UWS Academic Portal

Comparing technical debt in student exercises using test driven development, test last and ad hoc programming

Parodi, Eugenia; Matalonga, Santiago; Macchi, Dario; Solari, Martín

Published in:
XLII Latin American Computing Conference (CLEI), 2016

DOI:
10.1109/CLEI.2016.7833380

Published: 26/01/2017

Document Version
Peer reviewed version

Link to publication on the UWS Academic Portal

Citation for published version (APA):
Parodi, E., Matalonga, S., Macchi, D., & Solari, M. (2017). Comparing technical debt in student exercises using test driven development, test last and ad hoc programming. In *XLII Latin American Computing Conference (CLEI), 2016* (pp. 1-10). IEEE. https://doi.org/10.1109/CLEI.2016.7833380

General rights
Copyright and moral rights for the publications made accessible in the UWS Academic Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact pure@uws.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 21 Jan 2022

Comparing Technical Debt in Student Exercises Using Test Driven Development , Test Last, and Ad Hoc Programming

Eugenia Parodi
PEDECIBA Informática
Montevideo, Uruguay
eugeniaparodi@gmail.com

Santiago Matalonga, Darío Macchi, Martín Solari
Facultad de Ingeniería
Universidad ORT Uruguay
Montevideo, Uruguay
{smatalonga,macchi}@uni.ort.edu.uy,
martin.solari@ort.edu.uy

Abstract—**Technical Debt is a metaphor that explains a phenomenon that occurs in software development when programmers are faced with trade-off decisions (usually ship first vs. quality assurance). This work analyses the amount of technical debt incurred by undergraduate students using different coding techniques. This observational study uses source code from seventy-five students. We provided students with similar exercises to compare techniques by measuring with static code analyzers (Sonar, FindBugs). The techniques are TDD, Test Last, and ad hoc programming. Our results could not find a statistically significant difference of technical debt incurred by each development practices. Nonetheless, with both tools ad-hoc programming measures less technical debt than TDD, and TDD measures less than Test Last. Furthermore, we observed that the two measurement tools outputs are not statistically correlated. Finally, we discuss implications for the research of technical debt stemming from our observations.**

Keywords—Technical debt; TDD; Test First; Lest Last;

I. INTRODUCTION

Technical debt is a metaphor introduced by Cunningham (1992) that explains how "shipping first" incurs debt. For instance, in the case of shipping new buggy functionality. Developers can either push the release date forward or ship the new functionality without extensive testing. We say that they are incurring in Technical debt if the developers take the later decision [1]. Technical Debt is a topic that has been getting considerable attention in the field of software engineering over the last years[2]. Technical Debt has become relevant to the software engineering community due its ability to explain the hidden costs of poor quality [3]. For instance, the Gartner Group estimates that the global Technical Debt in the year 2010 was around 500 million dollars and could be doubled in five years [4].

This paper presents a comparison of Technical Debt involving undergraduate students from Universidad ORT Uruguay. These students were enrolled in courses in software engineering in the engineering school and the technical school. A total of 75 subjects participated in the observation, using three different coding technique: Test Driven Development, Test Last, and ad-hoc programming.

Test Driven Development (acronym TDD) is a software development practice in which automated test cases are written before the functionality development [5]. Test Last, encompass the subject who performed extensive unit testing, but who did not necessary applied the TDD principles. Finally, ad-hoc programming represents those subjects who did not follow a specific coding approach.

Based on the principles of each technique, our hypothesis was that the software developed applying TDD would have less Technical Debt than those developed with Test Last and ad hoc programming. For the purposes of the research, TD was measured using two static code analysis tools (Findbugs and Sonar). Measuring code technical debt with static code analyzer tools is a standard practice in TD research [3].

The results could not find statistically significant differences in the amount of Technical Debt incurred by each technique. A retrospective analysis of the results and the experimental design hits at several confounding factors that could explain our results. First of all, training given to students was not enough to support higher abstraction techniques like TDD. The adoption of TDD often requires a cultural change of each developer, as it presents a different way to write and design software [6]. Secondly, the coding tasks given to the students were similar (ACM programming exercises) they were not equivalent. As a result, this empirical observation is not a controlled experiment, and variations in the exercise can account for the variations in the results.

