Automated Recognition of Test-Driven Development with Zorro

Philip M. Johnson
Hongbing Kou
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu

Abstract

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. This paper introduces the Zorro system, its operational definition of TDD, the analyses made possible by Zorro, and our ongoing efforts to validate the system.

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 [1]. Unfortunately, the empirical research results have been equivocal. Some results are positive: Bhat and Nagappan found that introducing TDD at Microsoft decreased defect rates significantly in two projects [2], and Maximilien and Williams transitioned an IBM development team to TDD with a 50% improvement in quality [8]. But other results are negative: Muller and Hanger found that TDD resulted in lower reliable software than the control group [9] and Erdogmus found that TDD software was of no higher quality than the control group [3].

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

A first problem is that TDD is often introduced in a relatively simplistic way, such as with the "Red-Green-Yellow" stoplight metaphor. This definition of TDD can mislead developers into thinking that all software development to which TDD applies must easily reduce to "Write a little test that doesn't work; Make the test work quickly, committing whatever sins are necessary in the process; and eliminate the duplication created while keeping all tests passing." While that definition may suffice for the problems used to illustrate and teach TDD, real world software development scenarios tend to be more complicated. As one simple example, must one always begin in TDD with a test case that doesn't work? What about maintenance scenarios where one encounters code that is missing appropriate tests, and so the developer writes a test that happens to work the first time? Must that be classified as "not TDD" simply because it does not fit the Red-Green-Yellow pattern? As we learned in our research, developers don't tend to be binary, either utilizing TDD practices perfectly correctly all the time or, alternatively, never doing TDD at all.

A second methodological problem with TDD research involves compliance, or verification that the

1

participants who are *supposed* to be doing TDD are *actually* doing TDD. Many published papers on TDD case studies and experiments provides little discussion of how they verified compliance with the TDD process. Both Janzen and Wang discuss how the question of compliance weakens the validity of TDD research [4, 10].

These two issues lead to confusion in both the practice of and research on TDD. An overly simplistic definition of TDD can lead software developers to abandon the approach when they encounter development situations outside the contexts where the Red-Green-Yellow pattern applies directly. Alternatively, some developers might believe that they are doing TDD due to an excessively relaxed personal version of the definition, when other expert TDD practitioners might disagree. The lack of compliance controls on experimental settings means that differences in outcomes may be due, at least in part, to variance in understanding what TDD actually is, as opposed to differences between the control and experimental groups.

To address these problems, we believe that the community needs to agree upon one (or more) standard, operational definitions of TDD. Furthermore, these definition(s) must allow for a practical way to assess compliance in both laboratory and real-world settings.

In this paper, we report on our results so far with Zorro, 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 applies a rule-based system to determine whether or not each episode constitutes an instance of a TDD practice.

Zorro illustrates one approach to addressing the issues mentioned above that hinder the research and practice of TDD today. Automatic collection and analysis of data makes Zorro practical for use in both laboratory and real-world settings: once installed, there is no overhead on the developer with respect to data collection. 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 collected, the manner in which this timestamped stream of events are partitioned into episodes,

and the rules used to determine if an episode is TDD. 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.

2   Zorro's Architecture

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.

2.1   Hackystat

Hackystat 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 via SOAP to a web server for analysis via server-side Hackystat "applications". 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 [6], high performance computing [7], and software engineering education [5].

