*This is a preprint of an article:* Lech Madeyski and Łukasz Szała, "The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study", in Software Process Improvement, ser. Lecture Notes in Computer Science, P. Abrahamsson, N. Baddoo, T. Margaria, R. Messnarz, Eds., LNCS 4764. Springer, 2007, pp.200-211
http://dx.doi.org/10.1007/978-3-540-75381-0_18

# The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study

Lech Madeyski, Łukasz Szała

Institute of Applied Informatics, Wrocław University of Technology,
Wyb.Wyspiańskiego 27, 50370 Wrocław, POLAND
Lech.Madeyski@pwr.wroc.pl, Lukasz.Szala@e-informatyka.pl

**Abstract.** Test-driven development (TDD) is entering the mainstream of software development. We examined the software development process for the purpose of evaluation of the TDD impact, with respect to software development productivity, in the context of a web based system development. The design of the study is based on Goal-Question-Metric approach, and may be easily replicated in different industrial contexts where the number of subjects involved in the study is limited. The study reveals that TDD may have positive impact on software development productivity. Moreover, TDD is characterized by the higher ratio of active development time (described as typing and producing code) in total development time than test-last development approach.

## 1 Introduction

Experimentation in software engineering is a relatively young field. Nevertheless, relevance of experimentation to software engineering practitioners is growing because empirical results can help practitioners make better decisions and improve their products and processes. Beck suggests treating each software development practice as an experiment in improving effectiveness, productivity etc. [1]. Productivity is usually defined as output divided by the effort required to produce that output [2]. An interesting survey of productivity measures is also presented by Fowler [3]. Test-driven development (TDD) practice [4], also called test-first programming (TFP) [1], is a software development practice that has recently gained a lot of attention from both software practitioners and researchers, and is becoming a primary means of developing software worldwide [5,6,7,8,9,10,11,12,13,14]. Moreover, one of the most important advantages of TDD is high coverage rate. In this paper, we present how we evaluated the impact of TDD practice on software development productivity and activity. The design of the study is based on Goal-Question-Metric (GQM) approach [15], and can be easily replicated in different industrial contexts, where the number of subjects that may be involved in an empirical study is often limited and the generalization of the results is not the key issue.

## 2 Related work

Several empirical studies have focused on TDD, as promising alternative to traditional, test-last development (TLD), also called test-last programming (TLP). Some of them concern productivity. Müller and Hagner [5] report that TDD does not accelerate the implementation, and the resulting programs are not more reliable, but TDD seems to support better program understanding. George and Williams [6,7] show that TDD developer pairs took 16% more time for development. However, the TDD developers produced higher quality code, which passed 18% more functional black box test cases. Other empirical results obtained by Williams et al. [8,9] are more optimistic, as TDD practice had minimal impact on developer productivity, while positive one on defect density. Geras et al. [10] report that TDD had little or no impact on developer productivity. However, developers tended to run tests more frequently when using TDD. Erdogmus et al. [16] conclude that students using test-first approach on average wrote more tests than students using test-last approach and, in turn, students who wrote more tests tended to be more productive. Madeyski [11] conducted a large experiment in academic environment with 188 students and reports that solo programmers, as well as pairs using TDD, passed significantly fewer acceptance tests than solo programmers and pairs using test-last approach, ($p = .028$ and $p = .013$ respectively). Bhat and Nagappan [12] conducted two case studies in Microsoft and report that TDD slowed down the development process 15%-35%, and decreased defects/KLOC 2.6-4.2 times.. Canfora et al. [13] report that TDD significantly slowed down the development process. Müller [14] conducted a unique empirical study and concludes that the TDD practice leads to better-testable programs.
Summarizing, existing studies on TDD are contradictory. The differences in the context in which the studies were conducted may be one explanation for such results. Thus, case study conducted and valid in a project's specific context is a possible solution that can be applied in industrial projects.

