AUTOMATED INFERENCE OF SOFTWARE DEVELOPMENT BEHAVIORS: DESIGN, IMPLEMENTATION AND VALIDATION OF ZORRO FOR TEST-DRIVEN DEVELOPMENT

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

DECEMBER 2007

By
Hongbing Kou

We certify that we have read this dissertation and that, in our opinion, it is satisfactory in scope and quality as a dissertation for the degree of Doctor of Philosophy in Computer Science.

DISSERTATION COMMITTEE

Chairperson

Dedicated to:

My wife, Rong,

I will achieve nothing without you.

My Son, Daniel,

Every smile counts.

My Mother,

You enlighten me to do all the right things.

My Father,

Your love is always with me no matter where I go.

My Brothers,

Your unconditional supports are treasures beyond compare.

Acknowledgments

I would like to express my deep and sincere gratitude to my advisor, Professor Philip Johnson. Your wide knowledge and creative way of thinking have been a great value of me. Thank you for spending numerous hours on proof-reading this manuscript. This thesis would never have become possible without you.

I would like to thank Dr. Hakan Erdogmus from National Research Council of Canada. With your guidance and mentoring, August 2006 has become the most productive month in my thesis research.

I would like to thank Dr. Daniel Suthers, Dr. Dan Port and Dr. David Pager. Your advising and support have been priceless in my graduate studies.

I would like to thank my thesis committee members: Dr. Wes Peterson and Dr. Kim Binsted. Your time and support are precious to me.

I would like to thank my fellow CSDL hackying buddies: Qin Zhang, Joy Augustin, Aaron Kagawa, Julie Sakuda, Takuya Yamashita, Mike Paulding, Robert Brewer, Pavel Senin, Austen Ito, Jitender Miglani and Christopher Lofi.

I would like to thank Wesley Sugimoto and Janice Oda-Ng from the Department of Information and Computer Sciences. Thank you two for spending so much time on the tedious paper work for me.

Lastly, I would like to thank my friends Xin Zhao, Qin Guo, Ming Liu, Hao Zhou, Hu Li (Tiger), Decheng Yang, Hongbo Ding, Ying Zhang, Shidong Kai, Yangheng Zheng, Yuhuan Li, Rui Xue, Xiangli Xu and many others. Life would not be so wonderful without you guys.

Abstract

A recent focus of interest in software engineering research is on low-level software processes, which define how software developers or development teams should carry on development activities in short phases that last from several minutes to a few hours. Anecdotal evidence exists for the positive impact on quality and productivity of certain low-level software processes such as test-driven development and continuous integration. However, empirical research on low-level software processes often yields conflicting results. A significant threat to the validity of the empirical studies on low-level software processes is that they lack the ability to rigorously assess process conformance. That is to say, the degree to which developers follow the low-level software processes can not be evaluated.

In order to improve the quality of empirical research on low-level software processes, I developed a technique called Software Development Stream Analysis (SDSA) that can infer development behaviors using automatically collected in-process software metrics. The collection of development activities is supported by Hackystat, a framework for automated software process and product metrics collection and analysis. SDSA abstracts the collected software metrics into a software development stream, a time-series data structure containing time-stamped development events. It then partitions the development stream into episodes, and then uses a rule-based system to infer low-level development behaviors exhibited in episodes.

With the capabilities provided by Hackystat and SDSA, I developed the Zorro software system to study a specific low-level software process called Test-Driven Development (TDD). Experience reports have shown that TDD can greatly improve software quality with increased developer productivity, but empirical research findings on TDD are often mixed. An inability to rigorously assess process conformance is a possible explanation. Zorro can rigorously assess process conformance to a specific operational definition for TDD, and thus enable more controlled, comparable empirical studies.