2.2   SDSA

Software Development Stream Analysis (SDSA) is a Hackystat application that provides a generic framework for organizing and analyzing the various kinds
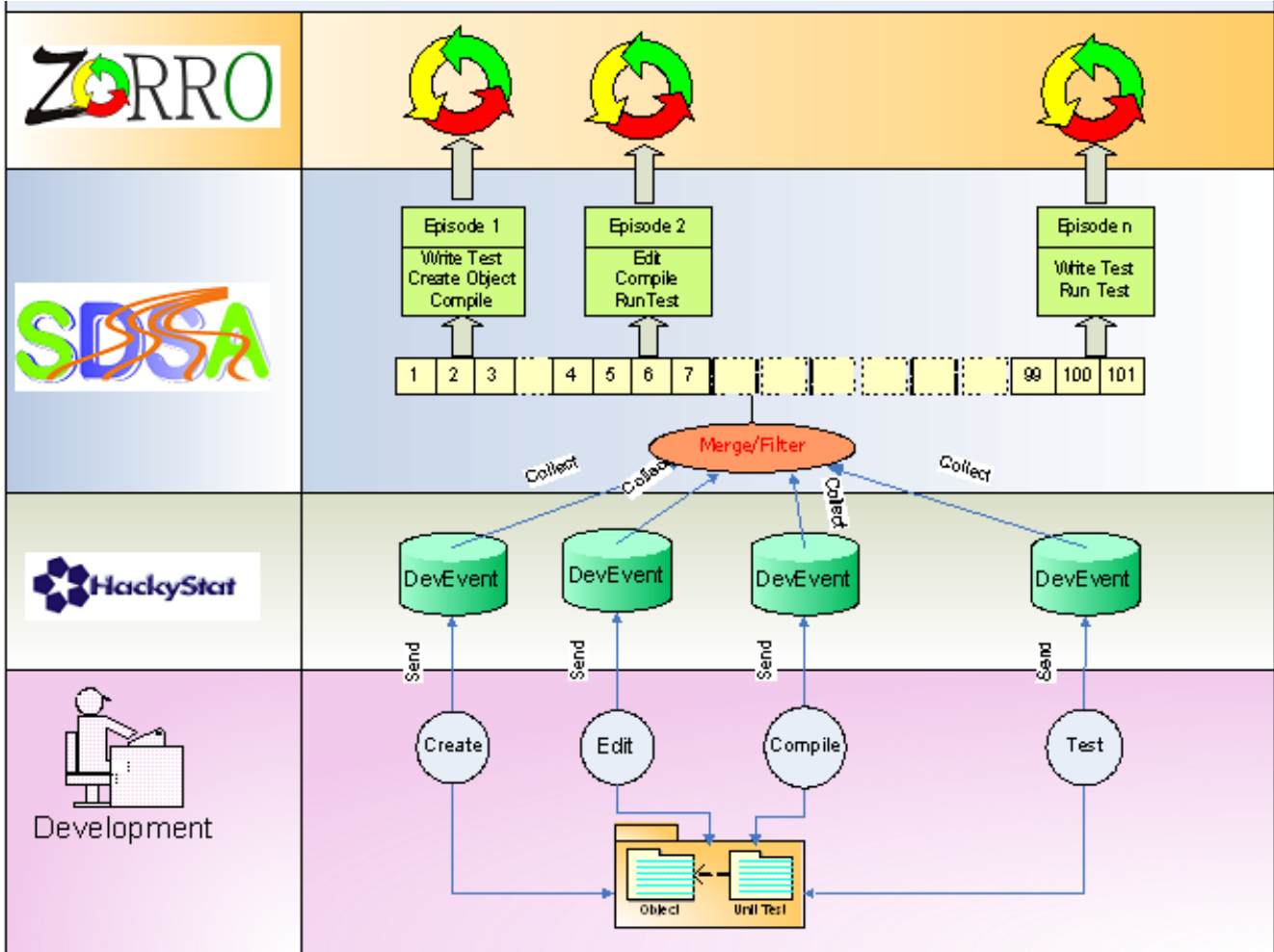
Figure 1. The Zorro Architecture

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.

2.3   Zorro

The Zorro architectural layer provides extensions to Hackystat and SDSA necessary for the automated recognition of Test Driven Development behaviors.

2.3.1 Zorro extensions to Hackystat sensors

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). While these event types are both language and IDE independent, our current implementation requires the use of the Java programming language, the JUnit testing framework, and the Eclipse IDE.

2.3.2 Zorro extensions to SDSA

Zorro's extensions to SDSA begin with the specification of the episode boundary condition, which for TDD is the occurrence of a passing test case.

Zorro also extends SDSA with a set of rules that enable instances of episodes to be classified as one of 22 episode types. Figure 2 lists these episode types, their definition in terms of their internal development stream structure, and an indication of their 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.)

Once each episode instance has been assigned an episode type by the SDSA rule set, the final step in the Zorro classification process is to determine the TDD conformance of that instance. Figure 2 shows that instances of some of the episode types are easy to characterize. For example, every instance of a Test First episode type is automatically TDD conformant, just every instance of a Test Last, Long and Unknown episode type is automatically not TDD conformant.

Interestingly, several of the episode types, such as Refactoring, Test Addition, Regression, and certain Code Productions are ambiguous: in certain contexts,

they could be TDD conformant, while in others they could be TDD non-conformant. This is because, for example, "Refactoring" can legitimately occur while a developer is either doing Test Driven Design or some different development approach, such as Test Last programming. In order to classify instances of these 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 these ambigous episodes types are classified as non-TDD conformant.