## 3 Empirical study

It is important to present the context of the project. `Java` and `AspectJ` programming languages, and hence aspect-oriented programming (AOP) [17], were used to implement the web-based system. The presentation tier was provided by Java Server Pages and Servlets. The persistence layer was used to store and retrieve data from XML files. An experienced programmer, with 8 years of programming experience and recent industrial experience, classified as E4 according to Höst et al. [18] classification scheme (i.e. recent industrial experience, between 3 months and 2 years), was asked to develop a web-based system for academic institution.
The whole development project consisted of 30 user stories. Additionally, three phases (with random number of users stories in each phase) could be distinguished. The first phase (10 user stories) was developed with traditional, TLD approach, the second (14 user stories) with TDD and the last 6 user stories again with TLD approach, see Figure 1.

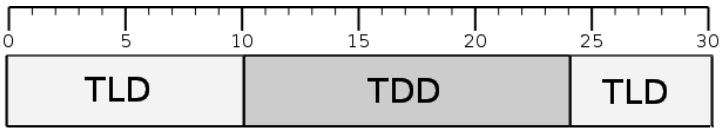


**Fig. 1.** User stories divided into development phases

### 3.1 User requirements

The project was led with the eXtreme Programming (XP) methodology, as TDD is a key practice of XP. Therefore, it seems reasonable to evaluate TDD practice in the context of XP. Although some practices (such as pair programming) were neglected, user stories were used for introducing requirements concerning the developed system. The whole set of 30 user stories was prepared to outline the system, which is a web-based paper submission and review system. It defines different user roles such as *Author*, *Reviewer*, *Chair* and *Content Manager*, and specifies multilevel authentication functionality. The system involves the management of papers and their reviews on each step in their life cycle. Additionally the application provides access to accepted and published papers to all registered and unregistered users allowing users to select lists of articles based on earlier defined set of criteria (e.g. published, accepted works). The system supports a simple repository of articles with uploading of text files and versioning.

### 3.2 Procedure

The Theme/Doc approach [19] provides support for identifying crosscutting behaviour and was used to decompose the system into aspects and classes. Themes are encapsulations of concerns and therefore are more general than classes and aspects. They may represent a core concept of a domain or behaviour triggered by other themes. The procedure used during the TLD phase is presented in Figure 2, and the analogous one for the TDD phase in Figure 3. In TLD phase the participant chooses a user story and then develops its themes (only these parts which are valid for a specified user story). After finishing each theme, a set of unit tests is written. When the whole user story is complete, the participant may perform a system refactoring. The TDD phase differs in first steps. After choosing a user story, the participant chooses a theme and writes tests as well as production code to the specified theme in small test first, then code cycles. The activity is repeated (for other themes related to the selected user story) until the user story is completed. From this point the procedure is the same as in traditional approach.

