Operational definition and automated inference of test-driven development with Zorro

Article *in* Automated Software Engineering · March 2010
DOI: 10.1007/s10515-009-0058-8 · Source: DBLP

3 authors, including:

Hongbing Kou
13 PUBLICATIONS **308** CITATIONS
SEE PROFILE

Hakan Erdogmus
Carnegie Mellon University
130 PUBLICATIONS **1,802** CITATIONS
SEE PROFILE

For submission to: Automated Software Engineering manuscript No.
(will be inserted by the editor)

Operational Definition and Automated Inference of Test-Driven Development with Zorro

Hongbing Kou · Philip M. Johnson · Hakan Erdogmus

January, 2009

Abstract Test-driven development (TDD) is a style of development named for its most visible characteristic: the design and implementation of test cases prior to the implementation of the code required to make them pass. Many claims have been made for TDD: that it can improve implementation as well as design quality, that it can improve productivity, that it results in 100% coverage, and so forth. However, research to validate these claims has yielded mixed and sometimes contradictory results. We believe that at least part of the reason for these results stems from differing interpretations of the TDD development style, along with an inability to determine whether programmers actually follow whatever definition of TDD is in use.

Zorro is a system designed to automatically determine whether a developer is complying with an operational definition of Test-Driven Development (TDD) practices. Automated recognition of TDD can benefit the software development community in a variety of ways, from inquiry into the "true nature" of TDD, to pedagogical aids to support the practice of test-driven development, to support for more rigorous empirical studies on the effectiveness of TDD in both laboratory and real world settings.

Hongbing Kou and Philip M. Johnson
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822
Tel.: 808-956-3489
Fax: 808-956-3548
E-mail: hongbing@hawaii.edu
E-mail: johnson@hawaii.edu

Hakan Erdogmus
Kalemun Research Inc. 4462 Bittersweet Pl.
Ottawa, ON K1V1R9 CANADA
Tel.: 613-822-8589
E-mail: hakan.erdogmus@computer.org

This paper describes the Zorro system, its operational definition of TDD, the analyses made possible by Zorro, two empirical evaluations of the system, and an attempted case study. Our research shows that it is possible to define an operational definition of TDD that is amenable to automated recognition, and illustrates the architectural and design issues that must be addressed in order to do so. Zorro has implications not only for the practice of TDD, but also for software engineering "micro-process" definition and recognition through its parent framework, Software Development Stream Analysis.

Keywords Test Driven Development · Hackystat · Process Measurement

1 Introduction

Substantial claims have been made regarding the effectiveness of test-driven development (TDD). Evangelists claim that it naturally generates 100% coverage, improves refactoring, provides useful executable documentation, produces higher code quality, and reduces defect rates (Beck, 2003). Unfortunately, the empirical research results have been equivocal. Some results are positive: Bhat and Nagappan (2006) found that introducing TDD at Microsoft decreased defect rates significantly in two projects, and Maximilien and Williams (2003) transitioned an IBM development team to TDD with a 50% improvement in quality. But other results are negative: Muller and Hagner (2002) found that TDD resulted in less reliable software than the control group. Yet other results vary regarding the quality and productivity benefits: for exmaple, Erdogmus et al (2005) found that software developed with TDD on average was of no higher quality than software developed by a control group, although a productivity advantage was observed with TDD.

Why might the research results on TDD be so mixed? We believe that part of the reason stems from two methodological issues that impede both progress on understanding TDD's current effectiveness and future improvements to the technique.

First, TDD is often defined in a relatively simplistic and ambiguous manner using toy examples. This can mislead developers into thinking that TDD does not apply to their more complex development situations. It can also lead to different organizations defining the practice of TDD in very different ways.

Second, research on TDD suffers from the "process compliance problem". In other words, the experimental designs do not have mechanisms in place to verify that subjects who are supposed to be using TDD practices are, indeed, using them. The lack of control over process compliance in these experiments means that differences in outcomes may be due, at least in part, to variance in understanding what it means to do TDD, as opposed to differences between the control and experimental groups. If compliance can be measured, meaningful reference points that represent acceptable or idealized patterns can be defined, and deviations, or distance, from these patterns can be correlated with productivity and quality outcomes. Erdogmus et al (2005) stress the importance of gauging process compliance in empirical studies of TDD.

From a pedagogical point of view, Mishali et al (2008) suggest that tool guidance is helpful while learning or mastering TDD. Thus process compliance information, abstracted at the right level and presented unobtrusively, may provide useful feedback to developers and that feedback could allow them better leverage the benefits of TDD.

To address compliance assessment in both laboratory and real-world settings, we believe that the software research and development community needs to agree upon standard, operational definitions of TDD. The definitions must be robust enough to allow for a wide variety of behaviors, rather than strive to enforce a strict, narrowly defined process. If consensus is not possible, at least in research settings it should be clear exactly how TDD is defined and it should be possible to measure objectively to what extent those definitions are adhered to.

In this paper, we present Zorro (Kou, 2007), a system for automated recognition of TDD practices. In essence, Zorro gathers a stream of low-level developer behaviors (such as invoking a unit test, editing production code, invoking a refactoring operation) while programming in an IDE, partitions this event stream into a sequence of development "episodes". Then it applies a rule-based system to determine the type of an episode from a set representing a wide variety of behaviors, whether or not the episode constitutes an instance of TDD practice according to the operational definitions encoded in the rules, and finally the extent of compliance with these definitions through adherence metrics and summary charts. The need for robustness in accomodating variant behaviors is demonstrated empirically in a prior study by Mishali et al (2008). The study evaluates a tool for guiding TDD activities, where many participants express disagreement with the behavior definitions that are too restrictive or narrow. Zorro overcomes this hurdle by providing a fine-grained categorization of possible behaviors and providing adherence metrics that represent a continuum.

Zorro illustrates one approach to addressing the issues that hinder the research and practice of TDD today. Automatic collection and analysis of data make Zorro practical for use in both laboratory and real-world settings: once installed, overhead on the developer with respect to data collection is minimal. Zorro is unobtrusive: it works in the background listening on the events of interest in the environment in which it is installed. It does not require input from the developer. However it incurs a small performance penalty, which depends on the environment.

Second, Zorro can be used to develop a variety of operational definitions of TDD. A Zorro "TDD definition" consists of the set of developer behaviors that must be recorded, the manner in which this timestamped stream of events are partitioned into episodes, and the rules used to classify an episode according to TDD terminology and determine whether the episode is compliant with idealized TDD patterns. By providing a way to define an operational definition of TDD, Zorro addresses the compliance problem by enabling researchers and practitioners to precisely characterize the extent to which the given definition of TDD was applied (or not) in any given development scenario. Furthermore,

Zorro's episode-based approach to TDD recognition provides a more nuanced approach to characterizing the use of TDD by developers: rather than a binary, "all-or-nothing" approach, Zorro enables TDD usage characterizations based on percentages, such as "Developer A used TDD 73% of the time".

Zorro has undergone initial empirical evaluation through classroom and industry-based case studies. The results from classroom studies indicate that Zorro is a viable approach to automated TDD recognition. The industrial case study was inconclusive.

This paper is organized as follows. The next section briefly introduces the practice of TDD. Section 3 presents work related to Zorro. Section 4 presents the architecture and implementation of Zorro with examples. Section 5 presents two empirical evaluation experiments we performed to validate the Zorro inference mechanism and gain insight into its strengths and weaknesses, as well as an attempted industrial case study. Section 6 summarizes the contributions and future directions for this research.

2 The practice of TDD

Test-Driven Development (Beck, 2003) is a software development best practice popularized by Extreme Programming (Jeffries, 2000; Beck, 2000). It is normally introduced as a very simple practice consisting of only two rules:

1. Write new code only if an automated test has failed.
2. Eliminate duplication.

An equally simple but more process-oriented description is the the "stop light" metaphor (Beck, 2003):

1. Red - Write a little test that does not work, and perhaps does not even compile at first.
2. Green - Make the test work quickly, committing whatever sins are necessary in the process.
3. Yellow (Refactor) - Eliminate all the duplication created by merely getting the test to work.

Important characteristics of TDD practice, according to Beck (2001, 2003), include the following.

Write the test first. This is the key characteristic of TDD. Some developers using TDD advocate that if you are using TDD, you should "never write a line of production code without a broken test case." Although adherence to this extent is not always practiced, the principle captures the essence of TDD.

Short iterations. Quickly adding production code to make test pass is important to TDD. An iteration should last a few seconds to several minutes only. If hours of work are needed to make a test pass, then this is a sign that the developer should have divided the programming task into smaller subtasks that could be solved in a shorter period of time.

Frequent refactoring. Code is consistently refactored in TDD to create the simplest possible design. The existence of a suite of unit tests gives developers the "courage" to refactor the code

Rapid feedback. Unit testing is usually supported by the XUnit framework that is now available for most languages. After new production code is added, developers should invoke the unit tests to test it right away. This feedback should be available within seconds or minutes.

One ball in the air at a time. In typical software development, the developer tries to simultaneously balance several requirements, including system structure design, algorithm choice, code efficiency, readability, communication with other code, and so forth. Martin Fowler is quoted as describing that process as being like "keeping several balls in the air at once". In contrast, it is claimed that in TDD, the developer only keeps "one ball in the air at once" and concentrates only on that ball. For example, in the development step, the developer only needs to make the test pass without worrying about whether it is a good or bad design. In the refactoring step, the developer only worries about what makes a good design.

The code should always work. In TDD, developers should run all tests at the end of each iteration. If any test has failed, the developer should fix it right away. The fix should be easy because only a small amount of code is written in each iteration. If running all tests after an iteration is not feasible, the continuous integration can be set up to run them all once a day or several times a day.

