Information and Software Technology 57 (2015) 494–508

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Besouro: A framework for exploring compliance rules in automatic TDD behavior assessment

Karin Becker [a,*], Bruno de Souza Costa Pedroso [b], Marcelo Soares Pimenta [a], Ricardo Pezzuol Jacobi [b]

[a] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
[b] Departamento de Computação, Universidade Nacional de Brasília, Brasília, Brazil

ARTICLE INFO

Article history:
Received 29 October 2013
Received in revised form 30 March 2014
Accepted 2 June 2014
Available online 19 June 2014

Keywords:
TDD
Conformance assessment
Framework

ABSTRACT

Context: The improvements promoted by Test-Driven Design (TDD) have not been confirmed by quantitative assessment studies. To a great extent, the problem lies in the lack of a rigorous definition for TDD. An emerging approach has been to measure the conformance of TDD practices with the support of automated systems that embed an operational definition, which represent the specific TDD process assumed and the validation tests used to determine its presence and quantity. The empirical construction of TDD understanding and consensus building requires the ability of comparing different definitions, evaluating them with regard to practitioners' perception, and exploring code information for improvement of automatic assessment.
Objective: This paper describes Besouro, a framework targeted at the development of systems for automatic TDD behavior assessment. The main rationale of Besouro's design is the ability to compare distinct operational definitions, evaluate them with regard to users' perception, and explore code information for further analysis and conformance assessment improvement.
Method: We developed an architecture with clear separation of concerns, which enables to vary: (a) the atomic events and respective metrics to be collected from developing and testing environments; (b) the organization of atomic events in streams of actions or processes; and (c) the classification and assessment components for each set of operational definitions adopted. The architecture also includes a mechanism for on-line user assessment awareness and feedback, and integrates event-related information with the respective code in a code version system.
Results: We illustrate the usefulness of Besouro's features for understanding the actions and processes underlying TDD through a prototype developed to support an experiment based on user feedback. We show how it was possible to compare variations of a same operational definition by exploring users' feedback, and use source code to improve the automatic classification of TDD practices.
Conclusion: Understanding the actions and processes underlying successful TDD application is key for leveraging TDD benefits. In the absence of a rigorous definition for TDD, the proposed approach aims at building consensus from experimentation and empirical validation.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Test-driven design (TDD) is a software development practice that has emerged in conjunction with the rise of agile methods [1]. Both are rooted in incremental, iterative and evolutionary process models, but have been compiled, shaped and refined by experienced software practitioners [2]. Although TDD is independent from agile methods, it was popularized by Extreme Programming (XP) [3], where it was related to other key practices like automated tests, simple design, refactoring, continuous integration, and small versions [4,5]. TDD replaces the traditional "code then test" cycle, by instructing a programmer to write production code only after writing a failing automated test case. The design of the system is improved incrementally by refactoring existing code. Thus, the TDD process consists of fine-grained cycles of the following three programming tasks: write test code, write production code and refactor [5,6].

Many claims have been made about the advantages of TDD, particularly with regard to improvements in software quality and productivity [6]. However, quantitative assessment studies reveal ambiguous results concerning TDD benefits, regardless the context (industry vs. academia), subjects (experienced vs. novice) and experiment type (controlled experiment, case study, pilot project)

* Corresponding author. Tel.: +55 51 33086150.
E-mail addresses: karin.becker@inf.ufrgs.br (K. Becker), brunopedroso@gmail.com (B. de Souza Costa Pedroso), mpimenta@inf.ufrgs.br (M.S. Pimenta), jacobi@unb.br (R.P. Jacobi).

http://dx.doi.org/10.1016/j.infsof.2014.06.003
0950-5849/© 2014 Elsevier B.V. All rights reserved.

[7]. Many quantitative assessment studies and systematic reviews describing and comparing experiments [7–20] report a diversity of findings, which are often inconclusive or contradictory. Without concrete evidence, TDD supporters and detractors will continue to issue subjective and biased opinions about the advantages or shortcoming s of TDD.

To a great extent, the problem with the lack of concrete evidence lies in the lack of a rigorous definition for TDD [6,7,9,2,14,15]. On the one hand, if studies do not express how TDD was defined or practiced, results from distinct studies cannot be accurately compared. On the other hand, it opens the possibility of inconsistent understanding or application of TDD by the people participating in the studies.

Having emerged from practice, rather from academic studies, TDD has been pragmatically described using general guidelines or metaphors [4,5,21]. Studies stress two common misconceptions surrounding the practice of TDD [6,2,8]. First, TDD is not a testing technique: it is a development technique that relies on automated testing for steering production activities. Therefore, the resulting test cases in the TDD context exhibit a very limited scope, exclusively related to the functionality to be coded. Traditional quality assurance activities must follow to complete the verification process. A second common misconception is that TDD is not merely an incremental approach based on test-first. TDD recommends the alternation of very short cycles of intertwined test case definition, production and refactoring. However, many practitioners tend to define a significant number of test cases upfront. In the ideal TDD process [5], the increments are small enough to be captured by a single test: write a single test and follow it up by implementing just enough functionality to satisfy that test, all without breaking the previous tests. Despite all these elaborations, TDD remains deceptively simple to describe, but deeply challenging to put into practice effectively.

An emerging approach has been to measure the conformance of TDD practices with the support of automated systems. In general, automatic TDD recognition systems [19,22,24–27] provide support the following activities: (a) capturing atomic events in development and testing environments (possibly together with other metrics); (b) mapping these events into streams of actions or processes, also referred to as *episodes* [6,19]; (c) classifying them according to TDD-related practices, and (d) assessing conformance. The automatic recognition of behavior and conformance assessment require an explicit, standard and robust interpretation of the development behaviors underlying TDD. These systems were designed for different assessment purposes, such as comparing test-first and test-last approaches [25,26], TDD learning [24], TDD compliance [19,22] and testing governance [27]. However, even when the purpose is the same, they use different definitions and rules for detecting and assessing TDD behavior compliance. Indeed, they vary with respect to the events they consider and the rules that detect them; how these are interpreted and organized into valid processes; as well as on the criteria used for conformance assessment.

In the lack of a consensual definition for TDD or even compliance criteria, Kou et al. [19] suggest the adoption of an *operational definition* [28], i.e. a definition in terms of the specific process or set of validation tests used to determine its presence and quantity. The adoption of an operational definition for TDD is important for addressing compliance assessment in both laboratory and real world settings. First of all, by considering an explicit definition, experiments become comparable, and their findings, conclusive. Second, assumed definitions can be embedded into the recognition system to automatically detect and assess developers' TDD behavior. Another obvious advantage of this type of system is their support for TDD-related experiments [7,18,20,22,23] and pedagogical activities [24,17]. Finally, a less obvious but essential benefit is the possibility of proposing, experimenting and comparing various definitions for TDD practices, as a means to reach understanding and agreement in a more rigorous definition and its key aspects. Understanding the actions and processes underlying successful TDD application is thus key for leveraging TDD benefits.

The empirical construction of both understanding and consensus requires the ability of comparing different definitions, evaluating them with regard to TDD practitioners' perception, exploring code information for further analysis and improvement of automatic assessment, among others.

In this paper, we take initial steps towards this ambitious requirement by proposing Besouro,[1] a compliance assessment framework targeted at the development of systems for automatic TDD behavior evaluation. Compared to related work, Besouro integrates in a single framework a set of features independently proposed in similar systems [19,24,25]. The striking features of Besouro framework are: (a) a modular and decoupled architecture, with a clear separation of concerns; (b) on-line functionality allowing users to provide immediate feedback on their perception of assessed TDD behavior, and (c) persistence of assessment data and respective code in an integrated way, with an automatic version control system (VCS). Besouro was designed by the re-engineering of Zorro [19], a sophisticated TDD conformance assessment system. The improved architecture's modularity makes it easier to vary all the operationalization of the development environment for assessing TDD practices conformance, from the atomic events captured, to the conformance rules assumed. The on-line feedback functionality has been proposed in TDDGuide [24] for learning purposes. In Besouro, it also enables the immediate confrontation between the perception of experienced practitioners and the operational definitions assumed. Finally, Besouro explores the benefits of an automatic VCS, such that all assessed behavior can be traced back to code for further inspection and off-line analysis. Unlike the system reported in [25], Besouro assessment data and source code are integrated in a single VCS persistence structure. Both the on-line feedback mechanism and the off-line analysis enabled by the VCS, may help to improve the processing rules for automatic TDD conformance assessment; to understand the key actions and properties of testing and production code in the TDD context; as well as to shed light to common misconceptions and mistakes.

We illustrate the usefulness of Besouro's features for understanding the actions and processes underlying TDD, by applying it in a real context. Using Zorro as baseline, we show how it was possible to compare variations of a same operational definition by exploring users' feedback, and how to use source code associated with each episode saved in a VCS to analyze ways to improve the automatic classification of episodes.

The rest of this paper is structured as follows. Section 2 describes related work. Section 3 provides more details on Zorro TDD assessment system, since it is used as baseline. Section 4 provides an overview of Besouro, describing its architecture and main contributions. Section 5 illustrates the use of Besouro to build a prototype, which enabled an experiment that compares variations of Zorro's operational definition. Conclusions and future work are discussed in Section 6.

2. Related work

As TDD is gaining growing attention, extensive material has been written to educate developers in the practice (e.g. [5,6,29]). Universities are also actively assessing how to incorporate it in

[1] The name is an intentional pun: "besouro" is the Portuguese translation for the term "bug".

their curriculum [30]. Beck [5] summarizes the practice of TDD with a five steps process of low granularity:

1. Write a new test case.
2. Run all the test cases and see the new one fails.
3. Write just enough code to make the test pass.
4. Re-run the test cases and see they all pass.
5. Refactor code to remove duplication.