To make this clear, let's consider some examples, such as the episode sequence [TF-1, RF-1, CP-1, TF-2]. In this sequence, Zorro classifies the 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 ambiguous episodes (RF-1 and CP-1) 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 ambiguous episodes (RF-1 and CP-1) are surrounded on one side by an unambiguously TDD conformant episode (TF-1) and on the other side by an unambiguously TDD non-conformant episode (TL-1). In this case, Zorro's rules could implement an "optimistic" classification, and assign the ambiguous episodes as TDD, or a "pessimistic" classification, and assign the ambiguous episodes as non-TDD. The current Zorro definition of TDD implements an "optimistic" classification for this situation.

The Zorro classification system illustrates two important advances in our approach to TDD. First, it replaces the simplistic three episode type (red, green, yellow) approach to TDD developer behavior with a 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. 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. We will return to

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

Figure 2. Zorro episode types, definitions, and TDD conformance

this issue in Section 3.

2.3.3 Zorro extensions to Hackystat Analyses

Having collected the raw data using Hackystat sensors, and having abstracted the raw data into episodes and classified it using SDSA, the final step in Zorro is to provide analyses that are useful to both TDD developers and TDD researchers. This section overviews a few of the analyses provided by Zorro to provide a flavor for what is possible with this approach.

The first analysis, illustrated in Figure 3, is designed to provide visibility into the Zorro data collection and classification process.

Figure 3 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 ap-

plicable), 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 the episode that way (via a textual summary of the episode structural characteristics used in the classification).

The analysis in Figure 3 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 4 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 3.

Zorro provides a number of additional analyses that enable the developer to understand the impact of TDD

Figure 3. Zorro Classification Analysis

TDD Episode Demography

(**72%** of the episodes in this session are TDD-conformant.)

TF TF RG PR TF TA RF RF TL RF RF RF TF TA RF TL RF PR

Episode Category Acronym

TF=test-first:4   RF=refactoring:7   TA=test-addition:2   RG=regression:1
PR=production:2   TL=test-last:2   LG=long:0   UN=unknown:0

**Figure 4. Zorro Episode Demography**

practices on their software product and process. Figure 5 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.

The final example analysis illustrated in Figure 6 pops up to yet another 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 real world data, 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. Interestingly, the level of test case coverage co-varies with the "level" of TDD practiced by the developer.

3 Validation

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 behav-
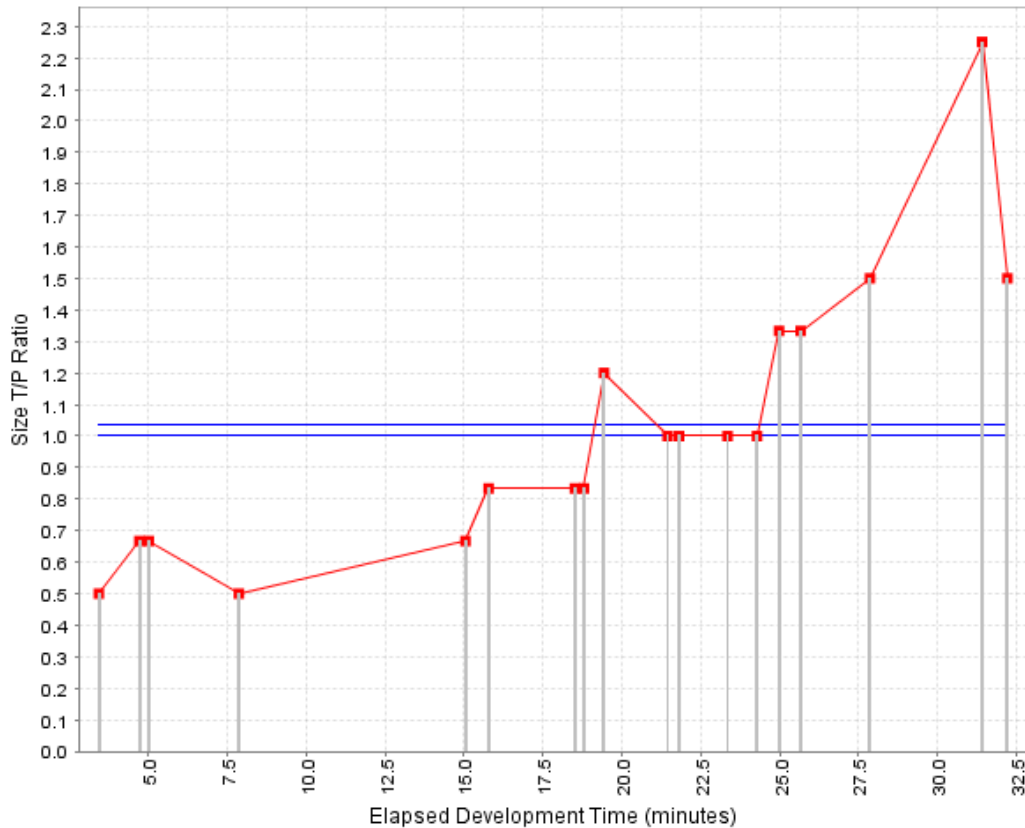
Figure 5. Zorro Test/Production Size Ratio