3 Related work

3.1 Claims about TDD

TDD advocates claim that adherence to this approach can simultaneously improve both quality and productivity (Beck, 2001; Janzen and Saiedian, 2005). Because software quality is sometimes hard to quantify in a universal manner, TDD practitioners and researchers often use code coverage as a proxy or precursor for software quality. The code developed in TDD should have very high coverage since in theory no production code is created without a corresponding unit test. Some advocates encourage 100% coverage, however this level may be hard or impossible to achieve in practice depending on the coverage criterion (method coverage is easy to achieve at 100%, but near perfect branch and path coverage are much more elusive).

The claimed rationale for why TDD improves productivity is a bit more complicated and controversial. Production code and test code are both software. Writing tests in advance is considered as part of the design process, and it often does not take a long time to write a small test once certain low-level design decisions are made. If developers need to write same amount of test code, TDD should save development time because less time is spent on tests than in the traditional test-last or ad-hoc development methods. Tests also

make progress visible, which may in turn affect motivation. Small tests written incrementally encourage finer task decomposition and better task focus. Regressing tests frequently allows production errors to be discovered early, thereby reducing costly snowball effects. In addition, TDD users claim that the method reduces the overall amount of time spent on debugging, rework and bug fixes following post-release failure, with a resulting increase in overall long-term productivity (Williams et al, 2003).

3.2 Empirical evaluation of TDD

Much research has been conducted on studying outcomes of TDD such as software quality and developer productivity in recent years. In addition to case studies and anecdotal experience reports (George and Williams, 2004; Maximilien and Williams, 2003; Williams et al, 2003; Kaufmann and Janzen, 2003; Edwards, 2004; Bhat and Nagappan, 2006; Damm and Lundberg, 2006; Sanchez et al, 2007; Janzen and Saiedian, 2008), researchers have run controlled and quasi-controlled experiments (Muller and Hagner, 2002; Pančur et al, 2003; Erdogmus et al, 2005; Janzen and Saiedian, 2008; Madeyski and Szala, 2007; Siniaalto and Abrahamsson, 2007; Gupta and Jalote, 2007) to compare TDD against other development methods such as test last and ad hoc. We categorize the research as "academic" or "industrial" depending upon whether the study subjects were students or professional developers.

3.2.1 Empirical evaluation in academic settings

Muller and Hagner (2002) conducted a study in an XP class in Germany to test TDD against traditional programming. The acceptance tests were provided to both the TDD group and the control group. Interestingly, students in the TDD group spent more time but their programs were less reliable than the control group.

Edwards (2004) adopted TDD in a junior-level class to compare whether students got more reliable code after the use of TDD and WEB-CAT, an assignment submission system. It turned out that the students using TDD reduced their defect rate dramatically (45% fewer defects/KSLOC using a proxy metric) after adopting TDD, and a posttest survey found that TDD students were more confident of the correctness and robustness of their programs.

Similarly, Kaufmann and Janzen (2003) conducted a pilot study on implications of TDD in an advanced project-oriented software engineering course. They also reported that TDD helped to improve software quality and programmers' confidence.

Pančur et al (2003) designed a controlled experiment involving 38 students to compare TDD with Iterative Test-Last approach (ITL), which is a slightly modified TDD development process in the order of "code-test-refactor". This study found no notable difference betweewn the two approaches.

Erdogmus et al (2005) used the well-defined test-last and TDD approaches as in Pančur et al (2003) to study the effectiveness of TDD through a controlled experiment. This study concluded that TDD programmers wrote more tests per unit of programming effort than test-last programmers. They found that test code tends to increase minimum software quality, but there was no difference between the average quality of the programs produced by the TDD and test-last groups. The TDD group achieved overall better productivity, although the difference was not statistically significant.

Madeyski and Szala (2007) conducted a single-subject experiment in which the subject used successively TDD and traditional development in a Java/AspectJ project to build a web-based conference management system. The subject spent 112 hours to develop the system. Subject's productivity initially improved 87 to 177% with TDD, but when TDD was withdrawn, productivity did not regress to its previous levels.

Siniaalto and Abrahamsson (2007) conducted an experiment with 13 students. The students had prior development experience in the industry. They were asked to develop a small mobicle stock market browser application in Java. The researchers evaluated the effect of TDD based on test coverage and design quality. Test coverage was improved with TDD. As for design quality, cohesion appeared to improve, but the result was weak. The effect on coupling was inconclusive.

The experiment by Gupta and Jalote (2007) involved 22 students. The subjects developed small registration and ATM applications using Java, spending 20-55 hours to complete the task. The researchers observed that overall productivity was improved with TDD, but quality results were inconclusive.

Janzen and Saiedian (2008)'s two experiments with graduate and undergraduate students yielded consistent results on software quality. One experiment used two teams of three students with 0-5 years of prior development experience and the other used a single team of three novices. The subjects developed programs of 800 to 1300 lines of Java code. TDD improved test coverage and resulted in programs with smaller modules, methods, and methods per class. The complexity of the programs measured by two different metrics were also higher. The difference was more dramatic so with the undergraduate students. Coupling and cohesion results were inconclusive.

3.2.2 Empirical evaluation in industrial settings

Several attempts have been made by researchers to study software quality and developer productivity improvements of TDD in industrial settings.

George and Williams (2004) ran a set of structured experiments with 24 professional pair programmers in three companies. Each pair was randomly assigned to a TDD group or a control group to develop a bowling game application. The final projects were assessed at the end of the experiment. They found that TDD practice appears to yield code with superior external code quality as measured by a set of blackbox test cases, and TDD group passed

18% more test cases. However, the TDD group spent 16% more time on development, which could have indicated that achieving higher quality requires some additional investment of time. Interestingly, and in the contrast to the empirical findings, 78% of the subjects indicated that TDD practice would improve programmers' productivity.

Maximilien and Williams (2003) transitioned a software team from an ad-hoc approach to testing to TDD unit testing practice at IBM, and this team improved software quality by 50% as measured by Functional Verification Tests (FVT).

Williams et al (2003) conducted another case study in IBM to study TDD. Compared to a baseline project developed in a traditional fashion, the defect density of the project developed in TDD was reduced by 40% as measured by functional verification and regression tests. The productivity was not impacted by the additional focus on producing test code.

Geras et al (2004) isolated TDD from other XP practices, and investigated the impact of TDD on developer productivity and software quality. In their research, TDD does not require more time but developers in TDD group wrote more tests and executed them more frequently, which may have led to future time savings on debugging and development.

Another study of TDD at Microsoft (Bhat and Nagappan, 2006) reported remarkable software quality improvement as measured in number of defects per KLOC. After introduction of TDD, project A (Windows) reduced its defects rate by 2.6 times, and project B (MSN) reduced its defect rate by 4.2 times, compared to the organizational average. Reportedly, developers in project A spent 35% more development time, and developers in project B spent 15% more development time, than the developers in non-TDD projects spent.

Damm and Lundberg (2006) conducted longitudinal case studies lasting up to 1.5 years with 100 professionals at Ericsson. The professionals developed components for mobile applications using C++ and Java. The researchers observed that TDD reduced total project costs by 5-6%, fault slip-through by 5-30%, and avoidable fault costs by 55%.

Sanchez et al (2007)'s longer, single case study at IBM lasted 5 years and involved 9-17 developers working on a medium-size device driver with legacy components. Over this period of time, the researchers noted a 19% increase in development effort with TDD in return for a 40% increase in internal defect rates.

Janzen and Saiedian (2008)'s suite of three industrial experiments and one case study involved overlapping teams and individuals of three subjects. The studies yielded moderately favorable results for TDD for test coverage and some size metrics, but were not consistent for complexity, coupling and cohesion measures. In all of the studies, the participants developed real-world J2EE applications ranging from 800 to 50,000 lines of code. Test coverage was improved with TDD in all studies but one. TDD also consistently resulted in smaller modules and methods per class (the latter except in one study), but not necessarily in smaller methods.

3.3 A comparative analysis of TDD studies

The research findings regarding the effect of TDD practices on software quality and developer productivity are mixed, as shown in Tables 1 (Controlled Studies) and 2 (Case Studies). The study conducted at Microsoft (Bhat and Nagappan, 2006) and the study conducted at the University of Karlsruhe (Muller and Hagner, 2002) are two extreme cases. In Bhat and Nagappan (2006), the developers improved software quality up to four times after adopting TDD. In comparison, the TDD group in Muller and Hagner (2002) yielded less reliable programs than the control group.