My research has demonstrated that Zorro can recognize the low-level software development behaviors that characterize TDD. Both the pilot and classroom case studies support this conclusion. The industrial case study shows that the automated data collection and development behavior inference has the potential to be useful for researchers.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Throughout the history of software engineering, much effort has been put on the description and understanding of high-level software processes. The waterfall model, the very first software process, has contributed to the success of many large software systems. High-level software processes divide the software development process into phases, where each phase lasts from a few days to several months [54, 56]. For example, the requirements analysis phase may last months before the design phase starts. Recently, increasing effort has been put on low-level software processes [46, 1], in which a phase may last from several minutes to a few hours only. Each phase defines how developers and the development team should carry on the work on a daily basis. The Personal Software Process (PSP) [29] and Extreme Programming (XP) [32, 4, 20] are two examples of a low-level software process. Although proven to be useful in improving software quality[21, 43, 70, 31], low-level software processes are hard to execute correctly and repeatedly. Low-level software processes have the potential to require new skills from software organizations, project managers, and software developers. For example, in Test-Driven Development (TDD), each developer is a requirements analyst, designer, tester, and coder. As a result, a low-level software process could be used differently in different software organizations. Worse yet, an organization might think they are using a particular low-level process, such as TDD, but in reality, they are doing something quite different. In order to improve the quality on practice and research of low-level software processes, it would be helpful to have tools to support the understanding of what low-level software processes are actually being performed by organizations, and what the impact of these processes are on the outcome of development. In my dissertation research, I focused on one low-level software process called Test-Driven Development (TDD) [6], and developed the Zorro software system to study it.

1.1 Test-Driven Development

“Clean code that works”[6] is the goal of Test-Driven Development. To achieve this goal, TDD summarizes its low-level software development process as two basic rules: “(1) Write new code only if an automated test has failed; (2) Eliminate duplication.” Kent Beck, the pioneer of Test-Driven Development, stated that there is an implicit order to software development using TDD [6]:

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. Refactor - Eliminate all the duplication created by merely getting the test to work.

The red/green/refactor order is a pithy summary of TDD. In reality, TDD is significantly more complicated than that. Given a requirement, a TDD developer analyzes it and outlines a To-Do list with a few tasks. After picking a task from the To-Do list, the developer immediately writes a test, which is then used to motivate the implementation of actual system. If the test fails, the developer does whatever is necessary to make it pass as quickly as possible, not worrying about the quality or generality of the results. Only after getting the test case to pass does the developer perform a pass over the code base to improve its quality without changing its functionality. This is called the “refactoring” phase. Then the developer crosses out this task from the To-Do list, and adds new tasks to it if new requirements occur to him.

Because a test is always created first to drive the design and implementation, TDD used to be called Test-First Design (TFD) or Test-First Development (TFD). My opinion is that “test first” is better than “test driven” for describing the order of test and production coding activities. Therefore, in the rest of this document, I will use “test first” when it is necessary to emphasize the order of programming in TDD, but there is no difference between “test first” and “test driven”.

TDD is one of the innovative practices of Extreme Programming. In TDD, the software development process is iterative and incremental [46]. There is only one task to accomplish in an iteration. In any particular iteration, a unit test corresponding to the task is created first, followed by production code implementation. TDD is built on the foundation of the XUnit framework [59], which has been ported to more than 30 languages. Unit testing has become a defacto standard in the software industry. TDD is widely adopted by software professionals. An informal survey [73] conducted by the Method and Survey magazine found that 46% of the studied software organizations perform unit testing informally, 41% of the studied organizations document their unit test cases, and 14% of the studied organizations use the TDD approach.

1.2 TDD Challenges

At first glance, TDD might seem easy, but in fact, it is a very difficult low-level software process that requires much discipline to carry out correctly. First, software developers are not typically educated to write unit tests for the programs they develop. Therefore, in a lot of cases, software systems are not designed for easy testing. Consequently, developers often find it is hard to write testing code at all, much less write testing code prior to implementation. Second, following the red/green/refactor software development pattern requires a lot of discipline. In TDD, software developers must continuously remain in the mindset of test-first, which is initially counter-intuitive to many of them [5, 74]. So they often apply it differently according to their own experience level and understanding [5]. Third, the best way to divide a complicated problem into a set of tasks that can be finished in short iterations is not always obvious. Perhaps as a result,the research findings on TDD's impact on software quality and developer productivity are mixed. In order to improve the TDD practices and empirical evaluations, it is necessary to measure the usage and adoption of TDD. In this section, I will first present some TDD research findings, followed by a discussion of the "construct validity" problem of these studies.