There exists a consensus that the TDD process is simple and consists of cycles involving three types of programming tasks: write test code, write production code and refactor [5,6,2]. Production code contains the actual functionality of the software system and is shipped to the customers, whereas test code is usually not shipped because it is meant for testing purposes only. Refactoring can involve both types of code. The limit of a cycle is defined in terms of unit test failure or success. A passing unit test is a unit test where all its tests cases pass, whereas in a failing unit test, one or more test cases fail. Obviously, the order of the programming tasks is crucial to TDD conformance. We adopt the term *episode* as an alternative for cycle, which is defined in [6] as "as the sum of activities that take place between successive stable states of a piece of code under development". A code is stable when all its respective tests pass. Beck [5] insists that programmers must consider a single, tiny functionality at a time, and start from the simplest success case. Other TDD experts [6,13] are more flexible, and advise programmers to strive to keep episode length as short as it's practicable to increase visibility, feedback frequency, and confidence in progress.

Despite its simplicity, putting TDD into practice effectively remains a challenge. Many authors [2,6,14,8] have discussed issues such as what the practice really entails in a software development process, how it differs from testing, its limitation as a design technique, the process for its adoption, common misconceptions and errors, among others. A study [31] analyzes the TDD practices of 218 volunteers, 90% of which claim that their TDD experience is characterized by projects in the industry. The survey asks these volunteers details on how they practice TDD in terms of activities such as refactoring and test case definition. This study provides some insights on how conceptions of TDD may vary in practical scenarios. One interesting point is refactoring. The study reveals that almost half of the subjects frequently skip refactoring as the last step of each cycle. A significant portion of them state that they perform refactoring from time to time, navigating through the code and changing parts they feel that must be improved. Another interesting aspect is the granularity of the cycle. The study reveals that the first step often involves a suite of tests, rather than a single test case, i.e. developers frequently start from a complex test scenario, rather than a tiny piece of functionality. The study classifies the above examples as errors based on [5], but if compared to the definitions and discussions in [6,13], the process could be considered conformant, characterizing less experience.

There have been many attempts to provide empirical evidence for TDD claimed benefits. However, quantitative studies, such as [10–15,17,18,20], report contradictory or deceptive results. Systematic reviews, such as [7–9,16], have confirmed the diversity of findings and the weak evidence of TDD benefits (if any). A comprehensive analysis is developed by Shull et al. in [7], which compares the results of 33 quantitative studies involving controlled experiments, pilot projects and industrial cases. They classify as "high rigor studies" the ones that somehow meet the following conditions: qualified subjects; the TDD process matches some textbook definition or show some degree of conformance; and significance of tasks. Their results also revealed little evidence of benefits or contradictory results, particularly when considering "high rigor studies". In the search of stronger evidences, Munir et al. [9] developed another systematic review in which the primary studies are classified according to "rigor" and "relevance", where rigor is defined in terms of the applied research method, and relevance refers to the practical impact and realism of the research setup. They concluded that the more rigorous and relevant are the primary studies, the less contradictory are the results. In particular, they highlight that there was strong evidence that external quality can be improved at the expense of a loss of productivity for high rigor/relevance studies. However, their study considers as relevant only industrial cases, and TDD process conformance, although explicitly defined, is assumed, rather than explicitly investigated. The lack of process conformance is a thus a threat of validity in all these studies.

Process conformance is a threat to both the statistical conclusion validity of an empirical study, through the variance in the way the processes are actually carried out, as well as to the construct validity, through possible discrepancies between the prescribed processes, and the processes as carried out [20]. However, rules for assessing TDD process conformance are difficult to define due to the lack of formal foundations for TDD practices. Programmers with different experience levels, preferences, reasoning processes, and established work patterns will apply TDD in different ways [6,24]. Therefore, operational definitions have been formalized and adopted either as a support for developing experiments, or automatic TDD conformance assessment systems.

Based on atomic events and related data captured from the development and testing environments, automatic TDD recognition systems abstract events in terms of the underlying practices, and asses their conformance. Table 1 summarizes the main characteristics of the TDD automatic recognition systems discussed in the remaining of this section. For each system, Table 1 describes its purpose, the main dependencies with regard to other systems, how it abstracts the atomic events in terms of coarse-grained behavior, whether it attempts to recognize refactoring, how it assesses conformance, and whether it provides users with explicit feedback about the assessed behavior. The last row of Table 1 summarizes Besouro's features with regard to these same characteristics. Besouro will be discussed in detail in Section 4. The gray cells in Table 1 highlight the features from other systems (namely Zorro, TDDGuide and the system described in [25]) that were adapted and integrated in Besouro.

Examples of systems designed to support specific experiments contrasting test-first vs. test-last approaches are [23,26]. The experiment reported in [23] compares productivity metrics in test-first and test-last approaches. It divides the development of a web-based system into phases in which TDD is/is not used. An Eclipse plugin collects events, metrics and possible deviation data. Hence, is limited to collecting data about assumed behavior that can be analyzed offline. Other related experiences are reported in [20].

TestFirstGauge [26] automatically identifies coding and test execution activities, together with related metrics, and organize them in cycles. It relies on Hackystat sensors [32] to collect atomic actions and metrics (time, code length) in development and testing environments, which are stored into files. These entries are then organized into three types of blocks (TestCode, ProductionCode and TestExecution), and sequenced to compose a cycle. A cycle is a successful TestExecution block preceded by any combination of other types of blocks. TestFirstGauge exports to a spreadsheet report that contains the details of each cycle and associated metrics. It allows insights underlying cycles (e.g. experienced programmers tend to display shorter TDD cycles, compared to novices), but it does not attempt to automatically classify cycle conformance. It also does not identify refactoring activities.

Systems such as [25,24,22,19] are more evolved, and additionally encompass TDD conformance assessment functionality. The architecture defined in [25] was designed to support an experiment that compares novice and experienced programmers in the practice of TDD. In the front-end, plugins capture and log

Table 1
Comparison of automatic TDD recognition systems and influence on Besouro's features.

Related Work or System	Purpose	Main Dependencies	Abstracted Behavior	Refactoring	Additional Information	Conformance Assessment	User Feedback
[23]	Productivity in TDD	activitySensor, spreadsheet	Production/ test code, test execution organized in stories	no	code length, time , Number of stories, passed tests	No	No
TestFirstGauge [26]	Insights on TDD practice	Hackystat, spreadsheet	TestCode, ProductionCode and TestExecution organized in cycles	no	Time, code length, number of blocks	No	No
[25]	Novice vs. experienced TDD practitioners	UserActionLogger, JUnitActionLogger , database, VCS	Creation and modification of production/test code, testing, organized in cycles	Yes, with user intervention	Modified source files with timestamps, number of tests and their result, extracted from code versions (VCS)	Strong and weak conformance rules	No
TDDGuide [24]	TDD Learning	Aspect-based framework	Events on the coding and testing space (creation, modification) which, when detected, are compared to predicates	no	Number of Passing/Broken tests	on track, deviation, warning	Yes
Zorro [19]	TDD compliance	Hackystat, SDSA, Jess	TestCode, ProductionCode and TestExecution organized in episodes, which are classified according to 8 categories	yes	Code size and structure	Context-sensitive conformance	No
SEEKE [27]	TDD Conformance assessing and governance	Hackystat, Esper, Adept2	Modified OpenUP processes	yes	Workflow related information	Workflow conformance	No
BESOURO	Build systems to explore and compare operational definitions	Listeners, JESS, VCS	Varying, according to the classification component implemented. Implemented classification component assumes the same categories as Zorro	yes	Code size and structure, plus ability to trace actions back on code versions	Varying, according to the classification component implemented	Yes

events in Eclipse and JUnit environments, and backup copies of each modified file after a compile or save operation. In the back-end, a postprocessor module extracts the name of the files that were changed, the difference between two successive versions of these files, timestamps and various types of testing information, and stores all this information in a database. The evaluator model works with the data in the database and the Java files stored in the VCS to examine conformance with regard to strong and weak conformance rules. Two conformance rules are rigorously defined: production code can only be changed after failing tests, and testing code can only be changed after passing tests. A Weak conformance rule does not require the execution of the failing test to write the respective production code. The authors adopt a definition for refactoring, but state that refactoring cannot be recognized without human intervention.

TDDGuide [24] is targeted at supporting the learning and mastering of TDD. It is integrated into Eclipse/Junit environments and guides the developer by providing on-line notifications that either encourage use of TDD (*on-track*), or warn about non-compliant behavior (*deviation, error*). An aspect-programming framework is used to detect creation and modification events into the coding and testing environments, together with metrics (e.g. number of passing tests). Rules are defined in terms of predicates that must be true when key events are detected (e.g. *NeverWriteCodeWithoutFailingTest* relates the key events *codeCreation* and *codeModification* to the predicate *numberOfFailingTests == 0&& numberOfBrokenTests ==0*). The rationale behind TDDGuide is that, with meaningful feedback, programmers can learn to master TDD, overcome difficulties, as well as correlate any changes in productivity and code quality with the way the technique is being applied. However, the intrusiveness of the TDDGuide feedback was highlighted during the evaluation of the tool.

Zorro [19,22] is a component of a client–server architecture that classifies and assesses TDD conformance behavior. It depends on two other software components: (a) Hackystat, for collecting coding and testing events; and (b) SDSA (Software Development Stream Analyzer), a stream processor that sequentially organizes events into episodes terminated by a successful test passing event. Zorro highlights the importance of the adoption of an operational definition for TDD as the basis for conformance assessment, and proposes one based on two dimensions: category of observed behavior and conformance assessment. Eight (8) categories of behavior are assumed, which are briefly described in Table 2. Rules establish the patterns of events sequences in each category (e.g. *test editing* → *test pass* is a rule for regression). Once the episode is classified, its conformance is evaluated. The adopted operational definition assumes that *test-first* episodes are conformant, whereas *long, unknown* and most *production* episodes are not. The conformance of all other categories of episode is context-sensitive, i.e. determined by the preceding and following episodes. Zorro also employs metrics related to code size to classify episodes. Further details are provided in Section 3.

SEEKE [27] is a general framework that combines software engineering task management tools with a process layer for confor-

Table 2
Zorro episode classification.