Table 1 Controlled and Quasi-Controlled Empirical Experiments on TDD

Investigator	A/I	Subjects	Software Quality	Developer Productivity
Janzen and Saiedian (2008)	I	teams of 1-3	TDD had better coverage and smaller modules	N/A
Janzen and Saiedian (2008)	A	1-2 teams of 3	TDD had better coverage, smaller methods and modules, and less complexity	N/A
Madeyski and Szala (2007)	A	1	N/A	TDD had 87-177% better productivity initially
Siniaalto and Abrahamsson (2007)	A	13	TDD improved coverage	N/A
Gupta and Jalote (2007)	A	22	Inconclusive	Improved overall productivity
George and Williams (2004)	I	24	TDD improved test coverage, possibly reduced cohesion	N/A
Geras et al (2004)	I	14	TDD had better quality	No impact
Kaufmann and Janzen (2003)	A	8	N/A	50% improvement
Erdogmus et al (2005)	A	35	No change	Improved productivity
Muller and Hagner (2002)	A	19	Less reliable, but better reuse	No change
Pančur et al (2003)	A	38	No change	No change

3.4 Research on automated inference of software process

Automated software process research systems are often top-down in nature: they take a high-level description of a system process and use it to constrain, control, or understand the actual development behaviors. Process programming (Sutton et al, 1995), modeling (Bill Curtis and Over, 1992) and simula-

Table 2 Empirical Case Studies on TDD

Investigator	A/I	Subjects	Software Quality	Developer Productivity
Janzen and Saiedian (2008)	I	team of 3	TDD had better coverage and smaller methods and modules	N/A
Sanchez et al (2007)	I	9-17	30% reduction in defect density	Increased effort 19%
Damm and Lundberg (2006)	I	100	5-30% reduction in fault slip-through, 55% reduction in fault costs	Project cost increased by 5-6%
Maximilien and Williams (2003)	I	9	50% reduction in defect density	Minimal impact
Williams et al (2003)	I	9	40% reduction in defect density	No change
Bhat and Nagappan (2006)	A	11	2-4 times reduction in defect density	35% and 15% more time
Edwards (2004)	A	59	54% fewer defects	N/A

tion (Turnu et al, 2004; Jensen and Scacchi, 2005) are typical research methods for studying software processes. Process conformance, at least as performed by Zorro, is a bottom-up process in which the actual development behaviors are the input to, as opposed to the output from, the system.

Cook and Wolf (Cook and Wolf, 1995; Cook, 1996) developed a client-server system named Balboa to automate the process discovery using finite state machine (FSM). Balboa collects developers' invocations of Unix commands and CVS commits to construct event streams. It then uses a neural network, a MARKOV chain, and data mining algorithms to discover the FSM of software processes. With Balboa, Cook and Wolf were able to reproduce the ISPW 6/7 process (Kellner et al, 1991) in their research.

Jensen and Scacchi (2004, 2005) simulated an automated approach to discover and model the open source software development processes. They took advantage of prior knowledge to discover the software development processes by modeling the process fragments using a PML description. Their prototype simulation found that they could detect unusually long activities and problematic cycles of activities. They suggested that a bottom-up strategy, together with a top-down process meta-modeling is suitable for automated process discovery.

For this research, we chose a rule-based system to study process conformance of low-level software processes. Instead of asking experts to inspect the FSM of the executed process as in Cook and Wolf (1995), we converted the process knowledge into a set of rules and used them to infer the software development behaviors. Our method is very close to Jensen and Scacchi (2004) except that we used rules rather than PML for process descriptions.

Wang and Erdogmus (2004) argued that empirical research on TDD suffers from the construct validity problem (as is also the case in some other empirical

software engineering research) because the experimental designs lack mechanisms to verify process conformance. They developed a prototype called "TestFirstGauge" to study process conformance in TDD by mining the in-process log data collected by Hackystat (Johnson et al, 2005; Johnson and Paulding, 2005; Johnson et al, 2004).

TestFirstGauge aggregates software development data collected by Hackystat to derive programming cycles of TDD. They used T/P ratio (lines of test code verse lines of production code), testing effort against production effort, and cycle time distribution as indicators of TDD process conformance. This project precedes the Zorro software system (Kou and Johnson, 2006), and in fact it stimulated our interest in studying low-level software process conformance. Unlike the prototype implementation of TestFirstGauge, which uses an Excel spreadsheet, Zorro is integrated into the Hackystat system and uses the JESS rule-based system (Friedman-Hill, 2003).

Similarly, Wege (2004) also focused on automated support of TDD process assessment, but his work has a limitation in that it uses the CVS history of code. Developers typically do not commit on-going project data at the granularity of seconds, minutes or hours, making this data collection technique problematic for the purpose of TDD inference. At least for the operational definition of TDD developed in this research, collecting rapid low-level development activities is crucial to correctly identifying its occurrence.

In their evaluation of TDD vs. test-last development, Madeyski and Szala (2007) used a plugin for Eclipse in order to assess the subjects' compliance with the assigned development processes. Their plug-in is similar to TestFirstGauge and Zorro in that it records low-level developer activity data from the IDE, and caches the data in the local project directory. Then it makes automated commits to the project repository. Also like Zorro, this tool offers basic off-line reports to developers as well as to the organizations for assessing activities involved in each commit. The plugin appears to achieve Zorro's and TestFirstGauge's reporting capabilities locally, without connecting to a HackyStat server. Development of this tool appears to have been discontinued, and further details about it are unavailable.

TDDGuide by Mishali et al (2008) is another process compliance tool that is integrated into the Eclipse IDE. The main motivation behind TDDGuide is not assessment, but guidance, particularly when learning TDD. Unlike Zorro and TestFirstGauge, TDDGuide operates in real time to instantly detect and report compliant and non-compliant patterns to respectively encourage and discourage the developer. It also has a logging capability for off-line analysis. Similar to Zorro, TDDGuide is rule-based, however it operates at a higher level of granularity than Zorro, hence its classification scheme for identifying patterns is coarser and less diverse than Zorro's. The tool reports any deviations from pre-defined TDD patterns to the developer and proposes activities for conformance. In empirical evaluations, users reported that TDDGuide helped them adhere to TDD behaviors, however nearly half of them found the tool to be intrusive. TDDGuide, unlike Zorro and TestFirstGauge, does not produce

adherence metrics or telemetry information for visualizing developers' patterns over time in order to understand trends in TDD behavior.

4 Zorro

4.1 Architectural components

As illustrated in Figure 1, the Zorro architecture consists of three subsystems: (1) Hackystat, which collects low-level developer behaviors; (2) SDSA (Software Development Stream Analysis), a Hackystat application that supports generic analysis of development event streams; and (3) Zorro, an SDSA application, which defines the specific rules and analyses necessary for recognition and interpretation of the TDD behavior of a developer.

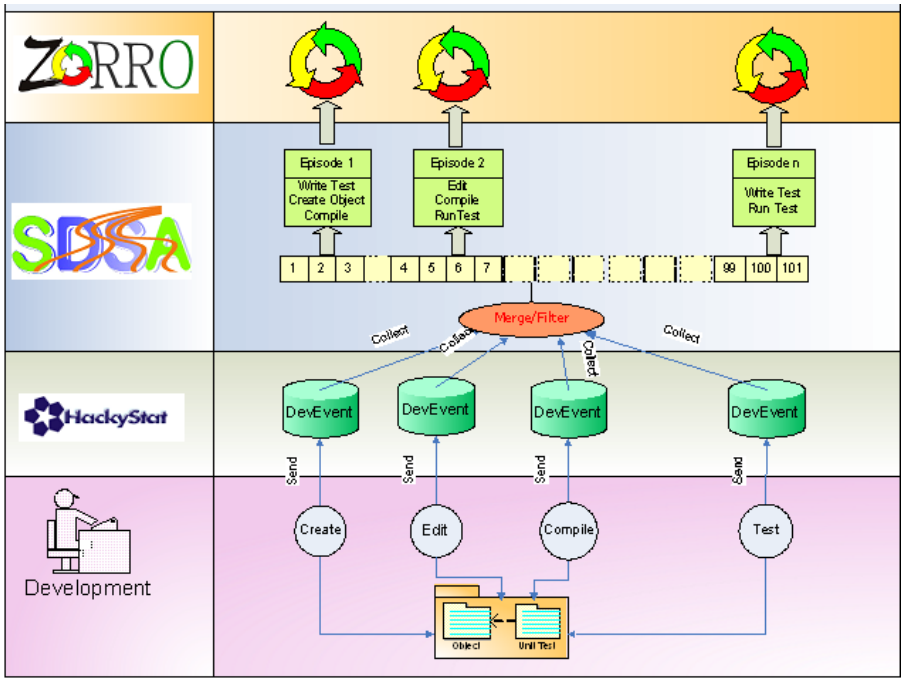


Fig. 1 The Zorro Architecture

Hackystat. Hackystat (Johnson et al, 2005; Johnson and Paulding, 2005; Johnson et al, 2004) is an open source framework for automated collection and analysis of software engineering process and product data that we have been developing since 2001. Hackystat supports unobtrusive data collection via specialized "sensors" that are attached to development environment tools and that send structured "sensor data type" instances to a web-based Hackystat service for subsequent analysis. Over two dozen sensors are currently available,

including sensors for IDEs (Emacs, Eclipse, Vim, VisualStudio, Idea), configuration management (CVS, Subversion), bug tracking (Jira, Bugzilla), testing and coverage (JUnit, CppUnit, Emma, JBlanket), system builds and packaging (Ant), static analysis (Checkstyle, PMD, FindBugs, LOCC, SCLC), and so forth. Applications of the Hackystat Framework in addition to our work on SDSA and Zorro include in-process project management (Johnson et al, 2005), high performance computing (Johnson and Paulding, 2005), and software engineering education (Johnson et al, 2004).

Zorro requires the developer's IDE to be instrumented with a Hackystat sensor that can collect at least the following kinds of events: unit test invocations (and their results), compilation events (and their results), refactoring events (such as renaming, moving), and editing (or code production) events (such as whether the file has changed in state during the previous 30 seconds, and what the resulting size of the file is in statements, methods, and/or test case assertions).

We have implemented a Hackystat sensor for the Eclipse IDE to collect these events for the Java language, and a Hackystat sensor for the Visual Studio IDE to collect these events for the C# language.

SDSA. Software Development Stream Analysis (SDSA) is a Hackystat-based application that provides a generic framework for organizing and analyzing the various kinds of data received by Hackystat as input to a rule-based, time-series analysis.