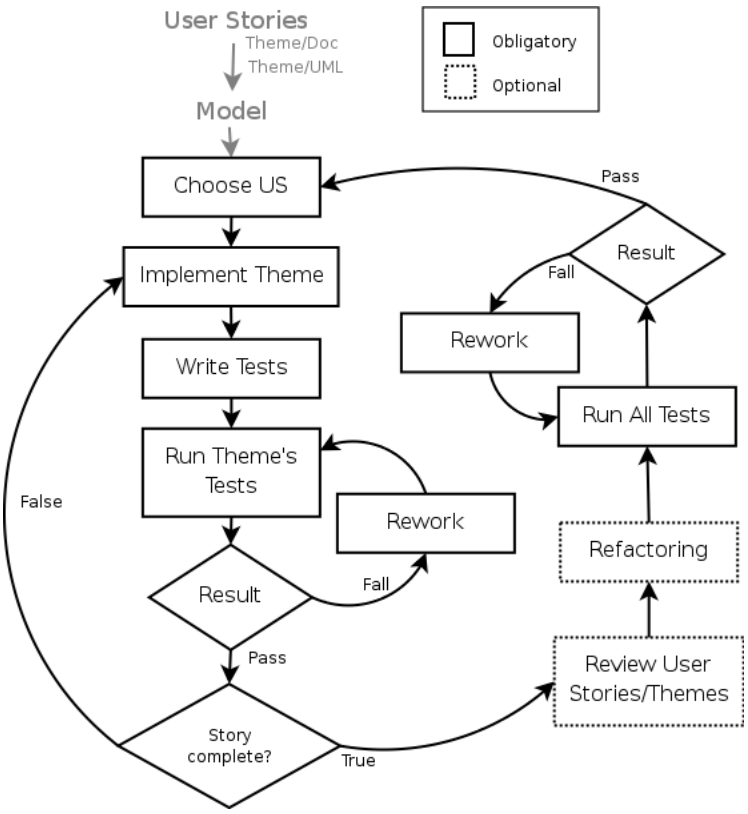


**Fig. 2.** Development procedure in the TLD phase

### 3.3 Validity evaluation

There are always threats to the validity of an empirical study. In evaluating the validity of the study, we follow the schema presented in [20]. As a *statistical conclusion* validity threat, we see the lack of inferential statistics in the analysis of the results. However, the points at which TDD is introduced and withdrawn are randomly determined to facilitate analysis. As with most empirical studies in software engineering, an important threat is process conformance, represented by the level of conformance of the subject (i.e. developer) to the prescribed approach. Process conformance is a threat to statistical conclusion validity, through the variance in the way the processes are actually carried out [21]. It is also a threat to construct validity, through possible discrepancies between the processes as prescribed, and the processes as carried out [21]. Process conformance threat was handled by monitoring possible deviations, with the help of ActivitySensor plugin integrated with Eclipse IDE (Integrated Development Environment). ActivitySensor controlled how development approaches (i.e. TDD or TLD) were carried out (e.g. whether tests were written before related pieces of a production code).

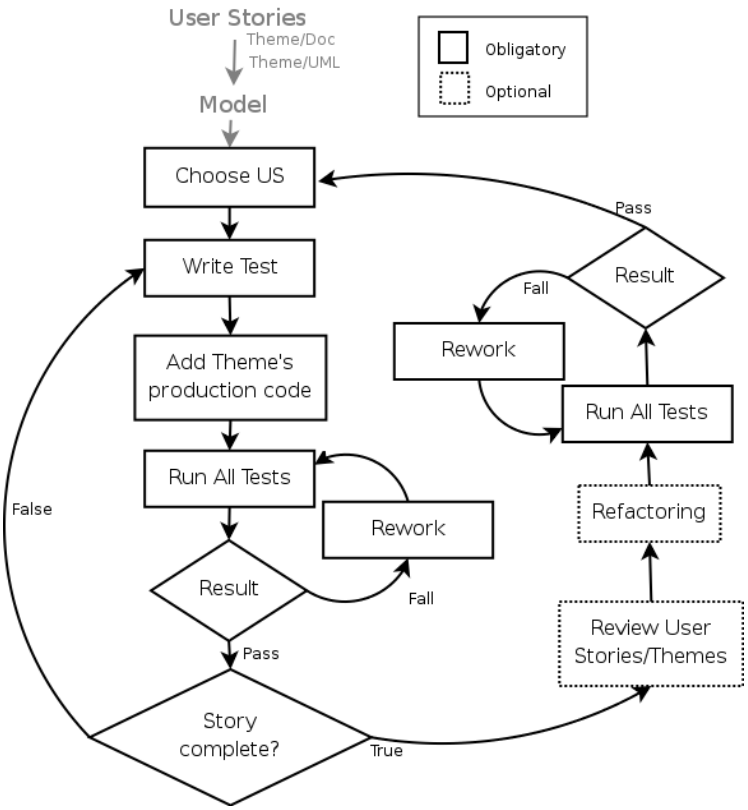


**Fig. 3.** Development procedure in the TDD phase

Moreover, the subject was informed of the importance of following assigned development approach in each phase.

The mono-operation bias is a *construct* validity threat, as the study was conducted on a single requirements set. Using a single type of measures is a mono-method bias threat. To reduce this threat, different measures (e.g. number of acceptance tests, user stories, lines of code per unit of effort) were used in the study, as well as the post-test questionnaire was added to enable qualitative validation of the results. It appeared that the subject was very much in favour of TDD approach, which is in line with the overall results. Interaction of different treatments is limited, due to the fact that the subject was involved in one study only. Other threats to construct validity are social threats (e.g. hypothesis guessing and experimenter's expectances). As neither the subject, nor the experimenters have any interest in favour of one approach or another, we do not expect it to be a large threat.

