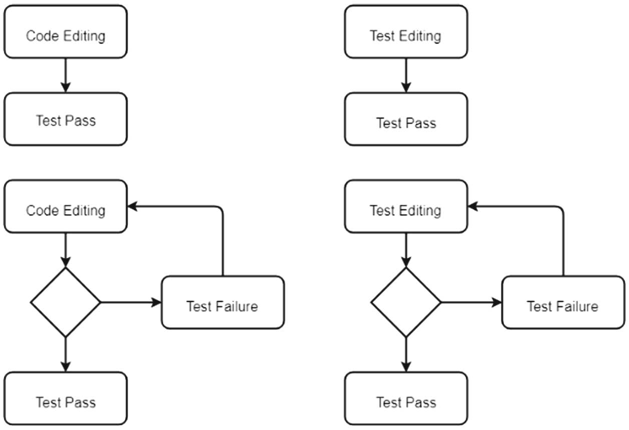

Fig. 4. Refactoring flows.

case of failure, further code editing is required. Respectively, there are two flows associated with test case improvement, where test cases are under modification.

Test Driven Development. In this category, it must be performed the test case creation before creating production source code. After the production code is created, it is necessary to perform the refactoring of the production and test source code. The possible flows are presented in Fig. 5. The first flow represents

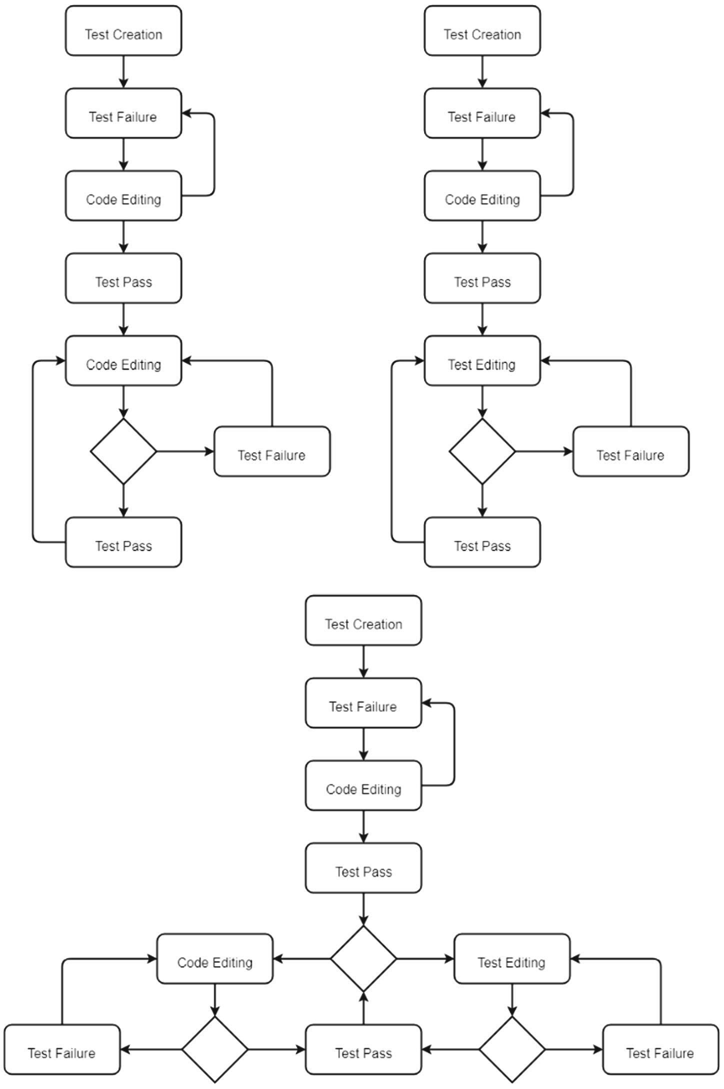

Fig. 5. Test driven development flows.

the traditional TDD cycle, in which test case is created, it fails, production code is created and modified until the test case pass, and code is improved (by refactoring), always considering the results of test cases execution. In the second flow, instead of improving the production code, the code for test cases is improved (for instance, considering a coverage criteria). Finally, the third flow is a combination of the both, improving production code and test cases.