SDSA begins by merging the events collected by various sensors into a single sequence, ordered by time-stamp, called the "development stream". This is followed by a process called tokenizing, which results in a sequence of higher-level "episodes". These constitute the atomic building blocks for whatever process is being recognized. For any given application of the SDSA framework, tokenization involving defining the specific events to be combined to generate the development stream, as well as the boundary condition that separates the final event in one episode from the initial event in the next. For example, development events could include things like a unit test invocation, a file compilation, a configuration management commit, or a refactoring operation. Example boundary conditions could include a configuration management system checkin, test pass event, or a buffer transition.

Once the development stream has been abstracted into a sequence of episodes, the next step in SDSA is to classify each episode according to whatever process is under analysis. SDSA provides an interface to the JESS rule-based system engine to enable developers to specify part or all of the classification process as a set of rules.

Zorro. The Zorro architectural layer provides extensions to Hackystat and SDSA necessary for the automated recognition of Test Driven Development behaviors. Let's now examine Zorro's inferencing mechanism in more detail.

4.2 TDD Inference using Zorro: A simple example

As introduced above, TDD inference in Zorro consists of the following steps: (a) collection of low-level developer sensor data using Hackystat; (b) abstraction of the sensor data into a developer event stream; (c) partitioning of the event stream into episodes; and finally (d) classification of the resulting episodes as either TDD-conformant or TDD non-conformant. Figure 2 illustrates an example of the last three steps in this process.

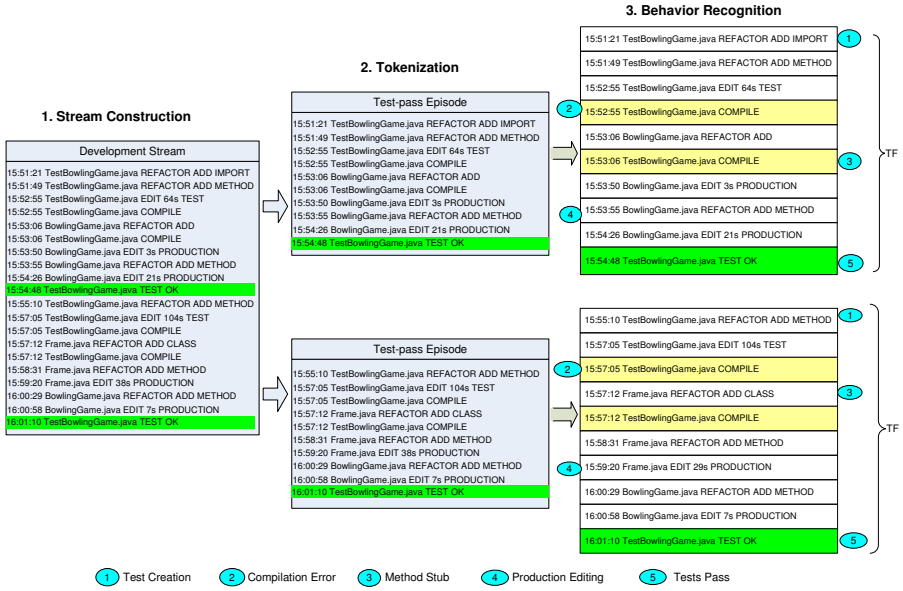


Fig. 2 Collecting, partitioning, and classifying developer behaviors

From 15:51:21 to 16:01:10, a developer implemented two user stories of an application for calculating the players' scores in a bowling game applying Test-Driven Development. We used Hackystat to instrument the development process and collect sensor data regarding refactoring, editing, compilation and test invocation activities. The "Stream Construction" phase results in the consolidation of the raw Hackystat sensor data into a sequence of 21 elementary developer actions. Each action includes a timestamp (such as 15:51:21), a file (such as "TestBowlingGame.java"), and an action type (such as "EDIT 64s TEST").

The actions are also augmented with static analysis metrics associated with the file on which the action was performed. The metrics contain information on production classes, test classes, production methods, test methods, assertions within test methods, statements within methods, and modified, added and deleted lines of code. The metrics help in determining the type of an elementary action.

For example, if an action increases the number of test methods, the number of assertions, and the size of a file containing a test class, the action is labelled as a test creation action. This way, Zorro can identify superfluous actions and discount them: for example if a new test has been added, but the number of assertions has not increased, the new test is "empty" and would not be counted as a test creation action.

Among the 21 developer actions in this example are two successful test invocation actions ("TEST OK"). Zorro partitions the stream of developer actions into episodes based upon the occurrence of a successful test invocation action, so this sequence of 21 actions is partitioned into two episodes, both ending with a successful test invocation action. Thus, successful test invocations delimit episodes composed of elmentary actions.

The next step is to determine which, if any, of these two episodes corresponds to a valid TDD development practice. To do this, Zorro applies a rule-based recognition system. In this example, both episodes contain the following sequence of actions: (a) a test method is created; (b) a compilation error results; (c) a method stub is created in production code which results in a successful compile; (d) more production code is edited; and (e) all tests pass. These actions correspond to the classic style of TDD development, and Zorro's rules will classify both of these episodes as instances of TDD, or as valid TDD behavior. Certain combinations of actions are indicative of non-TDD behavior, and similarly these can be recognized as such. For example in the episode on the bottom right of Figure 2, if test creation actions (1) follow production editing actions (4) rather than precede them, the episode would have been recognized as non-conformant to TDD.

In the case where a developer follows the canonical TDD approach involving a few episode types, identifying TDD behavior is relatively easy. The more important issue is how to deal with the complexities of real world software development behaviors. An example of variant TDD behavior that is difficult to recognize is test addition followed by successful test invocation, but with no production code editing. This behavior may happen both in the context of applying TDD and outside TDD. To be able to recognize it as TDD behavior, Zorro looks for neigboring episodes that are easily identified as typical TDD. If such an ambiguous episode occurs in the context of other episodes easily identified as TDD, they are considered to be part of TDD behavior. Otherwise, they are classified as non-TDD-conformant. Thus Zorro is capable of context-sensitive episode classification.

In the next section, we explain how Zorro handles more diverse and ambiguous episode types to recognize a wide range of behaviors.

4.3 Episode classification

The heart of Zorro is its episode classification algorithm, implemented as a set of 32 JESS rules along with additional templates and classifier definitions. JESS rules are applied on a raw development stream recorded by the Hackystat

sensor installed in the developer environment to infer elementary action types first. Once action types are assigned, higher level rules are used to infer episode types from streams of actions and classify episodes that can be identified as TDD-conformant or not in isolation. Finally, additional rules are applied to classify uncategorized episodes from the episode stream in which they occur using contextual information. Zorro is open source and the rules can be found online at http://hackystat-analysis-sdsa.googlecode.com/.

Figure 3 summarizes the effect of these rules, which is to classify any episode as belonging to one of 22 episode types.

ID	Definition	TDD Conformant
Test First		
TF-1	Test creation -> Test compilation error -> Code editing -> Test failure -> Code editing -> Test pass	Yes
TF-2	Test creation -> Test compilation error -> Code editing -> Test pass	Yes
TF-3	Test creation -> Code editing -> Test failure -> Code editing -> Test pass	Yes
TF-4	Test creation -> Code editing -> Test pass	Yes
Refactoring		
RF-1	Test editing -> Test pass	Context sensitive
RF-2	Test refactoring operation -> Test pass	Context sensitive
RF-3	Code editing (number of methods or statements decrease) -> Test pass	Context sensitive
RF-4	Code refactoring operation -> Test pass	Context sensitive
RF-5	[Test Editing && Code editing (number of methods or statements decrease)]+ -> Test pass	Context sensitive
Test Addition		
TA-1	Test creation -> Test pass	Context sensitive
TA-2	Test creation -> Test failure -> Test editing -> Test pass	Context sensitive
Regression		
RG-1	Non-editing activities -> Test pass	Context sensitive
RG-2	Test failure -> Non-editing activities -> Test pass	Context sensitive
Code Production		
CP-1	Code editing (number methods unchanged, statements increase) -> Test pass	Context sensitive
CP-2	Code editing (number methods /statements increase slightly (source code size increase <= 100 bytes) -> Test pass	Context sensitive
CP-3	Code editing (number methods /statements increase significantly (source code size increase > 100 bytes) -> Test pass	No
Test Last		
TL-1	Code editing -> Test editing -> Test pass	No
TL-2	Code editing -> Test editing -> Test failure -> Test pass	No
Long		
LN-1	Episode with many activities (> 200) -> Test pass	No
LN-2	Episode with a long duration (> 30 minutes) -> Test pass	No
Unknown		
UN-1	None of the above -> Test pass	No
UN-2	None of the above	No

Fig. 3 Zorro episode types, definitions, and TDD conformance

Zorro organizes the 22 episode types into eight categories: Test First (TF), Refactoring (RF), Test Last (TL), Test Addition (TA), Regression (RG), Code Production (CP), Long (LN), and Unknown (UN). All of these episode types (except UN-2) always ends with a "Test pass" event, since that is the episode boundary condition. (UN-2 is provided as a way to classify a development session where there is no unit testing at all.)

The definition column only gives a typical instance for each episode category. The implementation allows repetitions of certain sub-patterns inside each episode category as prescribed by that category. For example, the "Production editing -> Test failure" subpattern can be repeated one or more times inside an episode of category TF-1 although the instance in Figure 3 under the definition column shows a single occurrence of this sub-pattern.