On the other hand, our results also found that the measures of technical debt used in this study do not exhibit a monotonic relationship. The selected tools for measuring technical debt are those commonly used in the literature. To the best of our knowledge, there is no formal study comparing the output of these tools. Though both Sonar and Findbugs are different measures of Technical Debt, they both measure the same attribute. The absence of a monotonic relationship could have implications for the way code technical debt is currently

understood. Nonetheless, further experimental research is needed to confirm or deny this observation.

The rest of this article is structured as follows. Section II presents the study objectives and setting. Section III presents the data analysis. Section IV presents our answers to the research questions, including a discussion of the confounding factors that could have had an influence on the results. Threats to validity are discussed in Section V. A review of the current literature on measuring Technical Debt, and of the techniques employed in this empirical observation is presented in section VI. Finally, in Section VII concludes this article with a reflection about the results obtained and future research directions.

II. Study Objectives and Setting

This section describes the study objectives and settings.

Objective: Evaluate how different techniques introduce Technical Debt by looking at the source code.

Work hypothesis: Software developed using TDD will incur in less Technical Debt than those developed with Test last or ad-hoc techniques.

A. Description of the participants

All the selected participants were undergraduate students from Universidad ORT Uruguay. Based on each background, we identified the following groups:

The first two groups included students enrolled in technical degrees. They were trained in unit test using JUnit and in Test Driven Development. These students had four courses on software development through degree syllabus. Furthermore, for the purpose of this experience, they received a 12-hour workshop (spread over four weeks). The group is split into those who decided to apply TDD and those who decided to use unit testing but did not apply TDD – which we labeled Test Last. We will call each of these techniques TDD and TL respectively.

The third group included students who were training for the annual ACM Programming Concourse at the same university (referenced with the name ACM). They are referenced as ad-hoc programming since no methodology is imposed on them. Students who enroll in the programming training come from the first and second semester with one or two programming courses done until that moment.

B. Description of the programming exercises utilized

All subjects participating in this study were asked to solve exercises taken from acmsolver.com. Thereby giving reasonable confidence that students were faced with similar challenges, though it introduced variation in the observation (see section V.C).

These exercises were translated into Spanish by the researchers. It is worth to note that for the TDD and TL groups, the exercise is being used to grade the students. Student grading can be a threat to the experimental design, as it has been documented in [7].

C. Description of the treatment

TDD and TL first students received the same training. A 4-week workshop was designed to introduce students to automatic unit testing with JUnit, refactoring, and TDD. Students were expected to code training exercises in each of the weeks to develop their skills. For the final exercise, which was graded by the instructors, students were allowed to choose if they wanted to try TDD and those who had used TL.

On the other hand, ad hoc participated in a workshop designed to train them for the regional ACM programming competition. Unit testing with JUnit is not part of the syllabus of this workshop.

D. Data Collection Procedure

The instructors collected the source code from the TDD and TL groups. These instructors provided the classifications of subjects into TDD and TL before handing the source code to the authors of this articles. Each subject was assigned a sequential identifier.

Source code from ad hoc programming was downloaded directly from the training platform for the ACM competition from Universidad ORT Uruguay. From the available pool of training students, one source code exercise was selected and downloaded.

We set up Findbugs Eclipse plug-in and loaded each of the projects to obtain the Technical Debt measure. Likewise, we installed an instance of Sonar Server, and each of the source code was examined to obtain the Technical Debt measure (the resulting data set is presented in Appendix –Data Set).

E. Hypothesis and variables

As we have mentioned, our aim with this observation was to confirm that source code built using TDD had more quality than source code built using other techniques. The proxy for quality in this observation is Technical Debt measured by two different tools: Findbugs and Sonar. Therefore, our null hypothesis is:

H_0: Subjects using TDD have the same code quality than those using TL or ad hoc programming (measured as the total number of defects identified by Findbugs / Sonar).

The variables under analysis are the number of defects identified by each tool. Though both tools classify defects by severity, for the purpose of this observation, only the total number of identified defects is considered.

III. Data Analysis

Overall, 75 subjects were observed in this experience. Raw data from the observation is presented in Appendix –Data Set.

A. Descriptive statistics

The following table presents the summary of the obtained data, and also presented as a graph in Figure 1 and Figure 2.