*Internal* validity of the experiment concerns the question whether the effect is caused by independent variables, or by other factors. A natural variation in human performance, as well as maturation, is a threat. Pos-

sible diffusion, or imitation of treatments were under control with the help of ActivitySensor Eclipse plugin.
The main threat to the *external* validity is related to the fact that subject population may not be representative to the population we want to generalize. However, the programmer's experience is typical of a young programmer, with solid software engineering academic background and recent industrial experience. Thus, it seems to be relatively close to the population of interest.

## 4 Measurements definition

The empirical study was conducted using the GQM method, described in [15]. The measurement definition relates to the programmer's productivity (in terms of source code lines written, implemented user stories, and number of acceptance tests passed).

**Goal:** The analysis of *the software development process* for the purpose of *evaluation of the TDD approach impact*, with respect to *software development productivity and activity*, from the point of view of *the researchers*, in the context of *a web based, aspect-oriented system development.*

**Questions:**

- **Question 1:** *How does TDD affect the programmer's productivity in terms of the source code lines written per unit of effort?*
  **Metrics:** NCLOC (Non Comment Lines Of Code) per unit of effort (programming time) is one of productivity measures. However, NCLOC per unit of effort tend to emphasize longer rather than efficient, or high-quality programs. Refactoring effort may even results in negative productivity measured by NCLOC. Therefore, better metrics of a programmer's productivity will be used.
- **Question 2:** *How does TDD affect a programmer's productivity in terms of user stories provided per unit of effort?*
  **Metrics:** Because in XP methodology the user requirements are introduced as user stories, the implementation time of a single user story may be considered as a productivity indicator. Therefore, the number of user stories developed by a programmer per hour is measured.
- **Question 3:** *How does TDD affect a programmer's productivity in terms of Number of Acceptance Tests Passed per unit of effort?*
  **Metrics:** Because user stories have diverse sizes, we decided to measure the programmer's productivity using acceptance tests, as NATP (Number of Acceptance Tests Passed) per hour better reflects the project's progress and programmer's productivity. There were 87 acceptance tests specified for the system.
- **Question 4:** *How does TDD affect a programmer's activity in terms of passive time, compared with the total development time?*
  **Metrics:** The programmer's productivity may be expressed as a relation of active time $T_A$ to the total time (sum of active and passive

times $T_A + T_P$) spent on a single user story implementation. The active time may be described as typing and producing code, whilst the passive time is spent on reading the source code, looking for a bug etc. The ActivitySensor plugin [22] integrated with Eclipse IDE allows to automatically collect development time, as well as to divide total development time into active and passive times. A switch from active to passive time happens after 15 seconds of a programmer's inactivity (the threshold was proposed by the activity sensor authors). To separate passive time from breaks in programming the passive time counter is stopped (after 15 minutes of inactivity) until a programmer hits a key.

## 5 Results

The whole development process took 112 hours. The finished system was comprised of almost 4000 lines of source code (without comments, imports etc.). The system had 89 interfaces, classes, and aspects. There were 156 unit tests written to cover the functionality. Branch coverage was over 90%.

### 5.1 Productivity metrics analysis

Although the XP methodology puts pressure on source code quality (programming is not just typing!), the differences in software development productivity are essential. Table 1 contains a comparison of productivity metrics in TLD1, TDD and TLD2 phases. TLD1 and TLD2 phases shown in Figure 1 are treated jointly in the last column named TLD.

| | TLD1 | TDD | TLD2 | TLD (TLD1 and TLD2 combined) |
|---|---|---|---|---|
| Implementation time/US [h] | 6.42 | 2.32 | 2.50 | 4.97 |
| Lines of code/US | 159.70 | 107.21 | 133.17 | 149.75 |
| Lines of code/h | 24.76 | 46.18 | 53.27 | 30.14 |

**Table 1.** Productivity comparison in all development phases

It appeared that the implementation time of a single user story during the TLD phase took, on average, almost 5 hours, while during the TDD phase only 2.32 hours, see Table 1. User stories are common units of requirements, but their size and complexity level are not equal. The average size (expressed in lines of code) of a user story, developed with TLD approach, was almost 1.5 times bigger than a user story developed during the TDD phase, see Table 1. It may mean that the code written in TDD phase is more concise than its TLD equivalent.