1.2.1 Mixed TDD Research Findings

So far, software engineering researchers have focused heavily on the important outcomes that TDD brings to software products and software developers. Both pedagogical [50, 18, 25, 53, 19, 44] and industrial [23, 49, 7] evaluations of TDD have been conducted in the last few years. Table 1.1 lists most relevant TDD research work. It is interesting to note that while the industrial empirical evaluations results are often positive, on the contrary, the pedagogical evaluations results are often negative toward TDD. The industrial studies found that TDD helped to improve software quality at the cost of development time according to Table 1.1. Most pedagogical studies did not find the improvements of software quality and degradation of developer productivity. Instead, some pedagogical studies found that students as participants improved their productivity. As presented in Table 1.1, discrepancies exist among empirical research findings, and they are not trivial. Although

Table 1.1. Research Work of TDD on Software Quality and Developer Productivity

	Investigator	Participants	Software Quality	Developer Productivity
Industrial	George [23]	24	TDD passed 18% more tests	16% more time
	Geras [25]	14	TDD has the edge on quality	No impact
	Maximilien [49]	9	50% reduction in defect density	Minimal impact
	Williams[76]	9	40% reduction in defect density	No change
	Bhat [7]	11	2-4 times reduction in defect density	35% and 15% more time
Academic	Kaufmann [44]	8	N/A	50% improvement
	Edwards [18]	59	54% fewer defects	N/A
	Erdogmus [19]	35	No change	Improved productivity
	Müller [50]	19	Less reliable, but better reuse	No change
	Pancŭr [53]	38	No change	No change

it is commonly understandable that empirical research work can not be repeated easily, the TDD research findings are too inconsistent to discount as mere replication variability. Next, I will discuss the process conformance problem, one of the most possible reasons that contributed to the mixed results, and motivate the need to measure TDD usage and adoption.

1.2.2 Measures of TDD Usage and Adoption

Much of the research work on TDD suffers from the threat of "construct validity" [74] because of the what has been termed as the "process conformance" problem. Wang and Erdogmus defined process conformance as the ability and willingness of the subjects to follow a prescribed process. The process conformance problem of TDD can be expressed as a simple question, "Do test-driven developers really do test-first?" It is a question that implies two sub questions.

1. Do test-driven developers have the abilities to develop software in TDD?
2. Do test-driven developers develop software in TDD consistently?

An empirical study has the construct validity problem if the answer to the first question is not a firm "yes", and it has the internal validity problem if the answer to the second question is not a firm "yes".

Unfortunately, researchers have not paid much attention to either of these validity problems in the empirical studies I mentioned in the previous section. Some researchers used pair programming [23] and verbal confirmation [50] as the process control methods, but they are not reliable sources. So process conformance is one possible explanation to the mixed research findings of TDD.

In the software industry, TDD is gradually becoming well accepted for software development [73]. Some companies even put TDD in their job descriptions. Yet there are still problems in testability and differences in the understanding of this methodology. In [31], Janzen and Saiedian warned that measuring the adoption of TDD is necessary. Many organizations might be using it without talking about it. Others might claim to be using TDD when in fact they are misapplying it. Worse yet, they might be advertising its use falsely. Surveys could be conducted to gauge the usage of TDD, but often only those who are much in favor or much opposed to it will respond.

The inability to accurately characterize process conformance is harmful to TDD. Therefore, measuring the usage and adoption of TDD has become an important issue for both researchers and practitioners. However, Janzen and Saiedian [31] stated that measuring the use of a software development methodology is hard. They claimed it is so hard to do accurately that published data on the level of TDD adoption in industry is either indirect or inaccurate [31, 73].

Fortunately, as my initial case study demonstrates [45], measuring the use of certain software development methods such as TDD is becoming feasible with the emergence of technologies such as the Hackystat system [60, 40, 41, 39], an in-process software metrics collection and analysis framework.

1.3 Proposed Solution: Zorro Software System

In this research, I developed the Zorro software system to measure the usage of TDD with the capabilities introduced by the Hackystat system, a framework of automated software metrics collections and analyses. Figure 1.1 illustrates the infrastructure of the Zorro software system. Zorro is built on top of Hackystat, and it uses Hackystat's data collection and analysis services for development behavior inference of TDD. In Figure 1.1, between Hackystat and Zorro layers, there is a middle tier named SDSA. The SDSA stands for the Software Development Stream Analysis, a generic framework for development event stream analysis including three components —software development stream construction, development stream partition, and development behavior