Category	Category description	Conformance
Test-first (TF)	Characterized by the creation of a test, edition of a production code and successful execution of the test suite. Compilation actions and unsuccessful test executions can take place in between	Yes
Test last (TL)	Correspond to the case where production code is written first, followed by its corresponding test suite coding, and test pass (possibly preceded by test failures)	No
Refactoring (RF)	Correspond to modifying the code (test or production) while keeping its external functionality – i.e. without breaking any tests. The rules defined to recognize this category are formulated in terms of a test pass preceded by one of following actions: test operation editing; test production coding that results less methods or statements; code refactoring; combinations of test editing with code editing that results in less methods or statements	Context-sensitive
Test-addition (TA)	Addition of a new test to the suite, possibly after changing test code, without breaking the tests	Context-sensitive
Regression (RG)	Successful execution of the test suite without being preceded by editing activities	Context-sensitive
Production (PR)	Correspond to changes in the production code, by adding functionality without writing the corresponding test. Changes in production code are determined using metrics related to the increase of code size above some user defines threshold (100 by default), methods and/or statements	Mostly Context-sensitive
Long (LO)	Episodes are either too long (more than 30 min) or that involve too many activities other than test pass (more than 200)	No
Unknown	Episodes that could not be classified using above set of rules	No

mance assessment and process governance. The architecture consists of a stack of software components, in which atomic events are detected, enriched with context information to compose complex events, and finally organized, in workflows for conformance assessment and quality governance. A TDD scenario was developed as proof-of-concept, which is based on Hackystat for atomic event detection, Esper for composite event processing, and ADEPT2 for TDD workflow modeling and assessment. Expected practices were modeled according to modified OpenUP processes. However, the experiments reported focus on the feasibility and performance of the approach, while TDD workflows and conformance checking is merely illustrated.

The following claims can be drawn from the analysis of the conformance assessment tools discussed in this section:

(i) all systems adopt (even in implicit way) an operational definition of TDD, which defines the scope of the behavior assessed, a mapping between atomic actions (and additional information) into a coarser grained behavior, and criteria for conformance assessment;
(ii) all systems abstract behavior by analyzing sequences of intertwined testing and (production/test) coding activities, organized in units that are delimited by a passing test execution. Basically they all collect the same coding and testing events, but vary in the additional information associated with these events, as well as on the mapping rules to classify behavior and assess conformance;
(iii) none of these systems share the same conformance rules and criteria (e.g. weak and strong conformance [25]; on-track, deviation and error [24]; context-sensitive conformance [19]). In addition, conformance criteria are very dependent of the purpose of the system;
(iv) the most sophisticated mapping rules and conformance assessment criteria are the ones embedded into Zorro. However, to the best of our knowledge, there is no evidence that they represent a more accurate or comprehensive operational definition for TDD compared to others.

In this paper, we propose Besouro as a step towards a better understanding of TDD, to be built from experimentation and empirical validation. The main driver of Besouro is the ability to compare existing definitions, evaluate them with regard to users' perception, and explore code information for further off-line analysis and rule improvement. Besouro was partially designed through the re-engineering of Zorro, and therefore we further describe Zorro in Section 3, before detailing Besouro in Section 4.

3. Zorro

As mentioned, Zorro is very dependent of two other applications: Hackystat and SDSA. Hackystat sensors need to recognize the following events (with the respective metrics/results): unit test invocation, compilation, refactoring (such as renaming and moving), and editing (e.g. change in file size, number of methods and test case assertions).

SDSA is a generic framework for organizing and analyzing the various kinds of data received from Hackystat as input for time-series analysis. SDSA begins by merging the events collected by various sensors into a single sequence of actions, referred to as *development stream*, in which events are ordered by time-stamp. This is followed by a process called *tokenizing*, which results in a sequence of episodes. Any SDSA application has to define the specific events to be combined to generate the development stream, as well as the boundary condition that separates streams in episodes. In the case of Zorro, the boundary condition is the successful execution of the test suite.

Zorro is actually an SDSA application responsible for mapping the episodes into higher-level TDD behaviors, and determining their conformance. The classification component in charge of that mapping is at the heart of Zorro, and it is implemented on the top of Jess [33], a rule-based system. Rules are used in three ways: (a) to transform atomic actions and collected information into relevant actions (e.g. a test creation that increases the number of methods) and detect the boundary condition of an episode, (b) to determine the category of the episode, among the 8 defined ones (Table 2), and to establish the verdict of episode conformance assessment. The eight categories are briefly described below based on the nature of activities developed by the programmer and their sequence.

It is important to stress that rules assume the repetition of some sub-patterns. For instance, in test-first rule *test creation → code editing → test failure → test pass*, but an episode may contain the subsequence *code editing → test failure* multiple times before a test passes. Jess analyses all possible subsequences for each subsequences, and output their classification according to the rules. In other words, Jess outputs several possible classification results for a same input episode. In most cases, all these categories are the same one, but there are cases in which the subsequences fit rules from different categories (e.g. production and refactoring). When this happens, Zorro considers a single category for conformance assessment. However, we could not identify specifically the criterion Zorro uses.

After having categorized the episodes, a final recursive pass through higher-level rules assesses the conformance of the episodes. Initially, the conformance of *test-first*, *test-last*, *long* and *unknown* episodes is assessed. Then, the system recursively applies to rules to resolve the context-sensitive episodes. The rationale of the authors is that some programming practices (e.g. refactoring, regression) underlie many programming strategies, test-last and test-first approaches included. Therefore, they are conformant only if they are bounded by two conformant TDD-practices. The authors advocated that this heuristic is a means to differentiate between test-first and test-last approaches.

Zorro behavior recognition rules are much more sophisticated than related work [24–26], by considering other behaviors that are inherent to real world programming scenarios. However, their dependency on neighbor episodes for conformance assessment suits the specific purpose of comparing Test-First and Test-Last approaches. Hence, it may be a limitation for the analysis of TDD in other contexts, such as to which level one may master (or how often applies) techniques that underlie TDD practices, how well one can make combined use of them, or which patterns are more correlated to successful TDD practice, to mention a few.

To illustrate this point, let us consider the example depicted in Fig. 1. Suppose that in this scenario, the programmer aims at practicing TDD, but she makes a mistake, by developing an episode non-conformant (2 – *test last*). To get back on track, she develops the corresponding tests to the code added in previous episode (3 – *test-addition*), and decides to refactor the code before defining the next test suite, intertwined with some regression activities (4–9). Then she starts over, by performing a *test-first* episode (10). Because they are not bounded by two conformant episodes, Zorro assesses episodes 3–9 as non-conformant. If measured by number of episodes, 80% of this programmer‘s behavior would be assessed by Zorro as non-conformant. On the other hand, had she not attempted to fix her mistake, and immediately after episode 2 included new test cases for future production code (even if there were code without the corresponding test assertions), followed by the same sequence of episodes, most of her behavior would be conformant. This example shows that different purposes may lead to a distinct interpretation of the conformance of these episodes, requiring thus further experimentation for validation.

Obviously the set of rules can be changed to experiment other operational definitions. However, Zorro was not designed with this purpose in mind, so changing requires one being able to identify which part of the code should be changed, and how to change it. With this regard, one of the contributions of Besouro is the re-engineering of Zorro in a more modular architecture, such that changes can be performed more easily. The immediate consequence is that it is much easier to experiment with different operational definitions. Besouro will be further explained in the next section.

4. Besouro

Besouro is an automatic TDD compliance assessment framework targeted at exploring and comparing different operational definitions. As highlighted in the last row of Table 1, Besouro integrates in a single framework, important contributions from related work, namely: (a) Zorro's sophisticated behavior recognition system; (b) TDDGuide's on-line feedback mechanism; and (c) the ability of tracing back to source code the behavior identified by means of a VCS [25].

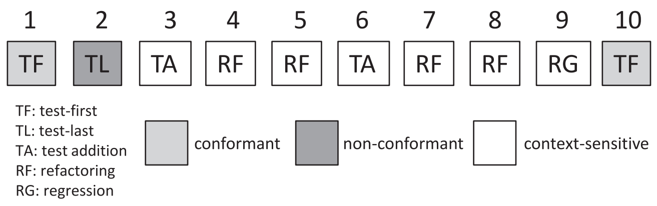

Fig. 1. Episodes sequence example.

The re-engineering of Zorro resulted in an architecture that maintains the overall information flow, but organizes code into modules according to a clear separation of concerns. Three immediate consequences derive from the new, decoupled architecture:

(i) it highlights several points of variability in automatic TDD assessment systems, for which software components can be developed to experiment with new operational definitions for TDD behavior;
(ii) it is less dependent on other software components, which can themselves be varied with less change impact;
(iii) it enables plugging in new functionality targeted at assessing operational definitions through experiments and immediate user feedback.

With regard to this last aspect, Besouro also encompasses functionality for on-line user feedback, and persistence of code and assessment data through a single VCS structure. The on-line feedback functionality allows users to become aware of assessed behavior as soon as an episode is processed, and provide their immediate feedback, through (dis)agreement and reassessment. In the case of TDD novices, this feature supports the learning of TDD practices. Additionally, it allows confronting one's perception with the assumed operational definitions. The feedback from experienced practitioners can help evolve the automatic recognition of TDD behavior and the operational definitions assumed. This confrontation can also shed light to different TDD interpretations, common misconceptions, or frequent errors.

Persistence aims to integrate by means of the VCS, all the collected data about episodes and their assessment, with the respective code. Such information can be used in off-line analyses, for purposes such as assessing and improving the system recognition and assessing rules; considering additional data for explanations; searching for hidden programming patterns, among others.

In the remaining of this section, we present Besouro's architecture and summarize its contributions. In Section 5, we describe the prototype we developed to experiment a variation of Zorro's operational definition.

4.1. Besouro architecture

Besouro is available as an Eclipse plugin, and its architecture is depicted in Fig. 2. The architecture is composed of a number of layers, each one with very clear concerns, and which are integrated to capture the same overall flow defined in Zorro.