iors 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.

We began Zorro validation in the Spring of 2006 with a pilot study, and are building on that initial work in 2007.

3.1 The pilot validation study

To obtain some initial validation data on Zorro, we conducted a pilot study in Spring of 2006 in which we instrumented the development environment with Zorro sensors, asked a small set of students to do some simple TDD development, then compared the resulting Zorro TDD classifications to an independently collected source of data regarding their development behaviors.

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

Instead, we developed a plugin to Eclipse called the "Eclipse Screen Recorder" (ESR). 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
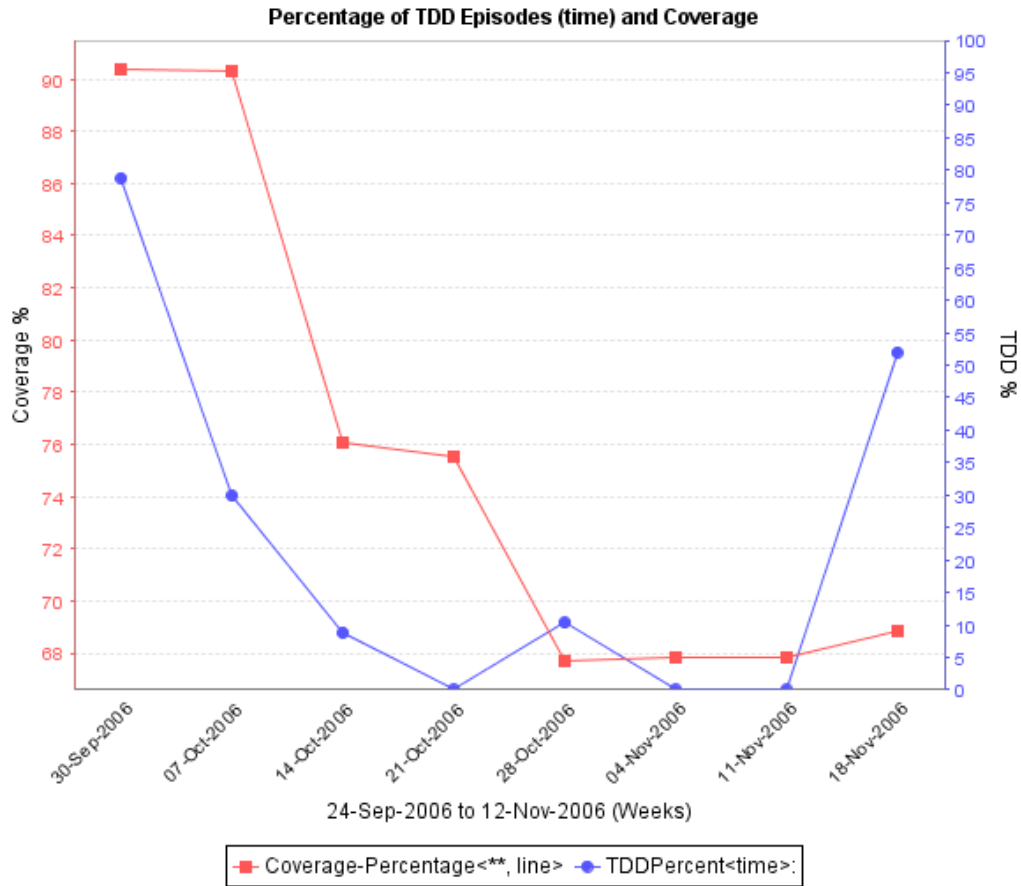
Figure 6. Zorro TDD Episode Telemetry

per hour of video. The Quicktime movie created by ESR provides a visual record of developer behavior that can be manually compared to the Zorro analysis using the timestamps and used to answer the two validation questions.