	Number of subjects	Find bugs		Sonar	
		Median	Std Dev	Median	Std Dev
Ad_hoc	32	2	1,56	14	8,0
Test Last	32	3	2,44	85	103
TDD	11	3	2,23	57	98,84

Table 1: Descriptive statistics of collected data

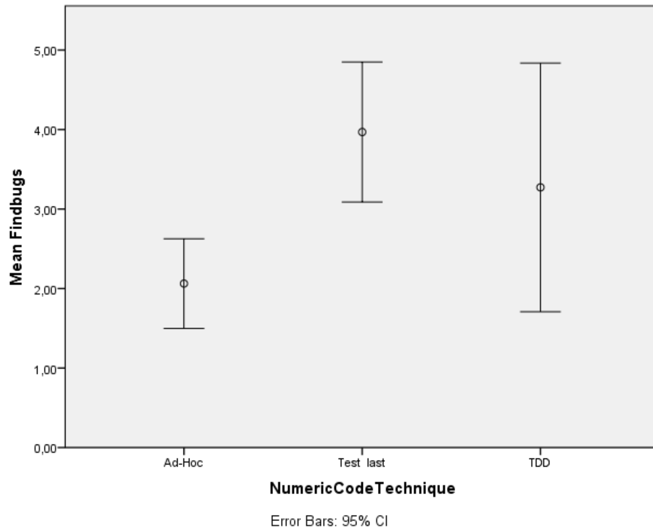


Figure 1: Descriptive statistics graph for Findbugs

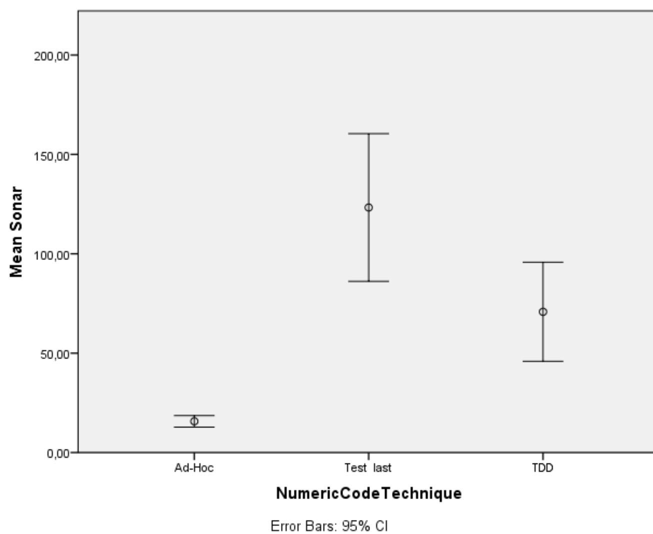


Figure 2: Descriptive statistics graph for Sonar

B. Analysis of hypothesis

In order to evaluate our work we have decomposed our alternative hypotheses into:

H_1f: There is a difference among the observed groups regarding Technical Debt as measured by Findbugs.

H_1s: There is a difference among the observed groups regarding Technical Debt as measured by Sonar.

In order to evaluate this two alternative hypothesis, we applied the Kruskall-Wallis H test. Table 2 presents the evaluation of the general assumptions of the selected statistical method.

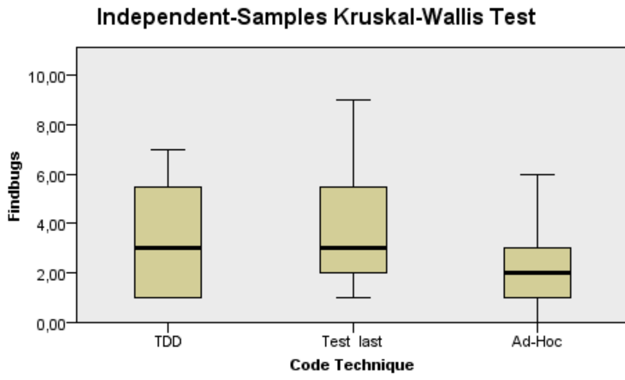
Assumption	Evaluation
The dependent variable is ordinal	Yes, see section Hypothesis and variables
The independent variable consists of two or more categorical groups	Yes, these are represented by the coding technique (TDD, TL, ad hoc).
There is independence of observation	Assumed true. As part of the University policies, students' exercises are checked for plagiarism.

Table 2: General Assumptions for the Kruskall-Wallis H Test

The remainder assumptions of the Kruskall-Wallis H Test are dependent on the data and are reviewed in each of the following subsections.

1) Evaluation of H_1f

A Kruskall-Wallis H Test was run to determine if there were differences in the accrued technical debt as measured by Findbugs for TDD, TL, and ad hoc programming. The distributions for the accrued Technical Debt were not similar for all groups as visually assessed through box plots (see Figure 3). The distributions were statistically significant between groups with X2(2) = 10,991, p=0,004.