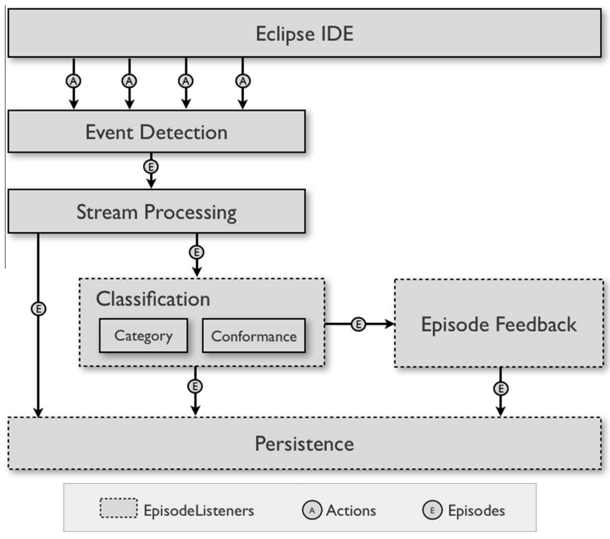

Fig. 2. Besouro architecture.

In the *Event Detection* layer, listeners capture the relevant events for interpreting programmers' behavior in the development and testing environments, together with related metrics necessary to interpret the events. We assume the Eclipse IDE as the target development/testing environment, as depicted in Fig. 2. Listeners are registered using a standard API for Eclipse plug-ins.

The *Stream Processing* layer is responsible for detecting the relevant actions and recognizing the episodes. Besouro collects all the atomic actions directly from the listeners in the Event Detection Layer and organizes them in a chronological sequence of events. As soon as an episode is detected in this layer, the other interested software components can process it.

A distinctive feature in the architecture is the use of *EpisodeListener* components that enable other software components to act upon episodes as soon as relevant events are detected, such as the detection of a new episode or its classification. According to the GoF Observer design pattern, components in the *Classification*, *Episode Feedback* and *Persistence* layers implement EpisodeListeners, registering for notification of events of interest. A particular rationale for this design choice is the ability to display conformance information to users and seek for their immediate feedback.

The *Classification* layer is responsible for mapping actions into observable categories of behavior (Category*Classification*), and assessing their conformance with regard to the aspect of TDD experimented (*ConformanceClassification*). The framework enables to integrate several classification components to act in parallel.

The *Episode Feedback* layer contains the components that present the episode classification for user awareness, and collect his/her feedback. Notice that the feedback should be organized according to the classification aspects assumed, namely, episode category and conformance. It is designed as a specialized EpisodeListener, such that as soon as an episode is classified, it can be displayed to users and their feedback can be input.

Finally, the *Persistence layer* is responsible for logging all this information, associated with the respective code, in a VCS (Git). The adoption of a VCS enables to associate code files with the outcome of episode recognition, classification and feedback components. These results are stored as plain text files, and versioned side by side with the code being produced. EpisodeListeners allow the persistence of information as soon as it becomes available.

4.2. Contributions of Besouro

Zorro, TDDGuide and the system reported in [25] constitute the main inspiration for the features encompassed in Besouro's architecture. As summarized in Table 3, they differ from their respective original propositions as follows:

- The instrumentation of an IDE with the Besouro plugin using only a standard Eclipse API is much simpler, compared to other systems that need a complex configuration (e.g. Zorro/Hackystat client–server architecture, post-processing module in [25];
- Besouro organizes its code according to a modular architecture that makes it easier to vary the instrumentation of the IDE for assessing TDD operational definitions. It highlights different points of variability, namely: (a) the atomic events and respective metrics to be collected from developing and testing environments (Event Detection layer); (b) how atomic events are organized in relevant actions streams (Stream Processing layer); and (c) the classification systems for each set of operational definitions adopted (Classification layer). In addition, it identifies and integrates two new functionalities, persistence and user feedback, which can also be varied according to the goals. The other related systems were not designed with this purpose in mind, and their adaptation for experimenting with a new operational definition requires the complex task of understanding how to change the code, and dealing with the challenges of change management.
- The on-line feedback mechanism of Besouro comprises both awareness and opinion elicitation. In TDDGuide this feature was designed for guiding novices on the TDD practice (e.g. on-track, deviation). In addition to awareness, Besouro feedback mechanism allows to actively confront one's perception with the assumed operational definitions. Feedback is immediate, such that the programmer has in mind his/her recent actions, and can easily compare his/her perception with the result presented by the tool. Unlike TDDGuide, it was designed not to be intrusive, and users can totally ignore it while programming.
- Besouro allows integrating several classifiers to run in parallel, whereas all related systems assume a single operational definition. This enables to experiment and compare variations of the operational definition. The classification component for the Zorro operational definition is fully implemented in Besouro.
- The adoption of a VCS enables combining all information produced by the automatic assessment system, with the respective code files versions, as existing in the VCS. With regard to [25], which combines files and a database to trace actions back to files in the VCS, Besouro uses the VCS as the only persistence

Table 3
Contributions of Besouro with regard to similar systems.

Feature	Besouro	Zorro	TDDGuide	Müller and Höfer [25]
IDE instrumentation	Standard Eclipse plugin	Configuration of Hachystat/Zorro client server architecture	Eclipse plugin	Post-processing stand-alone module
Variability points	Designed for change Separation of concerns Less dependency of external systems	NA	NA	NA
On-line feedback	Novice and practitioners Awareness and feedback elicitation Non-intrusive	NA	Novices (learning) Awareness Intrusive	NA
Episode types and conformance assessment	Customizable operational definition(s) Possibility of multiple, parallel classifiers	Operational definition described in Table 2	According to implemented predicates [24]	According to implemented processing rules [25]
Integration with VCS	VCS is the only persistence structure Assessment data and source code are versioned side by side	NA	NA	Combine data in files and database to trace actions back to code in VCS

NA – not applicable.

structure. Post-processing is not required to relate the two types of data because assessment results are versioned side by side with the respective code.

5. Besouro in action

In this section, we illustrate the usefulness of Besouro's features through a prototype developed to support a simple experiment. The experiments aims at comparing variations of an operation definition based on user feedback, using Zorro's classification rules as baseline. We used Besouro to develop a prototype that: (a) contrasts two different interpretations of episode conformance, (b) compares these interpretations using user feedback, and (c) enables to explore additional code properties to refine episode category classification rules.

More specifically, the prototype was designed to gain more insight on the following aspects of automatic episode classification:

(a) *Context-sensitive conformance:* Zorro's justifies context-sensitive episodes within the scope of comparing test-first and test-last approaches. We claim that, considering other goals with regard to TDD practice analyses, these rules may not lead to representative results, as discussed in Section 3.
(b) *Episode category classification:* in developing experiments with Zorro [34], we realized that we did not agree with the categories in which some episodes were automatically classified. We wanted to explore whether our perception was shared by other TDD practitioners.

The remaining of this section describes how the architecture was instantiated, and the support it has provided as an enabler to a simple experiment.

5.1. Besouro prototype

In this prototype, we propose to explore a variation of Zorro's operational definition, in which deviations or failures are regarded as discrete events that do not affect the conformance of neighbor episodes. To this end, we redefine the conformance of the following episode categories:

– conformant: *test-addition, regression, refactoring*;
– non-conformant: all types of *production* episodes.

We maintain the (non)conformance for the other category types (*test-first*, *long* and *unknown*). Considering the example in Section 3, if the same sequence of events of Fig. 1 is interpreted according to this new set of conformance rules, 90% of the programmer's behavior would be considered as conformant. It is our understanding that this interpretation is the one that best fits the original descriptions of the technique [5].

To this end, we included in the prototype two distinct classification components, such that the different results could be compared to each other, and confronted with users perception. The prototype's feedback interface is displayed in Fig. 3. It was designed such that users were presented with episode classification information (awareness) as soon as an episode is identified and classified. Two feedback elicitation strategies were considered:

- *Active disagreement:* users provide feedback by disagreeing with episode classification. In this case, users input their perception by changing the episode's category, the conformance verdict or both.

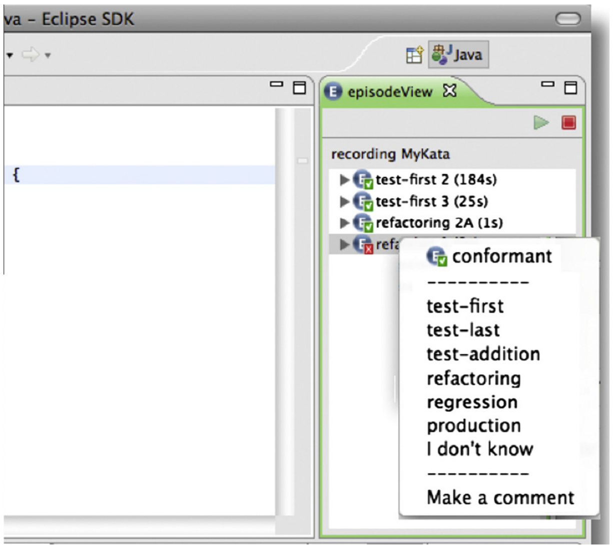

Fig. 3. Besouro feedback interface.

- *Passive agreement:* whenever the programmer agrees with the classification presented, no action is required. We consider this strategy a weaker form of opinion expression, because he/she may be distracted or confused, and may forget to provide the feedback about the automatic classification.