The next comparison concerns the number of lines of code written per one hour. The results favour the TDD approach with average 46.18 lines above the TLD with 30.14 lines per hour, see Table 1 and Figure 4.

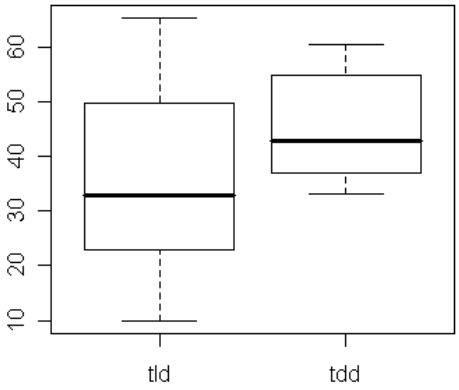


**Fig. 4.** Boxplot of average number of lines of code per hour in TLD, and TDD phases

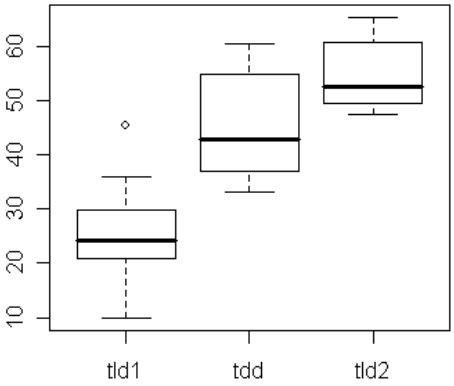


**Fig. 5.** Boxplot of average number of lines of code per hour in TLD1, TDD, and TLD2 phases

More deatailed observation of boxplots, in Figures 4 and 5, allows to reveal an interesting regularity. Although the TDD phase is characterised by higher productivity in juxtaposition with TLD phase (TLD1 and TLD2 treated jointly), when comparing all three phases, the productivity increases with the system's evolution. It may be explained by gaining skills and experience by the programmer, as well as making the programmer more familiar with the requirements, with each completed user story.

The productivity may be measured as a number of passed acceptance tests that cover added functionality, divided by number of hours spent on implementation. When looking at the development cycle divided into two phases (TDD vs. TLD), we measured the following values of passed acceptance tests per hour: 1.44 for TDD and 0.99 for TLD (TDD approach is characterised by a faster functionality delivery, see Figure 6). But when analysing the development cycle as 3 phases (TLD1, TDD and TLD2, see Figure 7), we found that the last two phases were similar while TLD1 phase was considerably worse.

### 5.2 Analysis of programming activities

Figure 8 presents a proportion of passive time to total development time. We can observe that in first (TLD1) phase the passive time took the