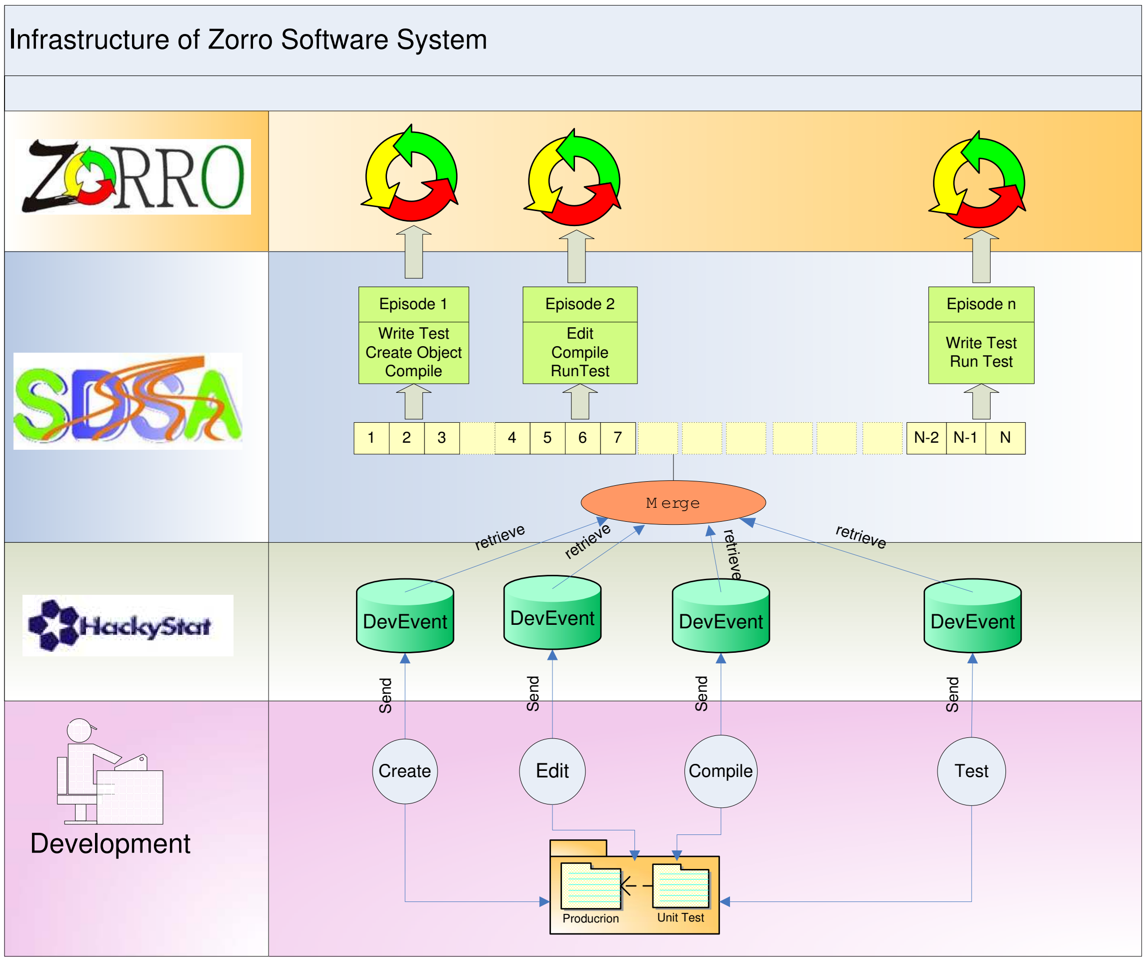


Figure 1.1. Zorro Infrastructure

inference. The SDSA is an extension of Hackystat, and it can be used to study low-level software processes.

1.3.1 SDSA: A Framework of Development Stream Analysis

The SDSA uses software metrics of development activities collected by Hackystat sensors to monitor, identify and characterize high-level development behaviors with the support of JESS [22], a rule-based system in Java. Figure 1.2 illustrates the data models and work flow of SDSA. The data