We instantiated Besouro architecture as follows (Fig. 4):

– *Event Detection:* we included three listeners that capture the same events and metrics necessary for Zorro, as described in Section 3. The events captured are open/close files, file editing, changing the program structure, compilation errors and test executions. The metrics are the name and size of files, class name, identification of production class or test case class, and number of methods, assertive and code statements.
– *Stream Processing:* the component *ActionStream* identifies relevant events and detect episodes according to the same set of rules described in Section 3. As mentioned, it captures events and notifies other components (Episode Listeners) when a new episode is recognized.
– *Classification:* we integrated two distinct components, referred to as *BesouroClassification* and *ZorroClassification*. Both assume the same categories of episode, using the same recognition rules, and therefore they share a category classification subcomponent (*ZorroCategory*). However, they apply different

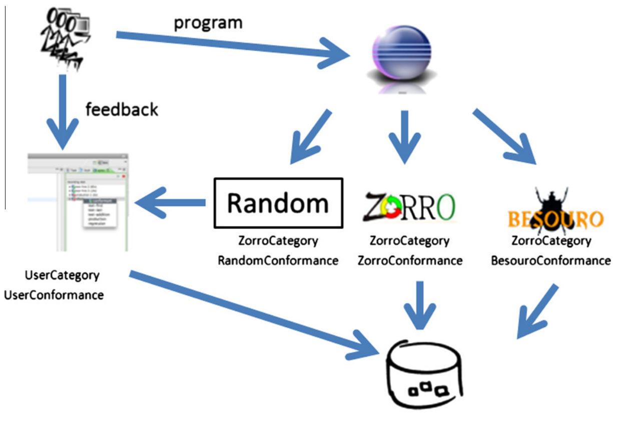

Fig. 4. Comparing classification rules with user perception feedback.

interpretations of conformance for these episode categories, which have been isolated in different subcomponents (*ZorroConformance* and *BesouroConformance*, respectively). In the remaining, we shall refer the user disagreements of have with regard to ZorroConformance and BesouroConformance components, to as Zorro and Besouro disagreements, respectively. ZorroCategory component also records all category outputs produced by Jess as the result of submitting an episode for classification.
– *Episode Feedback*: An Eclipse view shows the episode category and assessed conformance (Fig. 3). Feedback functionality was designed such that the user can disagree of the category type, conformance, or both. In the former case, the feedback is related towards the criteria used for automatic recognition of programming behavior, i.e. how a set of atomic actions is being mapped into a TDD behavior or process. In the later, the feedback is about the criteria used for conformance assessment of that behavior. In addition, the user can register a comment.
– *Persistence*: it stores for each episode the following result files: (a) the raw sequence of events with related metrics; (b) Zorro classification (episode category and conformance); (c) Besouro classification (only conformance, because category is the same as Zorro); and c) programmer feedback. These files are related to the respective code file versions in the VCS.

Since the goal of our experiments was to investigate the user perception of context-sensitive episode conformance, we attempted to avoid bias by displaying in the feedback component a completely random conformance verdict for episodes in the challenged categories (i.e. production, refactoring, regression and test-addition). This strategy is illustrated in Fig. 4. The random conformance verdict may coincide with the verdict yielded by one of the classification components (or both, in case they agree), or be completely distinct. Note that the episode category classification is the same one in both systems: it is only the conformance classification that may vary for context-sensitive types of episodes.

5.2. Experiment support

5.2.1. Experiment description

5.2.1.1. Scope. This experiment aims at exploring the features of Besouro for comparing variations of a TDD operational definition, assuming Zorro as baseline. Table 4 describes the experiment scope according to the GQM (Goal, Question, Metric) approach [35]. Most questions were derived from a previous experiment to evaluate Zorro's rules [34]. This paper puts more emphasis on the support provided by Besouro as an enabler of the experiment, than on the findings with regard to the questions it investigates. Section 5.2.5 discusses threats of validity for this experiment.

5.2.1.2. Context selection. The experiment was developed within the context of the graduate studies of one of the authors. It is a specific and off-line experiment, conducted by developers belonging the Brazilian Agile community.

5.2.1.3. Hypotheses. Considering the three questions described in Table 4, we formulated some hypotheses that we wish to investigate through this experiment, as described below.

The first question ("*Q1: Can the conformance of episode categories production, refactoring, test-addition and regression be established in isolation, regardless the neighbor episodes?*") involves comparing the context-sensitive conformance classification of Zorro with another conformance interpretation. Our hypotheses are that: (a) test-addition, refactoring and regression are always conformant; and (b) production is always non-conformant. Our strategy is to rely on user feedback provided during programming activities to confirm these hypotheses, using metrics M1–M5, described in Table 4.

The second question ("*Q2: Which types of episodes may result in multiple, conflicting category classification?*") is motivated by the category classification component, which relies on the Jess engine. Recall that Jess may output a set of categories for a same episode, based on all rules that are satisfied. The longer the episode, the more results are produced. Sometimes this set of categories is heterogeneous (e.g. {*production*, *production*, *regression*}), and Zorro selects one of these results to define the episode category. We are interested to know how often this happens (M6: number of episodes classified according to multiple, conflicting categories), and the most frequent types of conflict (M7: number of episodes per conflict found).

Finally, the last question ("*Q3: Is code coverage variation a good metric to distinguish between refactoring and production episodes*?") focuses on understanding whether additional criteria could improve the automatic recognition of production and refactoring episodes. To distinguish between them, Zorro uses rules that, in addition to a specific sequence of actions, take into account code size variation (Table 2). Our previous evaluation of Zorro highlighted that code size variation is not enough [34]. Our claim is that code coverage variation can be used as an additional metric to improve the identification of each type of episode. We make the hypotheses that: (a) in *production* episodes, code coverage variation is negative; and (b) in *refactoring* episodes, code coverage variation tends to zero.

Code coverage analysis aims at finding areas of a program that are not exercised by a set of test cases, and determining a quanti-

Table 4
Goal, questions and metrics.

Goal	Questions	Metrics
Analyze ***variations of a TDD operational definition*** For the purpose of ***evaluating*** With respect to ***TDD conformance criteria*** From the point of view of *developers* In the context of ***programming activities***	Q1: Can the conformance of episode categories *production*, *refactoring*, *test-addition* and *regression* be established in isolation, regardless the neighbor episodes?	M1: Number of times users actively disagreed with the presented assessment (category and/or conformance)
		M2: Number of times users actively disagreed with Zorro's assessment (category and/or conformance)
		M3: Number of times users actively disagreed with Besouro's assessment (category and/or conformance)
		M4: Episode conformance assessment according to users feedback (active disagreement and passive agreement)
		M5: Episode conformance assessment according to users active disagreement
	Q2: Which types of episodes may result in multiple, conflicting category classification?	M6: number of episodes classified according to multiple, conflicting categories
		M7: number of episodes per conflict found
	Q3: Is code coverage variation a good metric to distinguish between *refactoring* and *production* episodes?	M8: difference of line/block code coverage, by comparing coverage in the beginning and in the end of an episode

tative measure of code coverage. Code coverage is thus a direct measure of the quality of a suite of tests. Refactoring tends to reduce the code size, but the most important point is that it must not change the functionality. Refactoring can also involve test code, but it should not change the quality of the tests. Hence, code coverage should not vary for production episodes. Production, on the other hand, adds or modifies functionality, which changes the external behavior. It involves only production code: testing code should not be added, nor changed. Hence, code coverage should decrease.

There are many ways of measuring code coverage, ranging from simple line or block coverage (e.g. statements exercised) to condition–decision coverage (e.g. terms in Boolean expressions exercised) [36]. These types of coverage are orthogonal, and their measurements cannot be compared. For instance, 87% of line coverage does not imply the same measure of condition–decision coverage [37]. Our strategy is to start with simple coverage metrics, and if our findings are promising, to extend this analysis to other types of code coverage metrics in future research. Thus, we measured the ratio of covered code with regard to blocks and code lines due to the simplicity of the source code (M8: difference of line/block code coverage, by comparing code coverage in the beginning and in the end of an episode).

5.2.1.4. Selection of subjects and sample characteristics. TDD experience is necessary in this experiment because it aims at evaluating variations of a TDD operational definition. Volunteers from the Brazilian Agile community conducted the experiment. They are experienced programmers, who share a concern for quality. Many of them regularly participate in coding dojos, where they develop *katas* (programming exercises) to improve their skills in agile practices. Twitter and nail lists related to this community were used to find volunteers.

The experiment involved eighteen (18) subjects, who are profiled in Fig. 5 according to their programming experience and TDD experience in industrial settings. Programming experience was classified according to the experience levels defined by Höst et al. [38]. All subjects had at least some level of recent industrial programming experience, and 78% of them were very experienced. The incentive level was defined uniformly for the experiment as I2 (Artificial Project). TDD experience was characterized in terms of self-assessed number of years of TDD practice in the industry. It is possible to see that 83% of our subjects have at least one year of professional experience with TDD, and 44% of them consider themselves as very experienced.

5.2.1.5. Data collection procedure. Data was collected in three stages. First, a face-to-face session took place on November 3rd 2011 with 4 volunteers, when a document containing instructions about the experiment was distributed, and the participants were asked to follow it. One of the authors was present in the meeting, and answered the questions. Based on the questions and the collected data, the instruction document was revised and published in the Internet, together with the Besouro plugin. In the second and third stages, volunteers were asked to develop the experiment based solely on the instruction document [39], and upload the result. These two stages took place on November 15th–December 15th 2011 (6 volunteers), and March 1st–15th 2014 (8 volunteers), respectively.

5.2.1.6. Operation. The revised instruction document [39] was available at the Internet. It explains the tool, how to interact with the feedback mechanism, and what the volunteer is expected to do during the experiment. It also informally describes each category of episode according to standard TDD-related jargon. The study assumes that users should interpret episode categories and their conformance according to their own understanding of this jargon.

Subjects were asked to download and install Besouro, read the instructions of the document, develop at least one kata and to upload the results. The instructions contained a link to a popular list of katas as a suggestion, but participants could develop any kata they like. Participants were asked to develop each kata according to TDD practices, but to deliberately perform some actions that are not compliant to TDD. They should provide their immediate feedback as soon as episode classification information was displayed in the view shown in Fig. 3. Subjects could either passively agree with the proposed classification, or actively disagree with episode category classification, conformance verdict or both. They could also add a comment. A total of 21 katas were developed, which resulted in 360 episodes (Table 5).

We performed a sanity check on the data recorded for each kata. We noticed that there were no disagreements for three katas (katas 6, 7 and 15 in Table 5). When we inquired the respective subjects about the lack of disagreements, we realized that two of them did not fully understand the experiment instructions (the volunteers understood the system was evaluating them, rather than the opposite). These two katas (6 and 7) were developed during the live meeting, and we revised the instructions to make this point very clear. In all analyses that depend on user feedback, the episodes related to these two specific katas were disregarded (38 episodes). The author of the third kata with no disagreements (kata 15) confirmed that he did agree with all assessments presented, and therefore it was considered as valid.