Unknown. In order for this category to be classified by the tool, none of the known categories is achieved. In this way, everything that does not contemplate the heuristics of Table 3 will be classified as Unknown.

3.3 Environment

The tool can be used by developers who use the Eclipse Integrated Development Environment (IDE), which is compatible with the Oxygen and Photon versions. It requires Java Runtime Environment (JRE) and Java Development Kit (JDK) in version 8 or higher. The tool for running the automatic tests should be version 4 or higher of JUnit. The Butterfly tool is available on Github[1].

4 Case Study

To evaluate the tool the GQM (Goals, Questions and Metrics) paradigm was used. It consists of a mechanism to define and evolve a set of objectives using metrics [1]. According to some authors, the GQM approach is recommended for the definition of experimental studies [14,21] (Table 4).

Table 4. GQM - goals, questions and metrics.

Goals	Questions	Metrics
G1: Evaluate the compliance of the TDD process within iterative software development	Q1. Is it possible to sort the actions of the episodes in the categories of Test-addition, Test-first, Test-last, Test driven development, Refactoring, and Unknown?	M1. Number of classifications M2. Number of actions M3. Number of unit tests M4. Test coverage

[1] https://github.com/altitdb/butterfly.

4.1 Instrumentation

The development environment used by the participants has Java in version 8, IDE Eclipse Oxygen and JUnit 4. The results were generated by the Butterfly tool installed in the participants Eclipse. Each participant was responsible for sending the project data created during the experiment. The application chosen for development was the Bowling Game, which is often used for studies regarding TDD. The Bowling Game is responsible for calculating the score of a player in a bowling game according to the requirements specified in Table 5.

Table 5. Bowling game requirements.

R1	The score of the game can be consulted at any time
R2	The game consists of 10 rounds
R3	The player is entitled to two shots to reach the maximum score (10) on each round
R4	If on the first shot the maximum score is reached (strike), the player will not be entitled to the second shoot
R5	If in both shot the maximum score is reached (spare), the player will have as bonus the score obtained in the next move
R6	The strike score bonus is the value of the next two moves
R7	The player will have two extra shots if he strikes in the tenth round
R8	If he reaches the spare in the two shots after the tenth round, the player will be entitled to one extra shot

4.2 Subjects

The study was attended by professionals working in an organization focused on software development for the financial market with about 450 professionals in the area of Information Technology (IT). Among the professionals, 7 (seven) participated. Participants belong to a team of professionals who have completed advanced IT courses, such as Systems Analysis and Development, Computer Science, Information Systems, Computer Science and Software Engineering. They perform the role of Systems Analyst within the organization and are knowledgeable with Java language, Eclipse development IDE, test automation tools and TDD.

4.3 Execution

The execution was organized in three phases: (i) installation and training of the Butterfly tool, (ii) development of the Bowling Game and (iii) sending the results. In phase (i), the professionals installed the tool in the Eclipse IDE and received the training to learn how the tool should be handled. In phase (ii) the professionals developed the Bowling Game using the Java language, test tool JUnit and TDD. For each Bowling Game requirement, it is expected that a set of development activities are performed, from which the development cycles can be detected. No minimum or maximum time was set for the development of the game. In phase (iii) the professionals sent the developed game and the results to the authors by email for evaluation.

5 Results

Considering the purpose of this study, a new tool, called Butterfly, was developed. It is an evolution of Besouro tool (which, in turn, was an evolution of the Zorro tool). The main differences, as shown in Table 6, are related to compliance verification with respect to TDD and implementation dependencies.

Table 6. Comparison of automatic TDD recognition systems.