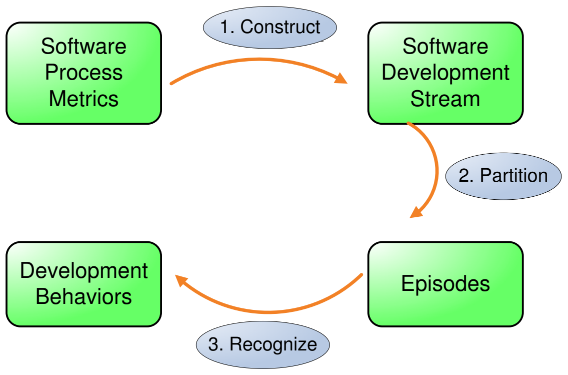


Figure 1.2. SDSA Framework

models of SDSA include “software development stream”, “episode”, and “development behavior”. A linear work flow connects software metrics and these three data models together in SDSA. First, SDSA processes the software metrics of development activities collected by Hackystat sensors. After reducing software process metrics into development activities, SDSA organizes development activities of a same type into a time-series event stream, which is a sub stream of the software development stream. SDSA assembles different event streams together to construct a “software development stream”, which is also a time-series. Second, due to the complexity of digesting long development streams with heterogeneous development activities, SDSA uses a partition technique. A development stream can be partitioned into many episodes delimited by characteristic development activities. The episode is in turn a time-series collection of development activities. Third, SDSA includes a driver and interface to recognize the development behaviors in episodes using JESS.

SDSA is tool and process independent. It can be instantiated to measure a software development method or low-level software process.

1.3.2 Zorro Software System

Zorro is an instantiation of the SDSA framework for TDD. It extends SDSA at three points:

1. Development Stream Construction

   As Figure 1.1 illustrates, Hackystat sensors collect a variety of software metrics, and SDSA can construct a development stream with all of them. But this is not desirable because some development activities might not be relevant to the development method or process under study. Using TDD as an example, debugging is not interesting because it is not part of the process of TDD. In a nutshell, the essential development activities required by Zorro for TDD behavior inference are:

   - editing activities including document, production and unit test editing,
   - buffer transition activities,
   - refactoring activities including addition, deletion, renaming, and moving of object components,
   - unit testing activities,
   - compilation activities.

   Respectively, Zorro constructs the development streams of TDD with event streams including “EditStream”, “BuffTransStream”, “RefactoringStream”, “UnitTestStream”, and “CompilationStream”.

2. Tokenization

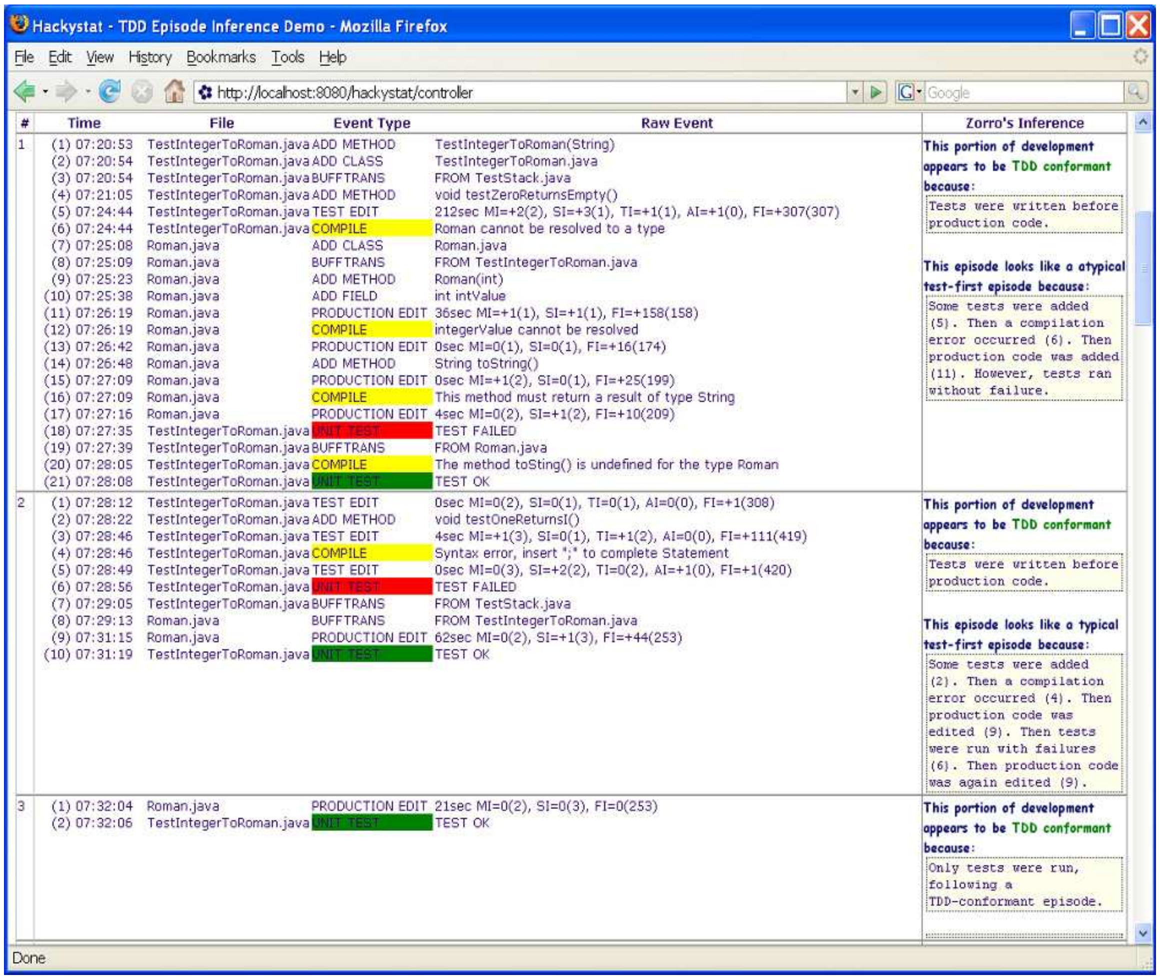
   Zorro uses the “test-pass” tokenizer, which partitions the TDD development stream into a set of episodes that are delimited by successful unit test invocations.