5.2.1.7. Analysis and interpretation. Sections 5.2.2–5.2.4 present the detailed analyses developed for each of one of the three questions in Table 4. Section 5.2.5 discusses the validity threats of the experiment.

5.2.2. Analysis and interpretation: comparing conformance criteria

Metrics M1 to M5 are used to answer Question Q1: "*Can the conformance of episode categories production, refactoring, test-addition and regression be established in isolation, regardless the neighbor episodes?*". Besouro enabled us to collected data regarding these

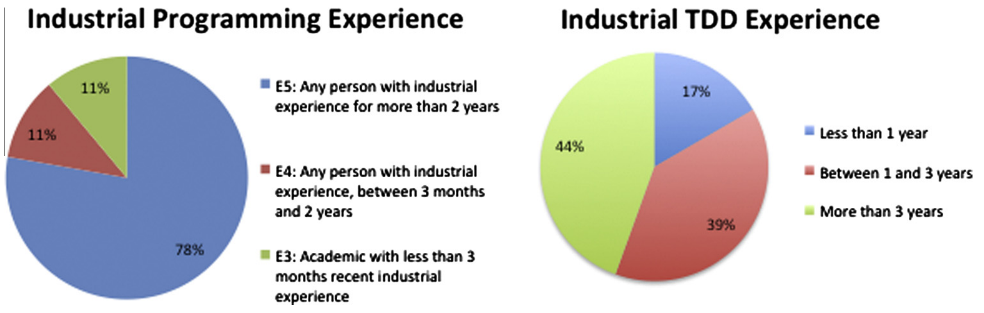

Fig. 5. Participants experience profiling.

Table 5
Kata data and user disagreements.

Kata	Nb. of Episodes	Kata Duration	Episode Average Duration (std dev)	M1: Nb. of Active User Disagreements	M2: Nb (%) of Active Disagreements wrt Zorro	M3: Nb (%) of Active Disagreements wrt Besouro
1	30	71m0s	2m22s (3m41s)	11	9 (30%)	9 (30%)
2	39	94m48s	2m24s (3m49s)	16	5 (12,8%)	5 (12,8%)
3	21	76m58s	4m37s (5m21s)	6	0 (0%)	0 (0%)
4	19	72m0s	4m47s (3m25s)	6	6 (31,6%)	6 (31,6%)
5	17	19m35s	1m6s (1m59s)	10	5 (29,4%)	5 (29,4%)
6	10	14m14s	1m25s (1m24s)	0	0 (0%)	0 (0%)
7	28	76m59s	3m43s (3m1s)	0	0 (0%)	0 (0%)
8	8	44m19s	6m32s (8m52s)	2	2 (25%)	2 (25%)
9	14	64m23s	5m36s (8m56s)	3	3 (21,4%)	3 (21,4%)
10	15	21m3s	1m24s (1m49s)	1	1 (6,7%)	1 (6,7%)
11	13	28m37s	2m7s(1m26s)	2	2 (15,4%)	2 (15,4%)
12	9	64m46s	7m5s(7m40s)	3	3 (33,3%)	2 (22,2%)
13	4	16m57s	4m14s (2m58s)	2	2 (50%)	2 (50%)
14	35	87m20s	2m30s (6m54s)	13	3 (8,6)%)	2 (5,7%)
15	4	13m31s	3m23s (1m8s)	0	0 (0%)	0 (0%)
16	9	12m25s	1m23s (1m22s)	3	3 (33,3%)	2 (22,2%)
17	24	41m21s	2m43s (2m38s)	13	8 (33,3%)	6 (25%)
18	10	41m34s	4m9s (2m25s)	1	1 (10%)	1 (10%)
19	31	52m3s	2m41s (2m4s)	17	9 (29%)	9 (29%)
20	7	21m33s	3m5s (2m37s)	3	2 (28,6%)	2 (28,6%)
21	13	45m42s	3m26s (3m52s)	2	2 (15,4%)	2 (15,4%)
Total	**360**	**16h13m**		**114**	**66 (46%)**	**61 (42,2%)**

metrics by: (a) running two distinct classifiers in parallel; (b) eliciting users' feedback about classification results, and (c) comparing all assessment data persisted in the VCS.

Table 5 presents data about each kata developed, which was retrieved from the VCS. From left to right, the columns represent: (a) kata identifier; (b) number of episodes; (c) total development time; (d) average episode duration (with respective standard deviation); (e) M1: Number of times users actively disagreed with the presented assessment (category, conformance or both); (f) M2: number of disagreements with regard to Zorro's assessment (with respective ratio with regard to the number of episodes of the kata); and (g) M3: number of disagreements with regard to Besouro's assessment (with respective ratio). This set of katas resulted in a total of 360 episodes.

Users have actively disagreed 114 times with the classification information provided by the feedback view (episode category and/or conformance verdict). When compared to disagreements with regard to Besouro and Zorro, the figures are significantly smaller (66 and 61, respectively). This difference is explained by the fact that users disagreed with the random conformant verdict, changing it to a verdict that matches either Zorro's or Besouro's verdict (or both). A detailed analysis on the persisted assessment data showed that most frequently users would disagree with the episode category, with the eventual disagreement regarding the conformance as a consequence. This fact explains why the number of disagreements with regard to Zorro and Besouro are quite similar. Users limited disagreement exclusively to the conformance verdict in 12 episodes only.

The need for better episode category recognition criteria was highlighted in a previous experiment with Zorro [22], which reports 89% of precision. In our experiment based on users' feedback, the precision of episode category classification is 78%. However, these two experiments are not comparable, because ours involved more experienced programmers, who were specifically instructed to question any classification provided by the system.

To collect metrics M4 and M5, which are based on user feedback, we disregarded the 38 episodes of katas 6 and 7. Thus, all analyses dependent on user feedback consider only 322 episodes, of which 154 (49%) are context-sensitive. The two classifiers produced distinct conformance verdicts in 32 of them.

First, we considered these 32 episodes in which Besouro and Zorro classifiers disagreed in the conformance verdict only. Unfortunately, users have actively disagreed about the conformance in only 7 of them, choosing the verdict produced by Besouro as the correct one 6 times. As these disagreements are distributed across the four types of episodes, we are unable to test our hypotheses due to lack of data.

As an alternative, we summarized the conformance perception of the user with regard to these four types of episodes. Table 6 presents a summary of the user feedback as registered in the VCS for all 19 katas with valid user feedbacks, where the user perceived category was adopted as baseline (Episode Category). For each episode category, the table displays: (a) the number of episodes of that category; (b) M4: the number of episodes considered by the user as conformant and non-conformant (regardless whether through active or passive feedback); and (c) M5: the number of episodes considered conformant and non-conformant through active disagreement.

It is possible to see that, in essence, users tend to agree with the claims that production is a non-conformant type of episode, and that test-addition and refactoring are conformant. Indeed, all *production* episodes (14) were considered non-conformant, and in

Table 6
User feedback on context sensitive episodes.

Episode category	Nb. episodes	M4: active/passive feedback		M5: Active disagreement feedback	
		Conformant	Non-conformant	Conformant	Non-conformant
Production	14	0	14	0	5
Test addition	74	63	11	0	1
Refactoring	54	40	14	12	4
Regression	16	10	6	0	1
Total	158	113	45	12	11

Table 7
Comparison of measured conformance adherence.

Kata	Programmer (%)	Zorro (%)	Besouro (%)
10	100	100	100
15	100	100	100
13	100	48.3	48.7
2	99.5	100	99.1
3	99.2	100	100
14	95.8	99.4	94.4
8	95.1	100	98.7
5	93.4	89	77.6
12	88.9	100	89
18	88.8	35.3	63.5
17	85.2	72.9	91.9
21	84.5	100	100
1	79.7	100	88.8
19	78.8	100	100
9	72.8	73.4	74.1
16	62.9	49.3	87
4	55.8	87.8	77.6
11	53.5	87.6	59.5
20	39.6	37.9	70.9

5 occasions, users have explicitly voiced this opinion by disagreement. *Test-addition* was also frequently perceived as conformant (63 out of 74 episodes – 85%), but user feedback was essentially passive for this category. *Refactoring* was considered a conformant category (40 out of 54–74%), and users have actively endorsed as conformant 12 times. However, in 4 occasions, they have explicitly perceived refactoring as non-conformant. *Regression* is the category with the least consensus: by passive feedback, 62% (10 out of 16) of the episodes were perceived as conformant.

In conclusion, the experiment does not enable us to confirm, nor to refute, the hypotheses that conformance can be established independently of context for episodes of type *refactoring*, *production*, *test-addition* and *regression*. However, it does provide clear evidences that Zorro's context-sensitivity conformance criteria need to be further investigated.

An interesting final analysis is the conformance adherence throughout the katas. Conformance adherence was measured by the proportion of time the programmer has spent on conformant practices [22]. Table 7 presents the conformance adherence measure of each kata, according to: (a) the programmer's own perception, as expressed by (dis)agreement; (b) Zorro's verdict; and (c) Besouro's verdict. In general, the conformance measures are high because programmers were asked develop the whole kata according to TDD practices, and eventually introduce non-conformant actions. Table 7 shows that, when users are essentially conformant (user conformance > 85%), Zorro and Besouro tend to present similar behavior, with a slight advantage for Besouro's assessment. On the other hand, when the overall behavior is less conformant, in general it is better assessed by Zorro. In other words, context-sensitive conformance tends to introduce bias towards a more optimistic (test-last) or pessimistic (test-first) interpretation of conformance, whereas Besouro's conformance is focused on how each type of practice fits the overall process. We used Pearson

Table 8
Most frequent types of conflicting category results.

Conflicting categories	M7: number of episodes
Refactoring vs. production	30
Test-first vs. test-last	13
Refactoring vs. regression	3
Test-first vs. production	1
Total	47

correlation to analyze the relationship between the user perception and each tool assessment, and both presented very similar, weak correlations: $r = 0.40$ for Besouro; and $r = 0.41$ for Zorro ($p > 0.05$). Thus, neither of the interpretations is endorsed by feedback, and therefore, additional experiments are required to shed light to conformance interpretation and misconceptions.