Our pilot validation study involved the following procedure. First, 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. Second, we 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, added their results to the spreadsheet, and validated the Zorro classifications against our analysis of the video record.

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.

Out of the 92 episodes under study, 82 were validated as correctly classified, for an accuracy rate of 89%.

3.2 Ongoing validation

The pilot study provided us with valuable feedback about the potential utility of Zorro, as well as deeper insight into the process of validation itself. Our current research is focused on enhancing our understanding of the strengths and weaknesses of Zorro with several additional validation studies.

Our next validation study will also involve students, and will expand on the pilot study by cross-validating the Zorro operational definition of TDD against two independent data sources: the ESR video stream and the feedback of the participants themselves following the TDD development session. We will ask them to review the TDD episodes generated by Zorro and provide their personal feedback on the classifications.

Following this second student-based validation study, we plan to gain insight into the strengths and weaknesses of Zorro in professional settings. For this case study, we will invite developers who practice TDD and who use Eclipse, Java, and JUnit to help us validate Zorro. The process involves installing the Hackystat Eclipse sensor for Zorro into their development environment, performing development for a few days with the sensors installed, then reviewing the classifications made by Zorro and providing us with feedback regarding its accuracy.

We also plan to obtain feedback from the TDD research community. We believe that Zorro can provide useful infrastructure for research on test driven development. For this case study, we will invite researchers to evaluate Zorro against their experimental requirements and provide us with feedback as to the suitability of Zorro for their own work.

4 Conclusions

TDD has great potential as a software development technique, but to fully realize this potential, the software development community must gain deeper insight into its strengths and weaknesses. The Zorro system shows how it is possible to define an operational definition of Test Driven Development that can be used to address the process conformance problem and other methodological issues confronting researchers in TDD. Zorro also enables TDD to be analyzed and understood with more precision and nuance than ever be-

fore: instead of red-green-refactor, one can now talk about the percentage of TDD episodes and which of 22 episode types have been used. When combined with other Hackystat analysis techniques such as Software Project Telemetry, Zorro enables developers to gain deeper empirical insight into how TDD practice impacts on other process and product metrics, such as test case size and coverage.

In our experience, however, Zorro is more than simply an experimental infrastructure or a TDD learning device. Zorro demonstrates that it is now possible to observe and analyze interesting "micro-processes" in software development. By building Zorro on top of the more generic SDSA and Hackystat frameworks, its architecture makes it more easily possible to study not only TDD, but other interesting developer "best practices" on this fine-grained level. For example, there are many best practices surrounding the appropriate time to commit file changes to a configuration management repository, and at least some of these best practices could be operationalized in a set of Hackystat sensors and SDSA rules.

Finally, we want to emphasize the open source nature of the Zorro system and the research process. We encourage you to download the system and try it out, or contact us if you wish to participate in the research process.

References

[1] K. Beck. *Test-Driven Development by Example*. Addison Wesley, Massachusetts, 2003.

[2] T. Bhat and N. Nagappan. 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*, pages 356–363, New York, NY, USA, 2006. ACM Press.

[3] H. Erdogmus. On the effectiveness of the test-first approach to programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237, 2005.

[4] D. Janzen and H. Saiedian. Test-driven development:concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[5] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Sym-*

posium on Empirical Software Engineering, Los Angeles, California, August 2004.

[6] P. M. Johnson, H. Kou, M. G. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Software*, August 2005.

[7] P. M. Johnson and M. G. Paulding. Understanding HPCS development through automated process and product measurement with Hackystat. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[8] E. M. Maximilien and L. Williams. Accessing Test-Driven Development at IBM. In *Proceedings of the 25th International Conference in Software Engineering*, page 564, Washington, DC, USA, 2003. IEEE Computer Society.

[9] M. M. Muller and O. Hagner. Experiment about Test-first Programming. In *Empirical Assesment in Software Engineering (EASE)*. IEEE Computer Society, 2002.

[10] Y. Wang and H. Erdogmus. The role of process measurement in test-driven development. In *XP/Agile Universe*, pages 32–42, 2004.