Tool	Dependencies	Compliance	User feedback	Compliance report
Zorro [15]	Hackystat, SDSA, Jess	Context-sensitive compliance	No	No
Besouro [4]	Listeners, JESS, VCS	Varying, according to the implemented component	Yes	No
Butterfly	Listeners, VCS	Standard implementation according to pre-established heuristics	Yes	Yes

The tool developed in this study aims to measure compliance and evaluate the TDD life cycle, which was limited or not possible using in other tools. Butterfly allows to check whether the developer is using TDD correctly, or if is only using TDD phases such as Test-first and Refactoring. Considering the features from Table 6, the following differences can be highlighted:

– Dependencies: In the previous tools, some dependencies were fundamental to the operation of the tool. For instance, the Jess tool was one of the main components required by them, but it was not free for commercial use. This led to its removal from the Butterfly tool, replacing its functionality by new code written by the authors.
– Compliance: Actions described in the literature were considered, and it no longer varies according to contexts or implementations as performed in the previous tools.
– User feedback: It was kept as in the Butterfly tool, given its importance in classifying actions that are not considered correct, making it possible to improve the tool in the future.
– Compliance report: A novelty in the Butterfly tool is the summarization of the actions performed by the user. With this report it is possible to analyze the percentage of use of the user's actions.

A case study was conducted to validate the use of the developed tool. In Table 7, it is presented a summary of the measurements taken from the executions for each developer during the execution of the empirical study described in Sect. 4. For metrics, the following definitions are adopted: classifications (M1) as the number of episodes classified by the tool according to the categories defined in Table 3; actions (M2) as the amount of actions carried out by each developer, as presented in Table 2; unit tests (M3) as the amount of automated test cases created by the developers; and test coverage (M4) as the percentage of production code that was covered by the unit tests with respect to control-flow criterion (statements coverage).

Table 7. Measurements regarding usage of Butterfly tool.

Developer	Classifications (M1)	Actions (M2)	Unit tests (M3)	Test coverage (M4)
D1	38	311	12	92
D2	32	223	6	**100**
D3	**63**	**692**	**21**	90.6
D4	**88**	**505**	13	**99.7**
D5	35	356	10	96.2
D6	**119**	**732**	**29**	95.2
D7	21	262	10	**100**
Average	56.57	440.14	14.42	96.24
Median	38	356	12	96.2

Metrics M1 and M2 are directly linked to the activity development effort while metrics M3 and M4 can be used to diagnose testing activities. Comparing the measurements with manual analysis of the code produced by the developers, There is no disagreement with the measurements (and classifications) made by the tool against the actions carried out by the user.

6 Discussion

Using the tool, it was possible to evaluate the TDD life cycle, evaluating the red-green-refactoring cycle as a whole and separately. The granularity, uniformity, sequence and effort of refactoring, which were presented in the Table 1, can be measured individually or in combination. Furthermore, providing facilities that can summarize the results generated by users actions facilitates the analysis of long-term empirical studies or with a large amount of user participation. Therefore, the tool enable to further evaluate development activities, and to try out new approaches within each process of the TDD life cycle, being able to evaluate the effects within a single TDD phase or for the complete TDD iteration. For instance, the inclusion of the use of test criteria in the TDD cycle is being evaluated, and this study provides a tool that will help evaluate this inclusion, so that gains are obtained and threats removed or mitigated.

Making the tool open-source gives researchers the possibility of new implementations without the need to learn new frameworks. In order for the extensibility of the tool to be carried out, only knowledge in the Java language is necessary.

7 Conclusion

In this study, it was presented the Butterfly tool, which is a tool built with the purpose of analyzing the process of developing iterative software by classifying it into categories. Each category has a set of heuristics, which are responsible for determining in which category the user's actions meet. The categories offered by the tool are: Test Addition, Test-first, Test-last, Test Driven Development, Refactoring, and Unknown.

To perform the tool evaluation the GQM paradigm was used, allowing to elaborate objectives, questions and metrics for the purpose of evaluating experimental studies. The Butterfly tool serves the purpose of this study, which is to evaluate the compliance of the TDD process in the development of iterative software. With the established metrics it is possible to verify that the actions and episodes generated were categorized in an expected way. There were no divergences between developers and the categorization performed by the tool.

Finally, a tool that can support empirical studies about TDD was developed, easing the analysis of the information generated. The Butterfly tool was provided on Github[2], so that the source code can be used by researchers or by the community at large. The tool is also open to further improvements, and the community can contribute using the open source format.