5.2.3. Analysis and interpretation: multiple, conflicting episode classifications

Metrics M6 and M7 are used to answer Question Q2: *Which types of episodes may result in multiple, conflicting category classification?*". For collecting this data, we explored Besouro modularity to change the category classifier, and used the episode listeners to persist all results yielded by Jess, before the category classification component makes a particular selection. We developed the analysis below by retrieving the assessment data persisted with each kata in the VCS.

All 21 katas of Table 5 were considered (360 episodes), since this analysis does not depend on the user feedback. The measured M6 = 47, i.e. Jess has output different, conflicting categories for 13% of the episodes. Table 8 displays the number of episodes per conflict (M7). The most frequent conflict involves the difficulty of distinguishing between production and refactoring episodes.

5.2.4. Analysis and interpretation: code coverage variation as a criterion for distinguishing between production and refactoring

Finally, metric Q8 is used to answer *Q3: "Is code coverage variation a good metric to distinguish between refactoring and production episodes*?". The confirmation that production and refactoring are indeed the most difficult types of episodes to distinguish reinforces the importance of this question. This analysis reveals the contribution of integrating a VCS into Besouro framework, since the atomic event sensors were not programmed to collect code coverage data. By versioning assessment data and source code side by side, we were able to look for additional information directly in the respective kata source code and relate it with assessment data. To calculate code coverage for each code version, we used Emma.

Out of the 89 episodes classified in these two categories, 10 were disregarded: 7 were classified by users into a category different from the ones considered here, and 3 received erroneous classification due to known issues of Zorro [22].

The original category of remaining 79 episodes, as produced by Jess rule engine, is distributed as depicted in the left hand side of Fig. 6: *production* (10), *refactoring* (44) and multiple, conflicting cat-

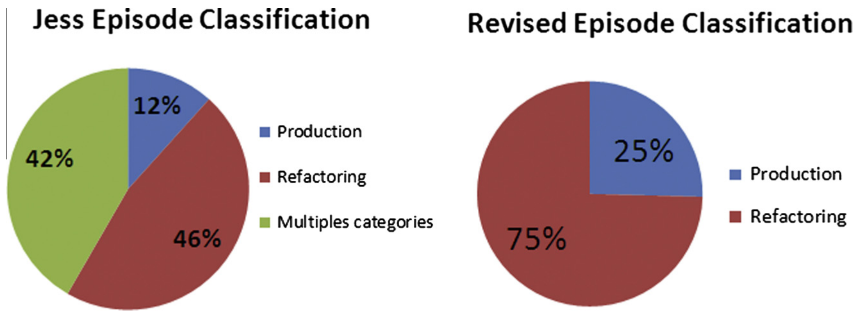

Fig. 6. Episode categories before and after classification.

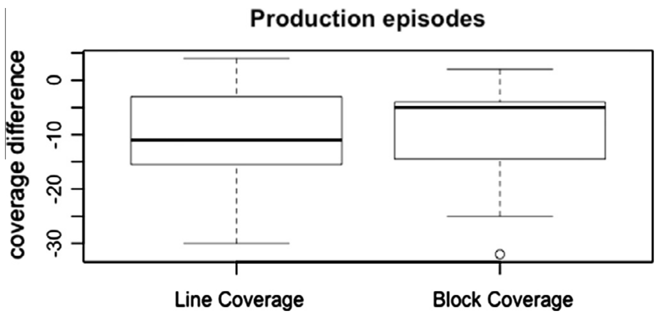

Fig. 7. Code coverage variation for production episodes.

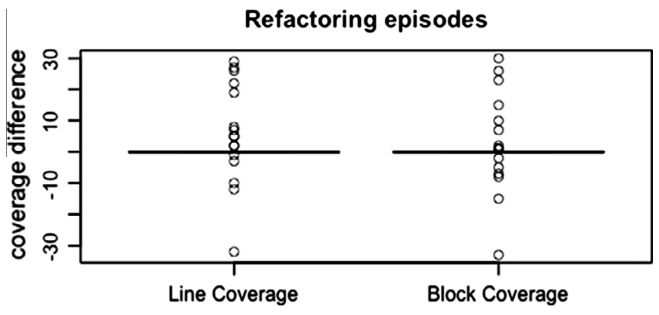

Fig. 8. Code coverage variation for refactoring episodes.

egories (25). Using user active feedback when available, or as a result of a manual inspection of code difference otherwise, the episodes were reclassified as follows (right hand side of Fig. 6):

- 7 out of the 10 *production* episodes were reclassified as *refactoring*;
- 3 out of the 28 *refactoring* episodes were reclassified as *production*;
- out of the 25 episodes classified with multiple, conflicting categories, 10 were classified as *production*, and 15 as *refactoring*;

Fig. 7 displays a boxplot with the variation of code coverage for production episodes, considering both line and block code coverage. We observed a reduction in code coverage (i.e. variation is negative) in 14 out of the 16 production episodes (88%), considering both line and block code coverage. In a single episode code coverage has not changed (variation = 0), and in another episode, it has increased, considering both line and block code coverage.

The boxplot in Fig. 8 shows that code coverage has not changed for 75% of the refactoring episodes (47 out of 63), and therefore code coverage difference is zero for both line and block coverage. Some outliers are observed in Fig. 8: about 16% of the refactoring episodes involved an increase in code coverage, and the remaining 9% presented a negative variation.

In conclusion, there are strong evidences towards our claim that code coverage variation may be used as an additional criterion to improve the automatic classification of refactoring and production episodes. Nevertheless, other code coverage criteria should be explored (e.g. Modified Condition/Decision coverage) and combined to understand how quality is defined and evolves during TDD. Above all, this last analysis revealed the usefulness of integrating code and automatic assessment results for various types of *a posteriori* analyses, to answer questions that rise during experiments.

5.2.5. Threats of validity

We conclude this section by discussing the threats that influences the validity of our experiment. The discussion is based on the four categories of threats for experimental software engineering detailed by Wholin et al. [35].

5.2.5.1. Internal validity. Internal validity concerns how the treatment affects the outcome. *Instrumentation* is a threat, because the documentation was written informally and most subjects performed the experiment completely unsupervised. Of particular concern is that the study is organized around the individual judgment of each subject, which is influenced by his/her personal understanding of TDD underlying techniques. TDD is ill-defined, and the instructions defined the practices to be evaluated in terms of the usual jargon. However, that is the precisely the point in developing systems such as Besouro and Zorro: to gain more insight about TDD practices and misconceptions. Another issue is the design of the on-line mechanism that collects user feedback. Volunteers might be focused on the resolution strategy of the kata, or tired, and therefore, they may forget to provide feedback. The study reduces this risk by giving more value to active user disagreements. Providing random answers as awareness feedback is another means to mitigate this threat, in order to avoid users becoming inattentive when they assume the tool performs correctly. A possible side effect of this design choice is that users might get discouraged to provide feedback due to the low precision of the tool assessment. The elevated number of total disagreements seems to indicate that this side effect was not relevant in this sample. Another instrumentation threat is the premise that Zorro's conformance rules are correctly encompassed in Besouro, despite the significant architectural changes. We mitigate this risk by maintaining the original critical code, and focusing on the modularity and interconnections. Each atomic sensor has also a counterpart in Besouro. Finally, we extensively tested Besouro, and we are confident that it produces the same results, including known errors.

Another internal validity threat is *maturity*, because we cannot guarantee that subjects will not change their interaction with the tool as time passes. They can gain insight on how to assess their own behavior through the feedback provided by the tool. We mitigated this threat through the random conformance verdict feedback, such that users are forced to think about their behavior due to different verdicts for a same practice. *Testing* is a threat that is hard to mitigate, because users might change their resolution strategy even if they solve the same kata, resulting in a different coding sequence. Users also can change their behavior with regard to passive and active feedback, as they alternate the focus from kata problem resolution to providing feedback about their TDD behavior (and vice versa).

5.2.5.2. Conclusion validity. This type of validity is concerned with issues that affect the ability to draw correct conclusions about relations and outcomes. As the goal of the experiment was to explore Besouro's support as an enabler of the experiment, this type of validity threat was not prioritized, despite its ultimate importance. *Low statistical power* is a threat because the sample is small. We did a significant effort to increase this sample, but the experiment design requires experienced TDD practitioners. Volunteers with this profile are not easy to find, as most of them are already overloaded with their own professional activities. The use of novices for increasing the sample would introduce even more risks. So we settled for a small, but high quality sample. The size of the sample justifies not using statistical methods in a more comprehensive and rigorously manner, as any conclusion might lack statistical significance or be due to data overfitting. Another threat comes from the *reliability* of user-related feedback measures, as the background of subjects can affect the study. We mitigated this risk by selecting experienced subjects. The instrumentation can affect the reliability, as already discussed above.

5.2.5.3. Construct validity. It refers to the extent to which the experiment setting actually reflects the construct under study. The lack of formal definition for TDD practice is at the root of this kind of validity. It may have influenced aspects such as conformance interpretation, as well as how subject interpreted they should introduce non-conformant practices during kata development. This risk is not very important as the study aims to explore the perception of practitioners about TDD. Other threats to the construct validity are considered small. The subjects did not know which hypotheses were stated, and were not involved in any discussion concerning the rules for TDD practice recognition and conformance assessment.

5.2.5.4. External validity. Threats to external validity are conditions that limit the ability to generalize the results of the experiment to the industrial practice. This threat was mitigated by: (a) selecting subjects that are representative of the industrial context; (b) developing Besouro as a plugin of a widely accepted IDE (Eclipse); and (c) allowing subjects to develop the kata at their own time and place.

6. Conclusions and future work

In this paper we have introduced Besouro, a compliance assessment framework focused on the development of systems for automatic TDD behavior assessment, which enables to explore and compare distinct TDD operational definitions. Assessing TDD behavior is an initial step towards a very important goal: planning and execution of non-ambiguous quantitative assessment studies involving TDD practice in order to collect concrete evidences about benefits and shortcomings of TDD. Otherwise, TDD supporters and detractors will continue to issue subjective and biased opinions about their own perspectives.