Total N	75
Test Statistic	10,991
Degrees of Freedom	2
Asymptotic Sig. (2-sided test)	,004

1. The test statistic is adjusted for ties.

Figure 3: Results of Findbugs Kruskall-Wallis H Test

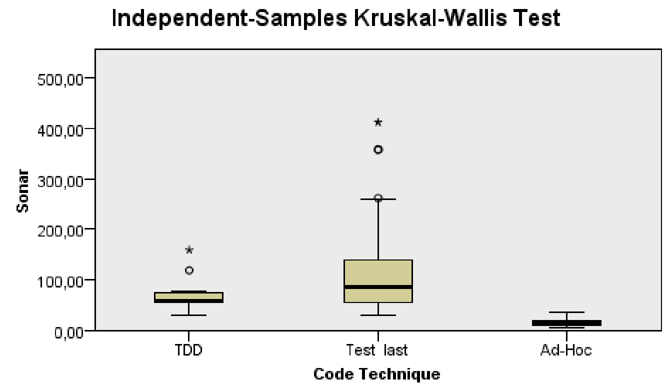
Despite not having statistical significance, an observation of the ranks hints that TL has more technical debt (47) than TDD (40), and finally ad hoc with 29. All values declared are mean ranks see Table 4.

Group	N	Mean Rank
Ad hoc	32	29
TDD	12	39
TL	32	47

Table 3: Mean rank for Findbugs by group

1) *Evaluation of H_1s*

A Kruskall-Wallis H Test was run to determine if there were differences in the accrued technical debt as measured by Sonar for TDD, TL, and Ad-hoc. The distributions for the accrued technical debt were not similar for all groups as visually assessed through box plots (see Figure 4). The distributions were statistically significant between groups with X2(2) = 53,403, p=0,000.



Total N	75
Test Statistic	53,403
Degrees of Freedom	2
Asymptotic Sig. (2-sided test)	,000

1. The test statistic is adjusted for ties.

Figure 4: Results of Sonar Kruskal-Wallis Test

In spite of not having statistical significance, an observation of the ranks hints that TL has more technical debt (47) than TDD (40), and finally ad hoc with 29. All values declared are mean ranks see Table 4.

Group	N	Mean Rank
Ad hoc	32	17
TDD	12	49
TL	32	56

Table 4: Mean rank for Sonar by group

C. *Other analysis*

In addition to the research objectives, there is the possibility to analyze if there is a relationship between the measures of technical debt provided by two different tools. To evaluate this relationship is important because both code analysis tools are used to measure technical debt in the literature (see Section VI.A). Therefore, it is expected that there is a monotonic relationship between them.

A Spearman’s rho was applied to test for correlation among the variables. The results could not confirm a significant correlation between them (rho = 0,223). Figure 4 presents a scatter plot to observe that this linear correlation is not present in the data. The implication for technical debt research of this result is presented in the section IV.D).

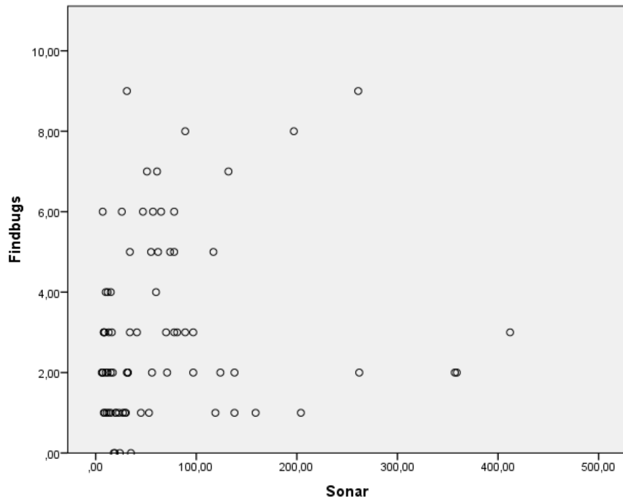


Figure 5: Sonar x Findbugs scatterplot

IV. ANSWER TO RESEARCH QUESTIONS AND INTERPRETATION OF RESULTS AND

Regarding the main objective of this research, to confirm that code produced using TDD had less technical debt than code produced using other techniques. This research could not support this hypothesis. In spite of the amount of participants involved in the observation (75 undergraduate students), the analysis did not yield statistically significant results.