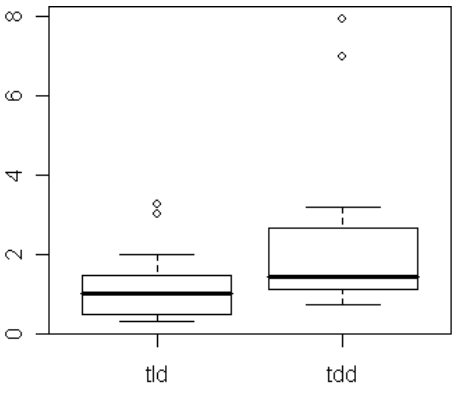


**Fig. 6.** Boxplot of the number of acceptance tests passed per hour in TLD, and TDD phases

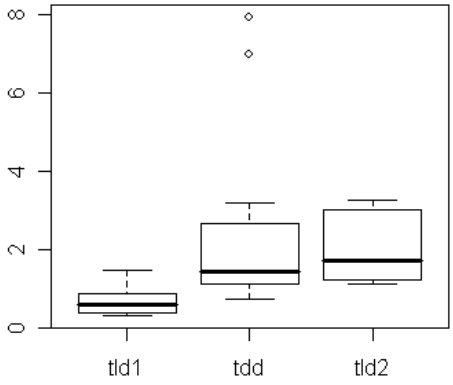


**Fig. 7.** Boxplot of the number of acceptance tests passed per hour in TLD1, TDD, and TLD2 phases

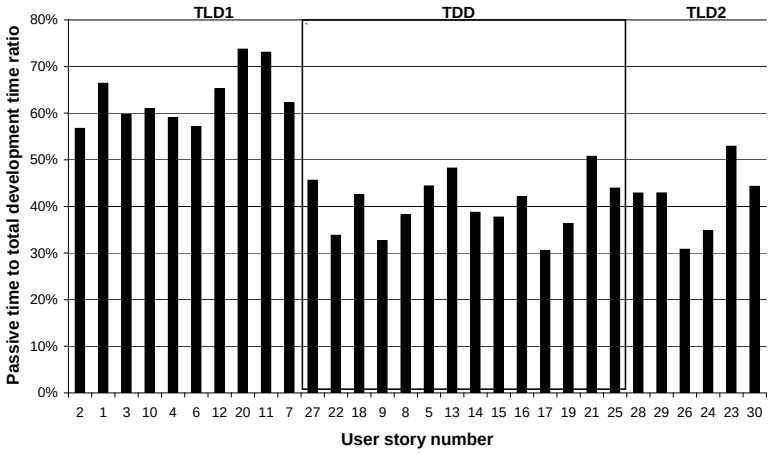


**Fig. 8.** The passive time to total development time proportion during the project

majority of total time (over 50%). This rule changed when the testing metohod was switched (the passive time only once exceeded 50% level).

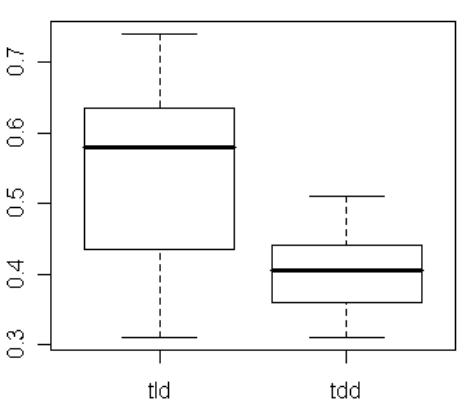


**Fig. 9.** Boxplot of the proportion of passive to overall development time in TLD, and TDD phases

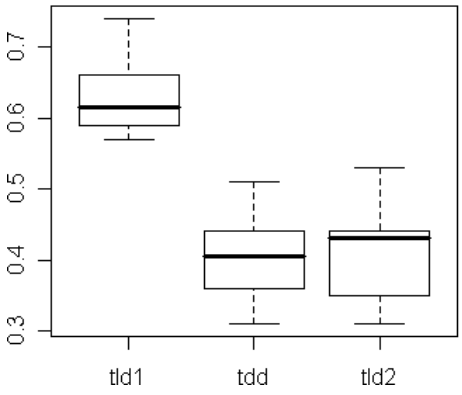


**Fig. 10.** Boxplot of the proportion of passive to overall development time in TLD1, TDD, and TLD2 phases

The boxplots of active and passive times are presented in Figures 9 and 10. We can observe that the passive time is higher in TLD phase. However, the difference is not so obvious when we analyse each phase separately, as results of TDD and TLD2 phases are similar.

## 6 Conclusions

If we analyse the development process divided into two phases (TLD and TDD), the programmer's productivity in TDD phase is definitely higher. A possible explanation is that TDD approach limits the feedback cycle length to minutes. Thus, the extent of potential bug is usually limited (a programmer knows exactly where should look for an improper system behaviour). Another plausible explanation, why TDD may increase software development productivity, is that improving quality by fixing defects at the earliest possible time (by means of continuous and rigorous testing and refactoring) costs up front but it pays off in the long run.
However, when the process is divided into three phases (TLD1, TDD, TLD2) a different pattern appears. In the case of source code lines written per unit of effort (Question 1) the productivity increases with the project development progress. The proportion of passive to overall development time (Question 4) falls in TDD phase, but in the last two phases (TDD and TLD2) is similar. In the case of user stories per unit of effort (Question 2), as well as acceptance tests per unit of effort (Question 3)

the programmer's productivity increases in TDD phase, whilst in the last two phases (TDD and TLD2) is similar as well. A plausible explanation, why productivity in TLD2 phase does not fall, may be that the programmer gains experience, as well as knowledge of the application domain, during the course of the project. Thus not only TDD, but also experience and knowledge of the application domain would drive productivity.
The study can benefit from several improvements before replication is attempted. The most significant one is to replicate the study finishing with TDD in the fourth phase. In order to conclude that TDD has in fact positive impact on productivity, it might be advisable to conduct an experiment securing a sample of large enough size to guarantee a high-power design.