We have presented the architecture of Besouro framework and the design rationale, and described how a prototype was built as an Eclipse plugin. The main features of Besouro are: (a) a modular, lowly coupled architecture allowing distinct variability points concerning events, metrics and classification, as well as persistence-related and view-related functionalities; (b) the easier experimentation with different operational definitions, allowing their comparison; (c) on-line functionality allowing users to provide immediate feedback on their perception of assessed TDD behavior, and (d) persistence of collected data and respective code in an integrated way, making all data related to a traceable behavior available for further analysis. The selection of these features reflects the authors' own personal experiences using and assessing TDD. Besouro integrates them into a single framework, unlike similar systems – mainly the well-known Zorro, in which they are dispersed. Besouro source code is available at [40], and we encourage other researchers to experiment it for developing similar systems.

We illustrated in a real context the usefulness of Besouro's features, by means of a prototype that supports a simple experiment to explore and compares variations of a TDD operational definition. This goal was achieved by: (a) incorporating two alternative classification components in the prototype; (b) using the feedback mechanism to obtain evidences about our hypothesis; and (c) using the information in the VCS for further analysis. The experiment itself was not broad enough in scope, nor rigorous in its design and execution, to confirm or refute our claims. Nevertheless, its suggests evidences that: (a) there is lack of consensus about Zorro‘s conformance rules, particularly when considered in a context other than test-first vs. test-last comparison; (b) production and refactoring activities are hard to distinguish automatically, and additional criteria to distinguish between them are required, and (c) code coverage variation seems to be a promising criteria for improving the recognition of production and refactoring episodes. Exploring other code coverage criteria is a next step in understanding refactoring activities in terms of the relation between production and test code, and how quality evolves during TDD practice.

We regard Besouro as a potential system for conducting quantitative TDD studies, which should not be taken as set in stone, but rather as a starting point for a vivid and creative discussion about alternative approaches and ideas. Interesting questions that rise from our findings include, among others:

(i) Are context-sensitive conformance rules of Zorro (in)adequate for general purposes? Are they necessary and sufficient?
(ii) Are Besouro's proposed variations more representative in general or specific scenarios? Should the notion of non-conformant behavior be reduced taking in account the stricter Besouro's perspective?
(iii) How flexible is Besouro when significantly different TDD operational definitions need to be compared?
(iv) Should a conformance interpretation be oriented towards specific approaches (e.g. test-first, test-last), or towards a generic practice considering a whole process?
(v) Can we gain insight on the aspects of TDD that indeed make the difference in software quality and productivity? To which level one should master techniques that underlie TDD practices, or be conformant, to be successful? Are there specific combinations of practices and contexts that lead to success?
(vi) How do our findings concerning conformance rules generalize to TDD practitioners? What else is lacking to effectively measure the conformance of TDD practices?

(vii) How does TDD will evolve as it is blended to other techniques? For example, Continuous TDD [41] has emerged from the opportunity of exploring continuous testing as a means to shorten the feedback loops between TDD and continuous integration – does it entails a change in TDD practice?
(viii) Are there specific interpretations of TDD according to a community culture (e.g. Brazilian Agile Community)? Is there a generic, universal TDD practice?
(ix) Are we able to define TDD more rigorously, and if so, would it preserve the agile spirit of TDD?

We intend to continue our work in order to find answers to these questions. We hope also that our work will entice more TDD researchers into this fast-growing part of the field, and that it will inspire further discussion.

References

[1] T. Dingsøyr, S. Nerur, V. Balijepally, N.B. Moe, A decade of agile methodologies: towards explaining agile software development, J. Syst. Softw. 85 (6) (2012) 1213–1221.
[2] D. Janzen, H. Saiedian, Test-driven development: concepts, taxonomy and future direction, IEEE Comput. 38 (9) (2005) 43–50.
[3] K. Beck, Extreme Programming Explained, Addison-Wesley, 2000.
[4] K. Beck, Aim Fire, IEEE Softw. 18 (5) (2001) 87–89.
[5] K. Beck, Test-Driven Development: By Example, Pearson Education, Boston, 2003.
[6] H. Erdogmus, G. Melnik, R. Jeffries, Test-driven development, in: Encyclopedia of Software Engineering, 2010, pp. 1211–1229.
[7] F. Shull, G. Melnik, B. Turhan, L. Layman, What do we know about test-driven development?, IEEE Softw 27 (6) (2010) 16–19.
[8] C. Desai, D. Janzen, K. Savage, A survey of evidence for test-driven development in academia, ACM SIGCSE Bull. 40 (2) (2008) 97–101.
[9] H. Munir, M. Moayyed, K. Petersen, Considering rigor and relevance when evaluating test driven development: a systematic review, Inf. Softw. Technol. 56 (4) (2014) 375–394.
[10] B. George, L. Williams, A structured experiment of test-driven development, Inf. Softw. Technol. 46 (5) (2004) 337–342.
[11] M. Pančur, M. Ciglarič, Impact of test-driven development on productivity, code and tests: a controlled experiment, Inf. Softw. Technol. 53 (6) (2011) 557–573.
[12] D.S. Janzen, H. Saiedian, A leveled examination of test-driven development acceptance, in: Proceedings of the 29th International Conference on Software Engineering, IEEE, Minneapolis, 2007, pp. 719–722.
[13] H. Erdogmus, On the effectiveness of test-first approach to programming, IEEE Trans. Softw. Eng. 31 (3) (2005) 226–237.
[14] S. Hammond, D. Umphress, Test driven development: the state of the practice, in: Proceedings of the ACM Southeast Regional Conference 2012, ACM Press, Tuscaloosa, 2012, pp. 158–163.
[15] D.S. Janzen, H. Saiedian, Does test-driven development really improve software design quality?, IEEE Softw 25 (2) (2008) 77–84.
[16] S. Kollanus, Test-driven development – still a promising approach?, in: Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology, IEEE, Porto, 2010, pp 403–408.
[17] S. Kollanus, V. Isomöttönen, Test-driven development in education: experiences with critical viewpoints, in: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, ACM, Madrid, 2008, pp. 124–127.
[18] D.S. Janzen, Test-driven learning in early programming courses, in: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, ACM, Portland, 2008, pp. 532–536.
[19] H. Kou, P.M. Johnson, H. Erdogmus, Operational definition and automated inference of test-driven development with Zorro, Autom. Softw. Eng. 17 (1) (2010) 57–85.
[20] L. Madeyski, Test-Driven Development: An Empirical Evaluation of Agile Practice, Springer, Heidelberg, 2010.
[21] W. Wake, The Test-first Stoplight, 2001. <http://xp123.com/articles/the-test-first-stoplight>.
[22] P.M. Johnson, H. Kou, Automated recognition of test-driven development with Zorro, in: Proceedings of the Agile 2007, IEEE, Washington, DC, 2007, pp. 15–25.
[23] L. Madeyski, L. Szala, The impact of test-driven development on software development productivity—an empirical study, in: R.M.P. Abrahamsson, N. Baddoo, T. Margaria (Eds.), Software Process Improvement, Springer, 2007, pp. 200–211.
[24] O. Mishali, Y. Dubinsky, S. Katz, The TDD-guide training and guidance tool for test-driven development, in: Proceedings of the 9th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP2008), Springer, Limerick, 2008, pp. 63–72.
[25] Matthias M. Müller, Andreas Höfer, The effect of experience on the test-driven development process, Empirical Softw. Eng. 12 (6) (2007) 593–615.
[26] Y. Wang, H. Erdogmus, The role of process measurement in test-driven development, in: Proceedings of the 4th Conference on Extreme Programming and Agile Methods, Springer, Calgary, 2004, pp. 32–42.
[27] R. Oberhauser, Towards automated test practice detection and governance, in: Proceedings of the First International Conference on Advances in System Testing and Validation Lifecycle, IEEE, 2009, pp. 19–24.
[28] L. Vandervert, Operational definitions made simple, useful, and lasting, in: M. Ware, C. Brewer (Eds.), Handbook for Teaching Statistics and Research Methods, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, pp. 132–134.
[29] L. Koskela, Test Driven: Practical TDD and Acceptance TDD for Java Developers, Manning Publications Co., Greenwich, 2008.
[30] C. Desai, D.S. Janzen, J. Clements, Implications of integrating test-driven development into CS1/CS2 curricula, in: Proceedings of the 40th SIGCSE Technical Symposium on Computer Science Education, ACM, Chattanooga, 2007, pp. 148–152.
[31] M.F. Aniche, M.A. Gerosa, Most common mistakes in test-driven development practice. results from an online survey with developers, in: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, IEEE, Paris, 2010, pp. 469–478.
[32] P. Johnson, S. Zhang, We need more coverage, stat! classroom experience with the software ICU, in: Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, 2009, pp. 168–178.
[33] JESS, "Jess rules system." (Online). <http://www.jessrules.com>.
[34] B. Pedroso, R. Jacobi, M. Pimenta, TDD effects: are we measuring the right things? in: Proceedings of the 11th International Conference Agile Processes in Software Engineering and Extreme Programming, Trondheim, 2010, pp. 393–394.
[35] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, et al., Experimentation in Software Engineering, Springer, Heidelberg, 2012.
[36] M.M. Tikir, J.K. Hollingsworth, Efficient instrumentation for code coverage testing, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM, Rome, 2002, pp. 86–96.
[37] E. Bouwers, A. Visser, J. Deursen, Getting what you measure: four common pitfalls in using software metrics for project management, Commun. ACM 55 (7) (2012) 54–59.
[38] M. Höst, C. Wohlin, T. Thelin, Experimental context classification: incentives and experience of subjects, in: Proceeding of the 27th International Conference on Software Engineering, ACM, St. Louis, 2005, pp. 470–478.
[39] Besouro Evaluation Instructions (Online). <http://sites.google.com/site/besouroeval/>.
[40] Besouro Source Code (Online). <https://github.com/brunopedroso/besouro>.
[41] L. Madeyski, M. Kawalerowicz, Continuous test-driven development – a novel agile software development practice and supporting tool, in: Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering, SciTePress, Angers, 2013, pp. 260–267.