Nevertheless, it is interesting to note that ordering the techniques by the number of issues reported by both tools shields the same order (see Table 3 and Table 4) for the techniques. This is, ad-hoc produces fewer issues than TDD, which produces fewer issues this ad-hoc produces than Test last. This would hint at a relationship between the three techniques. Even more so, since the measures appear unrelated.

The reported literature on quality improvement using TDD has mixed results (see Section Results from other comparisons involving TDD). Nonetheless, and evaluation of our research design shows that there are several confounding factors that must be taken into account when observing our results.

A. *Cognitive complexity of TDD*

Arguably, TDD is the most cognitively demanding the coding practice applied by the subjects of this empirical observation. Evidence of this is the comparatively low number of subjects that applied the technique (12 out of 75). Both TDD and TL subject received equal training, and yet only 12 out of 44 decided to apply the technique in the final exercise of the course.

No operationalization of the technique was required for the subjects. This lack of operationalization is the likely source of the spread of the results observed in the range of the issues detected by the measurement tools. Further research, must introduce an operationalization of the compared coding techniques in order to reduce variations, and increase confidence that the subjects are correctly applying the assigned technique.

B. *Error in the assignments of subjects groups (TDD and TL)*

As it was mentioned in section II.A, TDD and TL came from students taking the same course in a technical degree setting. Division into groups was based on the instructor judgment on the technique applied. The instructors judgment liable to selection bias and may be a source of the variation seen in the results. It is also a possible explanation for the lack of statistically significant results.

C. *Effect of code size on Technical Debt*

Another aspect that was not taken into account during data collection was the length of code involved in each group. It is likely that code also has an impact on the amount if issues identified by these tools. As code that is more sizable is likely to have more issues. We revisited the source code to add this data to the dataset; the summary is presented in Table 5 and Figure 6.

Code Technique	Number of Cases	Median (LOC)	Standard Deviation
Ad hoc	32	58,5	178,8
Test First	32	1153,5	904,4
TDD	11	291	25,8

Table 5: Summary of descriptive statistics for LOC of subjects' exercises

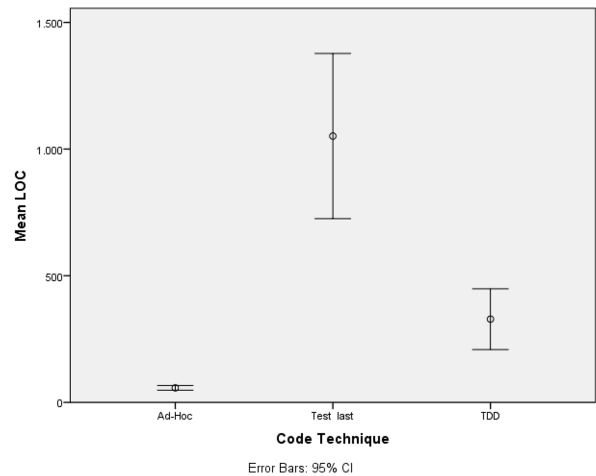


Figure 6: Descriptive statistics for LOC

Figure 6 show that ad hoc exercises have fewer LOC than TDD exercises. Moreover, that TDD exercises have less LOC than test first exercises. Which shields the same order than the observation of accrued technical debt (see section IV). Our experimental design prevents us from further analyzing this observation.

D. *Immaturity of Technical Debt measures*

Technical debt is a metaphor which has been successfully applied to understand events that occur during software

development. For that matter when working with code technical debt, the de-facto standard is to use the static checking tools as a measure of technical debt. Nonetheless, our observation showed that there is not relationship – or at least not linear relationship – between the two measures of technical debt.

In theory, they are both measuring the same attribute from the same entity. Both are tools for measuring Technical debt (attribute) from the entity source code. Therefore, a monotonic relation is to be expected of both measures. If an entity has more technical debt, this should be true regardless of the measurement scale. Nevertheless, our results could not confirm the existence of a monotonic relationship between the two measures (see Figure 4). This observation opens new research questions towards the validity of using these measures as a proxy for technical debt.

V. Threats to validity

This section discusses a set of threats to the validity [8] of the results discussed in this study.

A. *Statistical significance*

The number of subjects per group was not balanced. However, care was taken in the application of the statistical tests (we abide by variable type restrictions and the assumptions of each of the applied statistical methods). Our results failed to show statistical significance for the test applied. Therefore, all conclusions are based on the researcher's judgment – which is prone to a threat in generalization.