References

1. Basili, V.R.: Software modeling and measurement: the goal/question/metric paradigm. Technical report CS-TR-2956, UMIACS-TR-92-96, p. 24. University of Maryland, College Park, MD, USA, September 1992. http://www.cs.umd.edu/~basili/publications/technical/T78.pdf
2. Beck, K.: eXtreme Programming Explained: Embrace Change. Addison-Wesley, Boston (1999)
3. Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Boston (2002)
4. Becker, K., de Souza Costa Pedroso, B., Pimenta, M.S., Jacobi, R.P.: Besouro: a framework for exploring compliance rules in automatic TDD behavior assessment. Inf. Softw. Technol. **57**, 494–508 (2015)
5. Briand, L., Bianculli, D., Nejati, S., Pastore, F., Sabetzadeh, M.: The case for context-driven software engineering research: generalizability is overrated. IEEE Softw. **34**(5), 72–75 (2017)

[2] https://github.com/altitdb/butterfly.

6. Causevic, A., Punnekkat, S., Sundmark, D.: TDD^{HQ}: achieving higher quality testing in test driven development. In: Euromicro Conference Series on Software Engineering and Advanced Applications, pp. 33–36, Santander, Spain (2013)
7. Causevic, A., Sundmark, D., Punnekkat, S.: Factors limiting industrial adoption of test driven development: a systematic review. In: International Conference on Software Testing, Verification and Validation, pp. 337–346. IEEE, Berlin, Germany (2011)
8. Causevic, A., Shukla, R., Punnekkat, S., Sundmark, D.: Effects of negative testing on TDD: an industrial experiment. In: Baumeister, H., Weber, B. (eds.) XP 2013. LNBIP, vol. 149, pp. 91–105. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38314-4_7
9. Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., Juristo, N.: A dissection of the test-driven development process: does it really matter to test-first or to test-last? Trans. Softw. Eng. **43**(7), 597–614 (2017)
10. Garousi, V., Felderer, M., Kuhrmann, M., Herkiloglu, K.: What industry wants from academia in software testing?: Hearing practitioners' opinions. In: International Conference on Evaluation and Assessment in Software Engineering, pp. 65–69. ACM, Karlskrona, Sweden (2017)
11. George, B., Williams, L.: A structured experiment of test-driven development. Inf. Softw. Technol. **46**(5), 337–342 (2004)
12. Hammond, S., Umphress, D.: Test driven development: the state of the practice. In: Smith, R.K., Vrbsky, S.V. (eds.) ACM Annual Southeast Regional Conference, pp. 158–163. ACM, Tuscaloosa, Alabama, USA (2012)
13. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer **38**(9), 43–50 (2005)
14. Juristo, N., Moreno, A.M.: Basics of Software Engineering Experimentation. Kluwer Academic Publishers, Dordrecht (2001)
15. Kou, H., Johnson, P.M., Erdogmus, H.: Operational definition and automated inference of test-driven development with Zorro. Ann. Softw. Eng. **17**(1), 57–85 (2010)
16. Offutt, J.: Why don't we publish more TDD research papers? Softw. Test. Verif. Reliab. **28**(4), e1670 (2018)
17. Pachulski Camara, B.H., Graciotto Silva, M.A.: A strategy to combine test-driven development and test criteria to improve learning of programming skills. In: Technical Symposium on Computing Science Education, pp. 443–448. ACM, Memphis, TN, USA (2016)
18. Raulamo-Jurvanen, P., Mäntylä, M., Garousi, V.: Choosing the right test automation tool: a grey literature review of practitioner sources. In: International Conference on Evaluation and Assessment in Software Engineering, pp. 21–30. ACM, Karlskrona, Sweden (2017)
19. Shelton, W., Li, N., Ammann, P., Offutt, J.: Adding criteria-based tests to test driven development. In: International Conference on Software Testing, Verification and Validation, pp. 878–886. IEEE, Montreal, QC, Canada (2012)
20. Spinellis, D.: State-of-the-art software testing. IEEE Softw. **34**(5), 4–6 (2017)
21. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Sweden (2000)