Zorro uses several heuristics based on the file metrics assoicated with elementary actions. For example, in episode types CP-2 and CP-3, the underlying heuristic checks whether the production code editing action increases the size of the source file by more or less than a preset threshold (currently set to 100 bytes). The threshold determines whether the editing action involves significant amount of production code activity in a single chunk without running the tests. If so, the behavior is counter to TDD and classified as non-conformant. A small amount of production code editing would be indicative of minor tweaks or the addition of a small amount of functionality, such as the addition of getter and setter methods, that do not always warrant the addition of a corresponding test depending on the developer's personal style or the development team's standard practice. This latter behavior is permissible and expected in TDD. Similarly, the long-episode thresholds used in the episode types LN-1 and LN-2 are indicative of behaviors that are too coarse grained, or not sufficiently incremental, for TDD. TDD, being an inherently fine-grained incremental process, discourages large amounts of source or test code editing without running and passing tests. The significant activity threshold and the long-episode thresholds used in episode types LN-1 and LN-2 are arbitrary, but can be changed depending on the expected granularity of the applied process and the tolerance level for deviating from the expected granularity.

Once each episode instance has been assigned an episode type, the final step in the Zorro classification process is to determine the TDD conformance of that instance. Figure 3 shows that exactly half of the 22 episode types can be unambiguously characterized as either TDD conformant or TDD non-conformant. For example, all four Test First episode types are automatically TDD conformant, just the seven Test Last, Long and Unknown episode types are automatically TDD non-conformant.

An interesting discovery from our research is that only half of the episode types we designed could be unambiguously characterized with respect to TDD compliance. The remaining episode types, including Refactoring, Test Addition, Regression, and certain Code Productions are ambiguous: in certain contexts, they could be TDD conformant, while in other contexts they could be TDD non-conformant. For example, consider the "Refactoring" episode type. Code refactoring can occur when a developer is doing Test Driven Design, but it can just as easily occur when a developer is doing some other style of development, such as Test Last programming. In order to classify instances of these ambiguous episode types, Zorro applies the following heuristic: if a sequence of one or more ambiguous episodes are bounded on both sides by non-TDD conformant episodes, then the ambigous episode(s) are classified as non-TDD conformant. Otherwise, they are classified as TDD conformant.

To make this clear, let's consider some examples. For the episode sequence [TF-1, RF-1, CP-1, TF-2], Zorro classifies the interior two ambiguous episodes (RF-1 and CP-1) as TDD conformant, since they are surrounded by TDD conformant episode types (TF-1 and TF-2). Now consider the sequence [TL-1, RF-1, CP-1, TL-2]. In this sequence, Zorro classifies the same two interior

episodes as TDD non-conformant, since they are surrounded by non-TDD episode types (TL-1 and TL-2).

Now consider a sequence like: [TF-1, RF-1, CP-1, TL-1]. Here, the two interior ambiguous episodes (RF-1 and CP-1) are surrounded on one side by an unambiguous TDD conformant episode (TF-1) and on the other side by an unambiguous non-TDD episode (TL-1). In this case, Zorro's rules could implement an "optimistic" classification, and assign the interior ambiguous episodes as TDD conformant, or a "pessimistic" classification, and assign the interior ambiguous episodes as non-TDD. The current Zorro definition of TDD implements the "optimistic" classification for this situation.

Note that all refactoring episode types are context sensitive. This is because refactoring may occur both in TDD and non-TDD contexts. Refactoring may also involve refactoring test code, whether written in a TDD sytle or after the fact. RF-1 and RF-2 episode types represent test refactorings.

The Zorro classification system illustrates two important advances in our approach to TDD. First, it replaces the simplistic "red-green-yellow" three episode type approach to TDD developer behavior with a much more sophisticated classification scheme based upon 22 distinct episode types. Second, it reveals that the mapping from developer behaviors to TDD is not straightforward. One can reasonably question whether the "optimistic" classification scheme currently chosen for Zorro is "correct" or reflects standard or recommended practice. The resolution to this question, and indeed to questions regarding any chosen operational definition of TDD, is *validation*: the process of gathering evidence to determine whether the chosen definition matches reasonable expectations for what constitutes TDD and what doesn't. (Mishali et al, 2008)'s evaluation of their TDD guiding tool suggests that reaching consensus among developer about what consitutes valid TDD behavior may be difficult. We will return to this issue in Section 5.

4.4 The user interface

No matter how effective the classification mechanism, the usefulness of Zorro still depends upon a user interface that can help people understand how Zorro is performing its classification, and what the implications of TDD practice might be. This section overviews a few of the analyses provided by Zorro to provide a flavor for what is possible with this approach. For more details, see Kou (2007) and Wang and Erdogmus (2004).

The first analysis, illustrated in Figure 4, is designed to provide transparency regarding the Zorro data collection and classification process.

Figure 4 displays two episodes, the first containing 19 development stream events and the second containing 10 development stream events. The display of each event includes its time-stamp, its associated file (if applicable), and some additional information about the associated sensor data. The final column provides information about *how* Zorro classified the episode (as either TDD conformant, or TDD non-conformant), as well as *why* Zorro classified

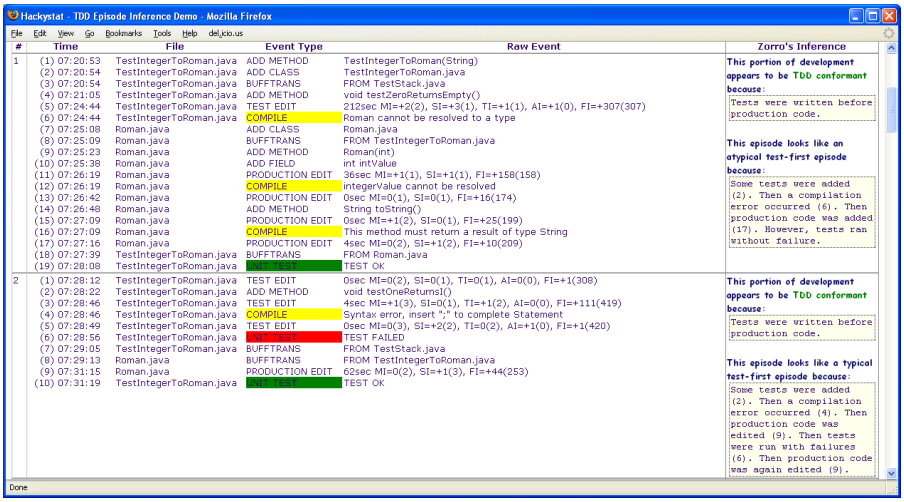


Fig. 4 Zorro Classification Analysis

the episode that way (via a textual summary of the episode's structural characteristics used in the classification). For example, the second case in the figure represents a TF-1 type episode. The summary enclosed in the dashed box lists a sequence of activities that match the TF-1 pattern in 3.

The analysis in Figure 4 is useful for those wishing to understand Zorro's operational definition of TDD in the context of actual development, either for learning or validation purposes. Figure 5 provides a higher level perspective, by showing only the sequence of episode types, with each TDD conformant episode shaded in green. Clicking on an episode type drills down to a more detailed description similar to that shown in Figure 4. Under the column titled Zorro's Inference, the episode classifications are always presented with uncertainty (with wording such as "this portion of developmetn appears to be..." and "this episode looks like...") because no "provably correct" classification exists. Zorro's episode classification, like in other similar tools, is heuristic-based and captures our best current understanding of the TDD process.

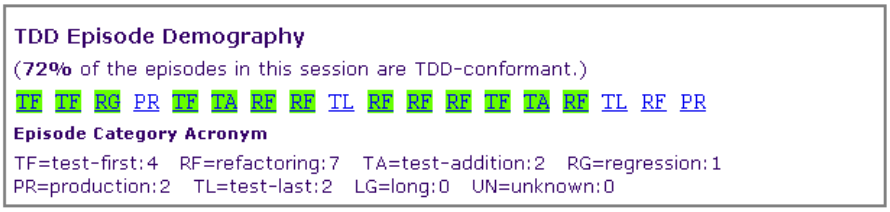


Fig. 5 Zorro Episode Demography

Zorro provides a number of additional analyses that enable the developer to understand the impact of TDD practices on their software product and process. Figure 6 shows how the ratio of test code to non-test (production) code changes during the course of a development session. The horizontal bar at 1.0 represents equal amounts of test and production code. This figure illustrates a scenario of initial module development in which there was significantly more production code than test code at the beginning of the session, but the proportion of test code rose until it doubled the amount of production code, before returning to 1.5 times the production code at the end of the session.

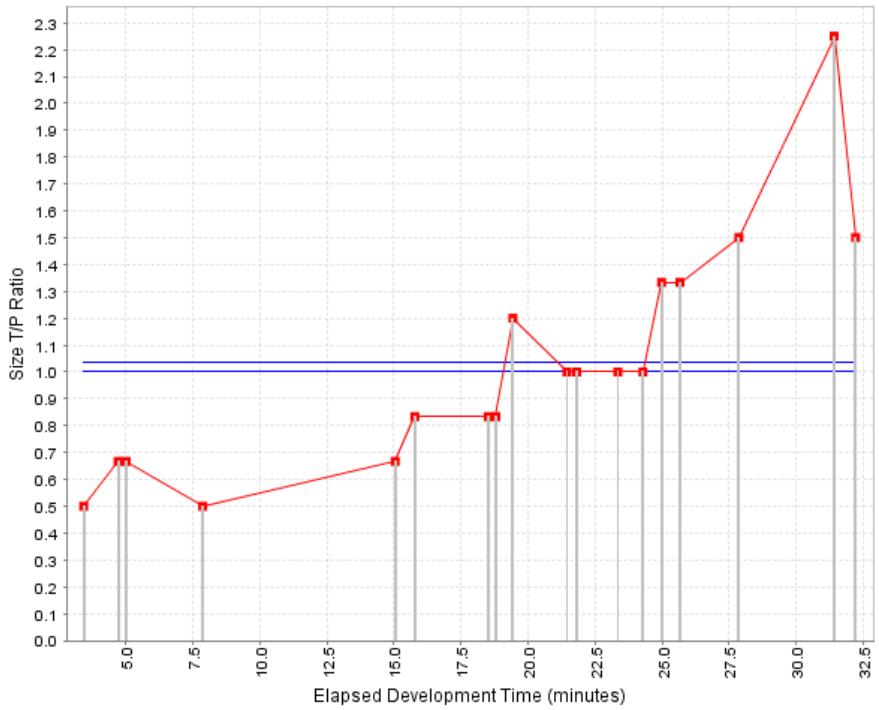


Fig. 6 Zorro Test/Production Size Ratio

The final example analysis illustrated in Figure 7 support a different level of abstraction by using Software Project Telemetry, a capability of Hackystat that enables the visualization of trends in process and product data over days, weeks, or months. In this example of actual development data from the development of the Zorro system itself, two trends are displayed over the course of eight weeks: the percentage of TDD conforming episodes, and the test case coverage of the system under development. One of the claims made for TDD is that consistent use of TDD results in high test coverage (Janzen and Saiedian, 2008). Figure 7 provides anecdotal evidence for an even stronger hypothesis: that test case coverage might co-vary with the consistency of TDD practiced

by the developer. In other words, not only might consistent use of TDD result in high test coverage, but that moving from consistent to inconsistent use of TDD might actually lead to decreasing levels of test coverage.