B. *Different goals between each group*

An aspect that can challenge the validity of the observation (construct validity) is that students involved in ad-hoc programming groups had different motivation that students involved in TDD or TL groups.

On one hand, the students which we label as ACM were willingly participating in training seminars for the ACM programming contest. They are not graded for their exercises, and their main goal is to provide source code that fulfills the functional specification in the shortest time possible.

On the other hand, students which we labeled as TDD or TL, were participating in a curricular course of their degree and were graded for the assignment of a source code programming exercise.

C. *Experimental design*

Throughout this paper, we have labeled this work as an empirical observation of the behavior of technical debt in undergraduate projects. The researchers had no control over the instrumentation or the assignments of subjects to groups. To make the results more definite, a different type of experimental design must be made to block several of the confounding factors that have been discussed in this paper.

VI. Related work

A. *Measuring technical debt*

Technical debt research has become a stronghold for the development of empirical research in software engineering. This observation was consolidated in [2], where a summary of empirical results in technical debt research is presented. As a result, a driver of technical debt research has been the need to reliable measurements. According to the reviewed literature, most of the authors discussed methods to determine technical debt by analyzing the source code [3]. These approaches were based on the cohesion and coupling measurement among others code indicators of each component. To the best of our knowledge, Findbugs and Sonar are the most used ones in the literature.

Other measures of technical debt have been proposed that do not rely on source code analysis. For instance in [9], Nord et al. describe an empirical model based on the cost of implementation and rework of alternative and conflicting architectural solutions.

, in an ongoing system development project, by determining the propagation metric [10], no methodology was defined in order to manage and monitor the debt across the software development.

Finally, Guo et al. [11] presented a general methodology, to deal with technical debt during the development life cycle. This framework defines three main activities that were executed during the whole SLDC phase, in order to create an incremental technical debt steps: Identification, measurement and monitoring. It also suggests four elements as sources of technical debt:

- Design debt: defined as any anomaly in the source code and/or documentation that could decrease the maintainability of the system.
- Defect debt: known defects not yet fixed.
- Testing debt: identified as planned tests that were not executed.
- Documentation debt: when the product's documentation is not up to date.

Though the evidence of the application of these model relies on source code measures for technical debt.

B. *Overview of Development techniques*

Test Last programming, also known as Test-Last Development or TLD is a popular testing model, with developers focused on programming functionalities before testing them. It popularity can be related to waterfall software development model popularity where it was the only testing model that made sense (testing phase is the last one in the waterfall).

Test Driven Development (acronym TDD) is a software development practice in which automated test cases are written before the functionality development [5]. It is considered part of the eXtreme Programming methodology [12] and a design flow paradigm with test cases acting as a starting point and central elements throughout the whole design process [13]. In TDD, test cases are specified first (challenging traditional way [14][15][16]), and, since no implementation is available, they will initially fail. Based on error messages from failing test cases, the implementation grows incrementally until all test cases pass eventually [17].

TDD technique not only encourages programmers to write code that can be easily tested but also requires an open mind and discipline. However, the test part of TDD name has lead to many misunderstandings (mostly in newcomers) due the belief that only implies testing and not analysis nor design. TDD development has a sharp learning curve. It requires advanced skills in order to be correctly implemented, because of how the paradigm forces developers to think and design test cases before implementing an unfamiliar functionality [18].

C. Results from other comparisons involving TDD

In the last ten years, there have been several scientific studies on TDD. The very large quantity and maturity of results have made possible the realization of secondary studies (systematic reviews) [19][5][20]; which are a good way to summarize and interpret quantitative results of a collection of individual studies over a specific topic.

Empirical studies on industrial and academic projects report improvements in quality, in some cases significant and in others not so much [5]. On external quality (perceived by the user as the satisfaction of use, flexibility and efficiency attributes) there are reports of simple improvements [21], some noticeable improvements or directly no improvement [19]. On the internal quality (how the code is developed, ease of correction, adaptation, and expansion), it is ensured that it increases [21]. On productivity, there are reports that range from indeterminate results [19] [21], that it worsens [22] and even that TDD improves productivity [23]. Regarding the impact on the development effort, the reports go from indeterminate results to improvements thanks to the use of TDD [5]. Finally, a survey of process improvement practices puts TDD as the second practice with the greatest impact on the success of a project (only after code inspection as the first one) [24]. All this lack of conclusive results and nuances about its uses are issues that still occupy many researchers.