3. Development Behavior Recognition

   In Zorro, I defined a set of specific rules for TDD according to Beck [5, 6] and others who have described the practices of TDD. The “test-pass” episodes are categorized as “test-first”, “refactoring”, “test-addition”, “regression”, “code-production”, “test-last”, “long”, or “unknown”. Chapter 4 has the detailed description of the episode classification schema.

After inferring development behaviors in episodes and categorizing them, Zorro uses the classification results as well as the context of episodes to reason the conformance of TDD. Figure 1.3

is an excerpt of Zorro's TDD inference result for an experienced TDD developer. This experienced

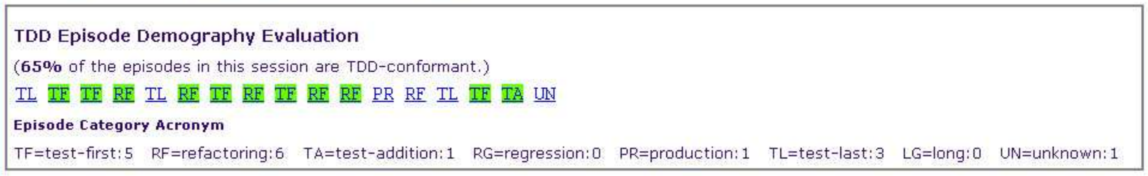

Hackystat - TDD Episode Inference Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/hackystat/controller
Google
Done