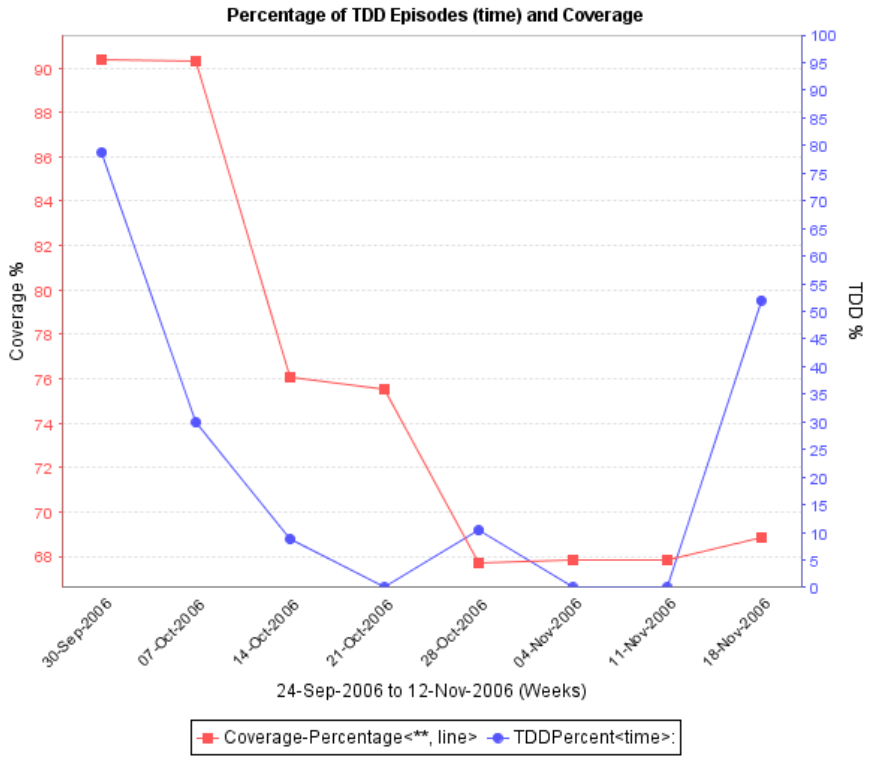


Fig. 7 Zorro TDD Episode Telemetry

5 Empirical validation and evaluation

In order to feel confident in Zorro as an appropriate tool to investigate TDD, we must address two basic validation questions: (1) Does Zorro collect the behaviors necessary to determine when TDD is occurring, and (2) Does Zorro correctly recognize test-driven development when it is occurring?

The first validation issue addresses the use of automated, unobtrusive, sensor-based data collection, and whether this approach can actually acquire the data necessary to determine when TDD is taking place.

The second validation issue addresses our operational definition of TDD based upon episode-based classification, and whether it provides a robust, useful, and acceptable definition of TDD.

An important experimental design issue in the validation of Zorro is the need to obtain an independent source of data regarding developer behaviors apart from the Hackystat sensor data itself. Without this independent source of data, we could not verify that the sensor data was capturing all relevent aspects of developer behavior, or even verify that the sensor implementation was correct.

One approach to independent data collection is to have an observer watching the developers as the program, and take notes as to whether they are performing TDD or not. We considered this approach but discarded it as unworkable: given the rapidity with which TDD cycles can occur, it would be quite hard for an observer to note all of the TDD-related events that can occur literally within seconds of each other. We would effectively need to validate our validation technique!

Instead, we developed a plugin to Eclipse called ESR, the "Eclipse Screen Recorder" (Kou, 2006). This system generates a Quicktime movie containing time-stamped screen shots of the Eclipse window at regular intervals. One frame/second was found to be sufficient for validation, generating file sizes of approximately 7-8 MB per hour of video. The Quicktime movie created by ESR provides an independent, fine-grained, accurate, and visual record of developer behavior that can be manually compared to the Zorro analysis using the timestamps and used to address validation questions.

The following sections summarize the two validation experiments we performed; full details are available in Kou (2007).

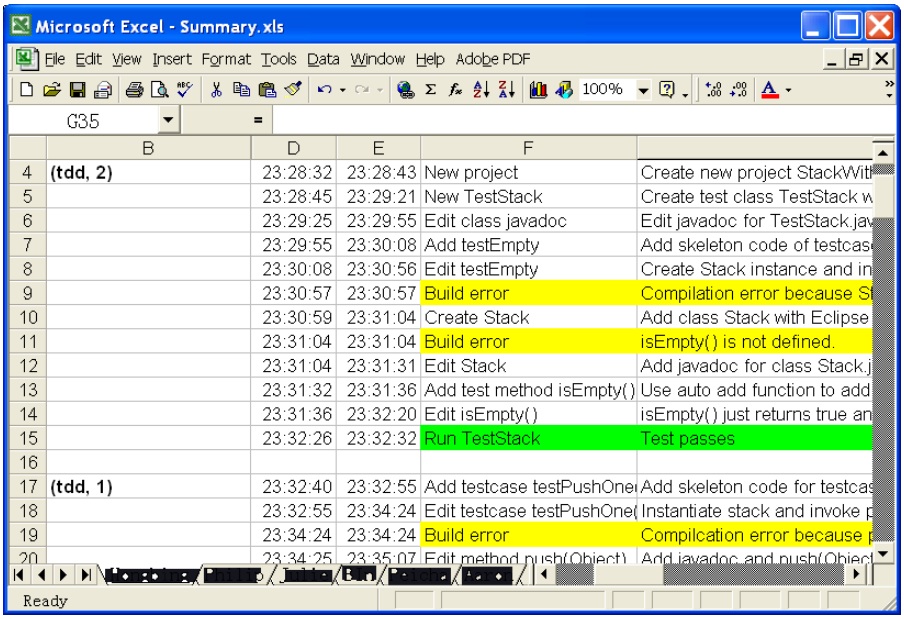
5.1 Experiment 1: Classroom pilot study

Goals. To obtain initial validation data on Zorro, we conducted a short pilot study in Spring of 2006. The goal of this study was to ensure that our data collection and analysis methodology was appropriate and effective.

Procedure. We obtained agreement from seven volunteer student subjects to participate in the pilot study. These subjects were experienced with both Java development and the Eclipse IDE, but not necessarily with test-driven development.

We then provided them with a short description of test-driven design, and a sample problem to implement in a test-driven design style. The problem was to develop a Stack abstract data type using test-driven design, and we supplied them with an ordered list of tests to write and some sample test methods to get them started. Finally, they carried out the task using Eclipse with both ESR and Zorro data collection enabled.

To analyze the data, we created a spreadsheet in which we recorded the results of watching the Quicktime movie and manually encoding the developer activities that occurred. Then, we ran the Zorro analyses and added their results to the spreadsheet. Figure 8 illustrates a portion of such a spreadsheet for one subject: the first column contains the Zorro inferences, while the remaining columns contain manually analyzed data from the ESR video.



	B	D	E	F	
4	(tdd, 2)	23:28:32	23:28:43	New project	Create new project StackWith
5		23:28:45	23:29:21	New TestStack	Create test class TestStack w
6		23:29:25	23:29:55	Edit class javadoc	Edit javadoc for TestStack.jav
7		23:29:55	23:30:08	Add testEmpty	Add skeleton code of testcas
8		23:30:08	23:30:56	Edit testEmpty	Create Stack instance and in
9		23:30:57	23:30:57	Build error	Compilation error because St
10		23:30:59	23:31:04	Create Stack	Add class Stack with Eclipse
11		23:31:04	23:31:04	Build error	isEmpty() is not defined.
12		23:31:04	23:31:31	Edit Stack	Add javadoc for class Stack.j
13		23:31:32	23:31:36	Add test method isEmpty()	Use auto add function to add
14		23:31:36	23:32:20	Edit isEmpty()	isEmpty() just returns true an
15		23:32:26	23:32:32	Run TestStack	Test passes
16					
17	(tdd, 1)	23:32:40	23:32:55	Add testcase testPushOne	Add skeleton code for testcas
18		23:32:55	23:34:24	Edit testcase testPushOne	Instantiate stack and invoke p
19		23:34:24	23:34:24	Build error	Compilcation error because p
20		23:34:25	23:35:07	Edit method push(Object)	Add javadoc and push(Object

Fig. 8 Validation data in Excel

We then checked for consistency: that the inferences made by Zorro matched the behaviors we saw in the video, and whether there were TDD behaviors in the video that Zorro did not capture.