Like other empirical experiments, the validity of our results are based on a large set of subjects that were not randomly selected; they were aware that TDD and Testing-Last must be applied. We hope that our focus on the comparison between both programming paradigms will assist others in order to get a better view on how the adopted technology incurs TD.

VII. CONCLUSIONS

Technical Debt is a metaphor for explaining events that happen during the development of software. TDD has proven a viable way to produce software, the hype around it signals that better quality and better readable software can be obtained when using these techniques [6]. Nonetheless, empirical studies of code quality resulting from applying TDD is ambiguous on the effects of the technique. This study provides another source of contrast since the expected quality improvement could not be observed from the data under study.

This manuscript has presented an empirical evaluation of TDD, TL and ad hoc programming techniques. We reviewed source code from programming assignments from 75 different students. The goal of this work was to evaluate if code produced with TDD had fewer Technical Debt than code produced using TL and ad ho programming techniques.

Technical Debt was measured using static code analysis tools (Sonar and Findbugs), which are standard tools for measuring Technical Debt in the community.

Our results could not support the working hypothesis, as no statistically significant difference was found in the observed source code. Nonetheless, several interesting affirmation can be made:

- In spite of not achieving statistical significance, the order in which the observed techniques produced Technical Debt remained the same in both measurement tools. The order is, from lowest to greatest: ad hoc → TDD → test last.
- There is no significant linear relationship between the two measurement tools.

As we have discussed, there are other confounding variables which our empirical study design cannot block in order to generalize the results. Specifically, training and motivation of the subjects in the evaluated techniques is a big source of variation in our study. In addition to this, our results show that code size is also a source of variation and it's likely to have more effect on Technical Debt than coding technique.

Future work needs to involve the study design to evaluate the research lines opened by this empirical observation. Static code analysis tools must be thoroughly studied in order to evaluate if they represent meaningful proxies for Technical Debt. In this study, we have only counted the total number of identified defects. Both tools classify the defects by severity. The analysis in this paper should be extended to accommodate for the severity of the identified defects.

Furthermore, the observation of students needs to evolve to a controlled experiment design reducing the variation of confounding factors to better observe each technique.

REFERENCES

[1] W. Cunningham, "Technical Debt," *Cut. It J.*, vol. 23, pp. 1–44, 2010.

[2] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, "Technical Debt: Showing the Way for Better Transfer of Empirical Results," in *Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach*, J. Münch and K. Schmid, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 179–190.

[3] A. Villar and S. Matalonga, “Definiciones y tendencia de deuda técnica: Un mapeo sistemático de la literatura,” in *Memorias de la XVI Conferencia Iberoamericana de Ingeniería de Software CIbSE 2013*, 2013, pp. 33–46.
[4] Gartner, “Gartner Estimates Global ‘IT Debt’ to Be $500 Billion This Year, with Potential to Grow to $1 Trillion by 2015.” .
[5] R. Jeffries and G. Melnik, “Guest Editors’ Introduction: TDD--The Art of Fearless Programming,” *IEEE Softw.*, vol. 24, no. 3, 2007.
[6] D. Janzen and H. Saiedian, “Test-driven learning in early programming courses,” *ACM SIGCSE Bulletin*, vol. 40. p. 532, 2008.
[7] N. Juristo and A. M. Moreno, “Basics of Software Engineering Experimentation,” *Analysis*, vol. 5/6, p. 420, 2001.
[8] C. Wohlin, M. Höst, P. Runeson, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, Massachusetts: Kluwer Academic Publishers, 2000.
[9] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In Search of a Metric for Managing Architectural Technical Debt,” *2012 Jt. Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, pp. 91–100, Aug. 2012.
[10] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *IEEE International Conference on Software Maintenance, ICSM*, 2004, pp. 350–359.
[11] Y. Guo, R. O. Spínola, and C. Seaman, “Exploring the costs of technical debt management – a case study,” *Empir. Softw. Eng.*, vol. 21, no. 1, pp. 159–182, Feb. 2016.
[12] K. Beck, *Extreme Programming Explained: Embrace Change*, no. c. Addison-Wesley Professional, 1999.
[13] M. Diepenbeck, M. Soeken, D. Grose, and R. Drechsler, “Behavior Driven Development for circuit design and verification,” in *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2012, pp. 9–16.
[14] K. Beck, *Test Driven Development: By Example*. 2002.
[15] H. Erdogmus, M. Morisio, and M. Torchiano, “On the effectiveness of the test-first approach to programming,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 226–237, 2005.
[16] D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction,” *Computer (Long. Beach. Calif).*, vol. 38, no. 9, pp. 43–50, 2005.
[17] D. Sundmark and S. Punnekkat, “Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment,” pp. 138–152, 2012.
[18] J. Buchan, L. Li, and S. G. MacDonell, “Causal factors, benefits and challenges of test-driven development: Practitioner perceptions,” in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2011, pp. 405–413.
[19] Y. Rafique and V. B. Mĭsí, “The effects of test-driven development on external quality and productivity: A meta-analysis,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 835–856, 2013.
[20] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, “What do we know about test-driven development?,” *IEEE Softw.*, vol. 27, no. 6, pp. 16–19, 2010.
[21] C. Desai, D. Janzen, and K. Savage, “A survey of evidence for test-driven development in academia,” *ACM SIGCSE Bull.*, vol. 40, no. 2, p. 97, 2008.
[22] T. Burak, L. Layman, M. Diep, H. Erdogmus, and F. Shull, “How Effective Is Test-Driven Development?,” in *Making Software: What Really Works, and Why We Believe It*, no. 12, 2010, pp. 207–220.
[23] a. Gupta and P. Jalote, “An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development,” *First Int. Symp. Empir. Softw. Eng. Meas. (ESEM 2007)*, no. Ccd, pp. 285–294, 2007.
[24] K. El Emam, “The ROI from Software Quality,” *Cut. Consort. Rep.*, vol. 5., p. 20, 2004.