## Acknowledgements

The authors thank Adam Piechowiak for ActivitySensor Eclipse plugin development.
This work has been financially supported by the Ministry of Education and Science as a research grant 3 T11C 061 30 (years 2006-2007).

## References

1. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change. 2nd edn. Addison-Wesley (2004)
2. Maxwell, K., Forselius, P.: Benchmarking Software-Development Productivity - Applied Research Results. IEEE Software **17**(1) (2000) 80–88
3. Fowler, M.: Cannot Measure Productivity (accessed March 2007) http://www.martinfowler.com/bliki/CannotMeasureProductivity.html
4. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
5. Müller, M.M., Hagner, O.: Experiment about test-first programming. IEE Proceedings - Software **149**(5) (2002) 131–136
6. George, B., Williams, L.A.: An Initial Investigation of Test Driven Development in Industry. In: Proceedings of the 2003 ACM Symposium on Applied Computing (SAC '03), ACM (2003) 1135–1139
7. George, B., Williams, L.A.: A structured experiment of test-driven development. Information and Software Technology **46**(5) (2004) 337–342
8. Williams, L., Maximilien, E.M., Vouk, M.: Test-Driven Development as a Defect-Reduction Practice. In: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03), Washington, DC, USA, IEEE Computer Society (2003) 34–48
9. Maximilien, E.M., Williams, L.A.: Assessing Test-Driven Development at IBM. In: Proceedings of the 25th International Conference on Software Engineering (ICSE '03), IEEE Computer Society (2003) 564–569

10. Geras, A., Smith, M.R., Miller, J.: A prototype empirical evaluation of test driven development. In: Proceedings of the 10th International Symposium on Software Metrics (METRICS '04), IEEE Computer Society (2004) 405–416
11. Madeyski, L.: Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. In Zieliński, K., Szmuc, T., eds.: Software Engineering: Evolution and Emerging Technologies. Volume 130 of Frontiers in Artificial Intelligence and Applications. IOS Press (2005) 113–123 http://madeyski.e-informatyka.pl/download/Madeyski05b.pdf
12. Bhat, T., Nagappan, N.: Evaluating the efficacy of test-driven development: industrial case studies. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06), New York, NY, USA, ACM Press (2006) 356–363
13. Canfora, G., Cimitile, A., Garcia, F., Piattini, M., Visaggio, C.A.: Evaluating advantages of test driven development: a controlled experiment with professionals. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06), New York, NY, USA, ACM Press (2006) 364–371
14. Müller, M.M.: The Effect of Test-Driven Development on Program Code. In Abrahamsson, P., Marchesi, M., Succi, G., eds.: XP. Volume 4044 of Lecture Notes in Computer Science., Springer (2006) 94–103
15. Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. In: Encyclopedia of Software Engineering. (1994) 528–532
16. Erdogmus, H., Morisio, M., Torchiano, M.: On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering **31**(3) (2005) 226–237
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Aksit, M., Matsuoka, S., eds.: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241 of Lecture Notes in Computer Science., Springer (1997) 220–242
18. Höst, M., , Wohlin, C., Thelin, T.: Experimental Context Classification: Incentives and Experience of Subjects. In: Proceedings of the 27th International Conference on Software Engineering (ICSE '05), New York, NY, USA, ACM Press (2005) 470–478
19. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design : The Theme Approach. Addison-Wesley (2005)
20. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA (2000)
21. Sørumgård, L.S.: Verification of Process Conformance in Empirical Studies of Software Development. PhD thesis, The Norwegian University of Science and Technology (1997)
22. ActivitySensor project (accessed March 2007) http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.ActivitySensor