Results. The participants spent between 28 and 66 minutes to complete the task. Zorro partitioned the overall development effort into 92 distinct episodes, out of which 86 were classified as either Test-Driven, Refactoring, or Test-Last; the remainder were "unclassified", which normally corresponded to startup or shutdown activities. Note that the version of Zorro used in Spring 2006 used a somewhat less sophisticated classification ruleset than the current version. Table 3 provides a summary of these validation results. For each subject (ID), the table shows the total number of episodes inferred by Zorro, the number of correctly classified episodes based upon ESR validation, and the resulting percentage correctly classified, ordered from highest to lowest.

Table 3 Case Study 1: TDD Development Behavior Validation

ID	Episodes	Correct	Percentage
5	16	15	94%
3	14	13	93%
4	14	13	93%
6	11	10	91%
7	9	8	89%
1	15	13	87%
2	13	10	77%
Total	92	82	89%

Out of the 92 episodes under study, 82 were validated as correctly classified, for an overall accuracy rate of 89%. The range in percentage correctness ranged from 94% to 77%. Analysis of the differences between Zorro inferences and the ESR data analyses indicated that some editing work and test case invocations were not captured correctly by Zorro. These fixes were made prior to the second case study.

Limitations. The most significant threats to validity of this study were the sample size, the sample population, and the nature of the problem. The study had only seven participants, some of whom were not experienced with TDD. Perhaps the most significant threat was the toy nature of the programming problem, which raised the possibility that we would not be observing "real world" software development behaviors with their attendent complexities.

5.2 Experiment 2: Classroom case study

Goals. After improving the Zorro data collection and analysis mechanisms based upon our first case study, we conducted a second case study in the Fall of 2006. The goal of this case study was to obtain better quality data regarding the strengths and limitations of Zorro for TDD inference by extending our experimental design to include direct participant evaluation of Zorro inferences.

Procedure. We obtained agreement from 11 senior and graduate-level students in two software engineering classes at the University of Hawaii. These students had all recently completed a unit of test driven development in their class, and had done a sample program using TDD principles.

The experimental session lasted approximately two hours. During the first 90 minutes, the students were given a brief introduction to the goals and purpose of the study, then asked to work on the "Bowling Score Keeper", a programming problem used in other empirical studies on TDD (George and Williams, 2003; Erdogmus et al, 2005). A set of user stories were provided to the students to avoid the need for them to know the rules of scoring in bowling, and also to provide them with a "To Do" list for programming. The students were stopped at the end of 90 minutes regardless of whether or not they had completed the entire programming problem.

Once the students finished the programming part of the experiment, they were given a five minute break and then asked to validate Zorro's inferences. To do this, we implemented a special Zorro analysis that would display a representation of each episode and allow the students to provide feedback regarding their agreement with the analysis. Figure 9 shows an example of this wizard.

To analyze the data, we repeated the same process of comparing Zorro analyses to the ESR video as in the first case study. In addition, we also had the subject validation data, which provided a second independent source of information regarding the validity of Zorro inference.

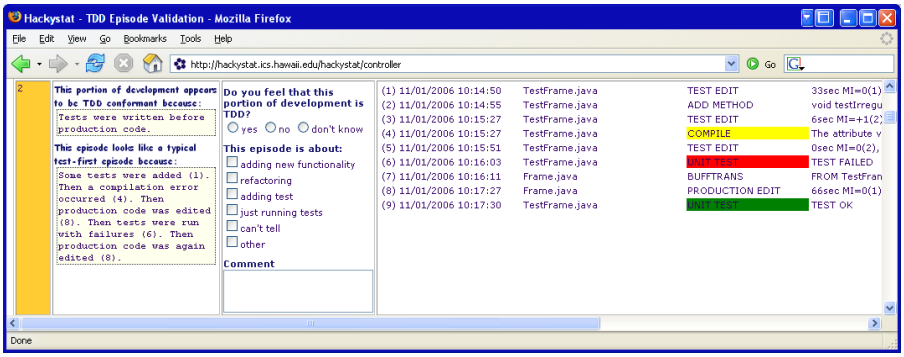


Fig. 9 TDD Episode Validation

Results. The much richer data set collected in this data set makes possible a wide variety of analyses; for a full description, please see Kou (2007). In this section we present only a summary of the results.

First, due to a bug in the data collection mechanism, we lost some data from one participant. We decided to exclude this subject's data from further analysis, reducing our subject pool to 10.

Table 4 provides a summary of the validation results. For each subject (ID), the table lists the number of episodes inferred by Zorro, and the number of those episodes that were correctly classified based upon ESR video data and participant feedback, and the percentage that were correctly classified, ordered from highest to lowest.

Table 4 Case Study 2: TDD Development Behavior Validation

ID	Episodes	Correct	Percentage
O	16	15	94%
R	14	13	93%
L	8	7	88%
K	10	8	80%
A	19	15	79%
T	13	9	69%
P	18	12	67%
Q	21	11	52%
N	9	4	44%
S	9	2	22%
Total	137	96	70%

These results are significantly worse than the first case study: the overall percentage of correctness has dropped from 89% to 70%, and the worst case performance dropped from 77% to 22%!

Analysis of the ESR video data and participant feedback revealed an unexpected developer behavior which appears to account for much of the difference in results between the two studies. In Eclipse, it is possible to invoke unit tests

even when the production code does not compile. Eclipse will issue a warning, but the developer can choose to ignore it and run the tests anyway as long as the non-compiling code is not actually invoked by the test cases. Invoking tests on non-compiling code is a violation of TDD principles, and had not occurred in our usage of Zorro prior to this experiment. Zorro's rules were not configured to deal with this situation, and as a result, 24 out of the 41 incorrectly classified episodes contained an occurrence of this phenomena.

There are two ways to approximate the impact of this developer behavior on Zorro's inference accuracy. First, if we include only those developers who never invoked unit tests when their code did not compile (i.e. the four top subjects K, L, O, R), then the average percentage correctness of episode inference is 90%. Alternatively, if we exclude the 24 episodes in which this phenomena was present from analysis, then the average percentage correctness is 85%. These two approaches together provide some evidence that this single behavior bears significant responsibility for the drop in overall accuracy from the first study.

We can propose two ways to deal with this phenomena in future. First, we could adjust Zorro's current set of rules and/or sensors to take into account the fact that some developers might wish to run unit tests in the presence of non-compiling code. In this case, we would need to make a decision about whether such a behavior would be TDD conformant or not. Alternatively, and more simply, we could recommend that developers intending to apply TDD principles configure Eclipse to disallow this behavior. We believe that the second approach is both easier and more congruent with the current understanding of TDD practice.

In addition to this source of error, our analysis of the data discovered several more minor problems. In two episodes, developers implemented trivial changes to production code but did not run unit tests. The subjects claimed that their behavior was TDD conformant, but without the invocation of test cases, Zorro could not provide the appropriate episode boundary. In four episodes, developers defined a unit test method header, then wrote production code, then wrote the unit test body. Zorro's inference rules were not able to handle a unit test whose initial definition was intermixed with production code editing. Finally, in four episodes, Zorro initially and incorrectly classified test code as production code since Zorro requires assertions in the method body in order to recognize it as a test.

Limitations. As with the first study, a primary threat to the validity of this study is the small sample size. In addition, the use of a student population might influence the external validity of the findings, as professional developers might behave differently than these students. Another threat to external validity is the use of the Bowling Game software problem, which is well suited to laboratory settings but not necessarily representative of real-world software development.

While participant-based validation of Zorro's inferences was quite helpful in detecting certain problems, the views of participants, particularly those with limited experience in TDD, must be taken with care. It is possible that

at least some of the subjects felt that the software might know better than they whether or not they were doing TDD.

5.3 Experiment 3: Industrial case study

One threat to the external validity of our first two case studies is the use of students in a classroom setting. To address this threat, we attempted an industrial case study in order to gain insight into the usefulness of Zorro in a professional setting.

In the summer of 2006, we were contacted by Dr. Geir Hanssen from SINTEF ICT of Norway. Dr. Hanssen was interested in using Zorro with an industrial client who desired to institute TDD and assess its effectiveness. We agreed to serve as technical support for this project, which meant we would provide help with setup and installation of Zorro sensors and analyses. Due to time frame, complexities involved in obtaining approval from our Institutional Research Board, and the nature of the industrial client, we were constrained to a "hands off" approach in which we had no access to the data collected by Zorro.

As part of the preparation for this study, we implemented sensors for the Visual Studio .NET environment to collect the data necessary for TDD inference. Upon initiation of data collection, we were told that the developers were using the TestDriven.NET add-in, and so we developed a sensor for that tool as well.

Out of the 20 developers involved in this industrial case study, it appeared that: four developers installed and activated the sensor; four developers never installed the sensor; four developers either stopped programming or uninstalled the sensor; four developers did not install the TestDriven.NET sensor, and four developers either never wrote test cases or never installed the TestDriven.NET sensor.

According to Dr. Hanssen, the company was faced with tight deadlines and decided that this research on TDD was a low priority for them. Thus, they decided it was not a priority to ensure installation of the sensors, and thus the collected data was insufficent to draw conclusions about the effectiveness of TDD in their setting.

Although the outcome of this study is disappointing from an empirical point of view, it provides at least one "Lessons Learned" for industrial case studies using Zorro: it is very important to have an on-site researcher who can aid in sensor installation and upgrades, as well as simply monitor development to see if lack of sensor data of a particular type represents reality as opposed to a sensor data collection problem. In this study, as neither we nor Dr. Hanssen was provided access to the developers, we could not distinguish between lack of sensor data as an indication that the developer was not working, and lack of sensor data as an indication that the sensor was not installed correctly.