VIII. Appendix –Data Set

Subject	**Technical Debt**	
	#FindBugs	#Sonar
TDD_Subject 3	1	53
TDD_Subject 4	5	55
TDD_Subject 5	1	159
TDD_Subject 9	7	61
TDD_Subject 11	6	78
TDD_Subject 19	1	30
TDD_Subject 20	1	119
TDD_Subject 21	2	56
TDD_Subject 43	3	41
TDD_Subject 44	6	57
TDD_Subject 45	3	70
TDD_Subject 46	3	523

Table 1: TDD

Subject	**Technical Debt**	
	# FindBugs	#Sonar
TL_Subject 1	5	34
TL_Subject 2	2	32
TL_Subject 6	1	138
TL_Subject 7	2	31
TL_Subject 8	2	32
TL_Subject 18	9	31
TL_Subject 22	3	89
TL_Subject 23	2	138
TL_Subject 24	2	124

TL_Subject 25	8	197
TL_Subject 26	3	412
TL_Subject 27	9	261
TL_Subject 28	3	97
TL_Subject 29	2	359
TL_Subject 30	2	262
TL_Subject 31	2	97
TL_Subject 32	2	357
TL_Subject 33	5	62
TL_Subject 34	4	60
TL_Subject 35	2	71
TL_Subject 36	3	81
TL_Subject 37	7	132
TL_Subject 38	5	78
TL_Subject 39	5	117
TL_Subject 40	1	204
TL_Subject 41	1	45
TL_Subject 42	7	51
TL_Subject 51	6	47
TL_Subject 52	3	78
TL_Subject 53	8	89
TL_Subject 54	5	74
TL_Subject 55	6	65

Table 2:TL

Subject	**Technical Debt**	
	# FindBugs	#Sonar
ACM_Subject 1	0	19
ACM_Subject 2	0	18
ACM_Subject 3	1	20
ACM_Subject 5	1	27
ACM_Subject 6	2	17
ACM_Subject 7	0	35
ACM_Subject 10	2	7
ACM_Subject 11	1	29
ACM_Subject 12	3	13
ACM_Subject 13	3	16
ACM_Subject 14	3	34
ACM_Subject 15	4	10
ACM_Subject 16	4	15
ACM_Subject 17	1	11
ACM_Subject 18	3	8
ACM_Subject 19	1	13
ACM_Subject 21	4	12
ACM_Subject 22	3	9
ACM_Subject 24	6	26
ACM_Subject 25	2	7
ACM_Subject 26	1	9
ACM_Subject 27	2	12
ACM_Subject 28	1	20
ACM_Subject 29	2	15
ACM_Subject 30	1	8
ACM_Subject 31	3	8
ACM_Subject 32	6	7
ACM_Subject 33	2	10
ACM_Subject 34	0	24
ACM_Subject 35	1	23
ACM_Subject 36	1	15
ACM_Subject 37	2	6

Table 3: ACM

.