6 Contributions, limitations and future directions

This research contributes in the following ways to the research and practice of test-driven development in particular, and automated software engineering in general.

First, to our knowledge, Zorro is the first, and so far only, fully-automated system capable of recognizing test-driven development practices. Such a system can address the process compliance problem from which much current TDD research suffers.

Second, by providing fully automated recognition, Zorro provides the first precise, operational definition of TDD practice. Our research revealed that this operational definition was not straightforward to implement, as certain kinds of episode types were intrinsicially ambiguous. Resolving these episode types required the implementation of disambiguation heuristics. Our empirical evaluations demonstrated that these heuristics seemed satisfactory in most situations, and that TDD episode recognition accuracy of between 85-90% is quite feasible.

Third, this research results in a new approach to measuring TDD compliance based upon "episodes", or short-duration intervals of development. Using Zorro, one can speak of a development team as having used TDD "55% of the time during the previous week". This much more fine-grained characterization enables new kinds of research on TDD, such as whether there is a threshold for TDD use. For example, perhaps productivity and quality do climb with TDD use up to a threshold of, say, 80%, beyond which there is no discernable improvement. As another example, perhaps there is a steep drop-off in quality and/or productivity if TDD usage drops below, say, 30%.

The Zorro system was implemented within the Software Development Stream Analysis (SDSA) framework. SDSA provides a generic means for recognition of developer "micro-processes" such as TDD. In future work, we hope to write new applications on top of SDSA in addition to Zorro, such as an application for recognizing Continuous Integration best practices.

One limitation of Zorro is due to the fact that the development stream is sequentially partitioned into episodes using test-pass events. This means that Zorro is unable to represent "concurrent" TDD, in which a developer is working on two or more TDD red-green-yellow activities simultaneously. More research is required to determine if such concurrent forms of development occur in practice and what would be required to implement support for automated inference.

Acknowledgements We gratefully acknowledge the members of the Collaborative Software Development Laboratory who supported this research, as well as the student and industrial participants in the evaluation. The National Reseach Council of Canada supported the initial stages of this research under a cooperative agreement. Dr. Burak Turhan of University of Oulu, Finland, provided valuable comments and information regarding tool support for TDD adoption.

References

Beck K (2000) Extreme Programming Explained: Embrace Change. Addison Wesley, Massachusetts

Beck K (2001) Aim, fire. IEEE Softw 18(5):87–89, DOI http://dx.doi.org/10.1109/52.951502

Beck K (2003) Test-Driven Development by Example. Addison Wesley, Massachusetts

Bhat T, Nagappan N (2006) Evaluating the efficacy of test-driven development: industrial case studies. In: ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering, ACM Press, New York, NY, USA, pp 356–363, DOI http://doi.acm.org/10.1145/1159733.1159787

Bill Curtis MIK, Over J (1992) Process modeling. Communications of the ACM 35(9):75–90

Cook JE (1996) Process discovery and validation through event-data analysis. Ph.d thesis, University of Colorado

Cook JE, Wolf AL (1995) Automating process discovery through event-data analysis. In: ICSE '95: Proceedings of the 17th international conference on Software engineering, ACM Press, New York, NY, USA, pp 73–82, DOI http://doi.acm.org/10.1145/225014.225021

Damm L, Lundberg L (2006) Results from introducing component-level test automation and test-driven development. Journal of Systems and Software 79(7):1001–1014

Edwards SH (2004) Using software testing to move students from trial-and-error to reflection-in-action. In: Proceedings of the 35th SIGCSE technical symposium on Computer science education, ACM Press, pp 26–30, DOI http://doi.acm.org/10.1145/971300.971312

Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Trans Softw Eng 31(3):226–237, DOI http://dx.doi.org/10.1109/TSE.2005.37

Friedman-Hill E (2003) JESS in Action. Mannig Publications Co., Greenwich, CT

George B, Williams L (2003) An Initial Investigation of Test-Driven Development in Industry. ACM Sympoium on Applied Computing 3(1):23

George B, Williams L (2004) A Structured Experiment of Test-Driven Development. Information & Software Technology 46(5):337–342

Geras A, Smith M, Miller J (2004) A Prototype Empirical Evaluation of Test Driven Development. In: Software Metrics, 10th International Symposium on (METRICS'04), IEEE Computer Society, Chicago Illionis, USA, p 405

Gupta A, Jalote P (2007) An experimental evaluation of the effectiveness of test-driven development. In: ESEM 2007, 1st International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, pp 285–294

Janzen D, Saiedian H (2005) Test-driven development:concepts, taxonomy, and future direction. Computer 38(9):43–50, DOI http://doi.

ieeecomputersociety.org/10.1109/MC.2005.314
Janzen D, Saiedian H (2008) Does test-driven development really improve software design quality. IEEE Software pp 77–84
Jeffries R (2000) Extreme Programming Installed. Addison Wesley, Upper Saddle River, NJ
Jensen C, Scacchi W (2004) Process modeling across the web information infrastructure. In: Special Issue on ProSim 2004, The Fifth International Workshop on Software Process Simulation and Modeling, Edinburgh, Scotland
Jensen C, Scacchi W (2005) Experience in discovering, modeling, and reenacting open source software development processes. In: Proceedings of the International Software Process Workshop
Johnson PM, Paulding MG (2005) Understanding HPCS development through automated process and product measurement with Hackystat. In: Second Workshop on Productivity and Performance in High-End Computing (P-PHEC), URL http://csdl.ics.hawaii.edu/techreports/04-22/04-22.pdf
Johnson PM, Kou H, Agustin JM, Zhang Q, Kagawa A, Yamashita T (2004) Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In: Proceedings of the 2004 International Symposium on Empirical Software Engineering, Los Angeles, California, URL http://csdl.ics.hawaii.edu/techreports/03-12/03-12.pdf
Johnson PM, Kou H, Paulding MG, Zhang Q, Kagawa A, Yamashita T (2005) Improving software development management through software project telemetry. IEEE Software URL http://csdl.ics.hawaii.edu/techreports/04-11/04-11.pdf
Kaufmann R, Janzen D (2003) Implications of test-driven development: a pilot study. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, New York, NY, USA, pp 298–299, DOI http://doi.acm.org/10.1145/949344.949421
Kellner M, Feiler P, Finkelstein A, Katayama T, Osterweil L, Penedo M, Rombach H (1991) Ispw-6 software process example. In: Proceedings of the First International Conference on the Software Process, Pittsburgh, PA
Kou H (2006) Eclipse screen recorder. http://csdl.ics.hawaii.edu/Tools/Esr/
Kou H (2007) Automated inference of software development behaviors: Design, implementation and validation of zorro for test-driven development. Ph.D. thesis, University of Hawaii, Department of Information and Computer Sciences, URL http://csdl.ics.hawaii.edu/techreports/07-04/07-04.pdf
Kou H, Johnson PM (2006) Automated recognition of low-level process: A pilot validation study of Zorro for test-driven development. In: Proceedings of the 2006 International Workshop on Software Process, Shanghai, China, URL http://csdl.ics.hawaii.edu/techreports/06-02/06-02.pdf

Madeyski L, Szala L (2007) The impact of test-driven development on software development productivity - an empirical study. In: Abrahamsson P, Badoo N, Margaria T, Messanarz R (eds) Software Process Improvement, 9th International Conference on Agile Processes in Software Engineering and Extreme Programming, Springer-Verlag, Heidelberg, Germany, Lecture Notes in Computer Science, vol 4764, pp 200–211

Maximilien EM, Williams L (2003) Accessing Test-Driven Development at IBM. In: Proceedings of the 25th International Conference in Software Engineering, IEEE Computer Society, Washington, DC, USA, p 564

Mishali O, Dubinsky Y, Katz S (2008) The tdd-guide training and guidance tool for test-driven development. In: Abrahamsson P, Baskerville R, Conboy K, Fitzgerald B, Morgan L, Wang X (eds) XP 2008: 9th International Conference on Agile Processes in Software Engineering and Extreme Programming, Springer-Verlag, Heidelberg, Germany, Lecture Notes in Business Information Processing, vol 9, pp 63–72

Muller MM, Hagner O (2002) Experiment about Test-first Programming. In: Empirical Assesment in Software Engineering (EASE), IEEE Computer Society

Pančur M, Ciglarič M, Trampuš M, Vidmar T (2003) Towards empirical evaluation of test-driven development in a university environment. In: Proceedings of EUROCON 2003, IEEE

Sanchez J, Williams L, Maximillien E (2007) On the sustained use of test-driven development practice at ibm. In: Agile 2007 Conference, pp 5–14

Siniaalto M, Abrahamsson PA (2007) Comparative case study on the impact of test-driven development on program design and test coverage. In: ESEM 2007, 1st International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, pp 275–284

Sutton SM, Heimbigner D, Osterwell LJ (1995) APPL/A: A language for software process programming. ACM Transaction on Software Engineering and Methodology 4(3):221–286

Turnu I, Melis M, Cau A (2004) Introducing tdd on a free libre open source software project: a simulation experiment. In: QUTE-SWAP Workshop, ACM Press, New York, NY, USA

Wang Y, Erdogmus H (2004) The role of process measurement in test-driven development. In: XP/Agile Universe, pp 32–42

Wege C (2004) Automated support for process assessment in test-driven development. Ph.d thesis, Eberhard-Karls-Universit at Tubingen

Williams L, Maximilien EM, Vouk M (2003) Test-driven development as a defect-reduction practice. In: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE03), ACM Press, New York, NY, USA, pp 298–299