#	Time	File	Event Type	Raw Event	Zorro's Inference
1	(1) 07:20:53	TestIntegerToRoman.java	ADD METHOD	TestIntegerToRoman(String)	**This portion of development appears to be TDD conformant because:** Tests were written before production code. **This episode looks like a atypical test-first episode because:** Some tests were added (5). Then a compilation error occurred (6). Then production code was added (11). However, tests ran without failure.
	(2) 07:20:54	TestIntegerToRoman.java	ADD CLASS	TestIntegerToRoman.java	
	(3) 07:20:54	TestIntegerToRoman.java	BUFFTRANS	FROM TestStack.java	
	(4) 07:21:05	TestIntegerToRoman.java	ADD METHOD	void testZeroReturnsEmpty()	
	(5) 07:24:44	TestIntegerToRoman.java	TEST EDIT	212sec MI=+2(2), SI=+3(1), TI=+1(1), AI=+1(0), FI=+307(307)	
	(6) 07:24:44	TestIntegerToRoman.java	COMPILE	Roman cannot be resolved to a type	
	(7) 07:25:08	Roman.java	ADD CLASS	Roman.java	
	(8) 07:25:09	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(9) 07:25:23	Roman.java	ADD METHOD	Roman(int)	
	(10) 07:25:38	Roman.java	ADD FIELD	int intValue	
	(11) 07:26:19	Roman.java	PRODUCTION EDIT	36sec MI=+1(1), SI=+1(1), FI=+158(158)	
	(12) 07:26:19	Roman.java	COMPILE	integerValue cannot be resolved	
	(13) 07:26:42	Roman.java	PRODUCTION EDIT	0sec MI=0(1), SI=0(1), FI=+16(174)	
	(14) 07:26:48	Roman.java	ADD METHOD	String toString()	
	(15) 07:27:09	Roman.java	PRODUCTION EDIT	0sec MI=+1(2), SI=0(1), FI=+25(199)	
	(16) 07:27:09	Roman.java	COMPILE	This method must return a result of type String	
	(17) 07:27:16	Roman.java	PRODUCTION EDIT	4sec MI=0(2), SI=+1(2), FI=+10(209)	
	(18) 07:27:35	TestIntegerToRoman.java	UNIT TEST	TEST FAILED	
	(19) 07:27:39	TestIntegerToRoman.java	BUFFTRANS	FROM Roman.java	
	(20) 07:28:05	TestIntegerToRoman.java	COMPILE	The method toSting() is undefined for the type Roman	
	(21) 07:28:08	TestIntegerToRoman.java	UNIT TEST	TEST OK	
2	(1) 07:28:12	TestIntegerToRoman.java	TEST EDIT	0sec MI=0(2), SI=0(1), TI=0(1), AI=0(0), FI=+1(308)	**This portion of development appears to be TDD conformant because:** Tests were written before production code. **This episode looks like a typical test-first episode because:** Some tests were added (2). Then a compilation error occurred (4). Then production code was edited (9). Then tests were run with failures (6). Then production code was again edited (9).
	(2) 07:28:22	TestIntegerToRoman.java	ADD METHOD	void testOneReturnsI()	
	(3) 07:28:46	TestIntegerToRoman.java	TEST EDIT	4sec MI=+1(3), SI=0(1), TI=+1(2), AI=0(0), FI=+111(419)	
	(4) 07:28:46	TestIntegerToRoman.java	COMPILE	Syntax error, insert ";" to complete Statement	
	(5) 07:28:49	TestIntegerToRoman.java	TEST EDIT	0sec MI=0(3), SI=+2(2), TI=0(2), AI=+1(0), FI=+1(420)	
	(6) 07:28:56	TestIntegerToRoman.java	UNIT TEST	TEST FAILED	
	(7) 07:29:05	TestIntegerToRoman.java	BUFFTRANS	FROM TestStack.java	
	(8) 07:29:13	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(9) 07:31:15	Roman.java	PRODUCTION EDIT	62sec MI=0(2), SI=+1(3), FI=+44(253)	
	(10) 07:31:19	TestIntegerToRoman.java	UNIT TEST	TEST OK	
3	(1) 07:32:04	Roman.java	PRODUCTION EDIT	21sec MI=0(2), SI=0(3), FI=0(253)	**This portion of development appears to be TDD conformant because:** Only tests were run, following a TDD-conformant episode.
	(2) 07:32:06	TestIntegerToRoman.java	UNIT TEST	TEST OK	

Figure 1.3. Demo of Zorro's TDD Inference

developer solved the Roman numeral conversion problem (Appendix C) using TDD in the Eclipse IDE. The Hackystat Eclipse sensor was installed to instrument the development process to collect development activities. Zorro partitioned them into 16 episodes using the "test-pass" tokenizer, and inferred the process conformance of TDD. In the table illustrated in Figure 1.3, the last column contains the reasoning process and result. According to this table, the first three episodes are all TDD conformant. The first episode is "test-first" but atypical because production code was not edited to make test pass after the test invocation failed at (18). The second episode is "test-first" due to the perfect match of development activities to the red/green/refactor metaphor. The last episode is "regression' because no progress was made although the production code was edited. Based upon the reasoned development behaviors and the context of episodes, Zorro inferred that all the three episodes are TDD compliant.

With the automated software metrics collection and inference of TDD, a lot of useful analyses that once were thought impossible have become plausible. A handful of analyses (see Chapter 4) were implemented in Zorro. Figure 1.4 illustrates one of them, the "TDD Episode Demography" analysis. This analysis provides an overview of a TDD programming session, which is partitioned


TDD Episode Demography Evaluation

(**65%** of the episodes in this session are TDD-conformant.)

TL TF TF RF TL RF TF RF TF RF RF PR RF TL TF TA UN

Episode Category Acronym

TF=test-first:5 RF=refactoring:6 TA=test-addition:1 RG=regression:0 PR=production:1 TL=test-last:3 LG=long:0 UN=unknown:1

Figure 1.4. TDD Episode Demography

into 17 episodes, and 65% of them are TDD compliant. Note that

- each small box with a two-letter acronyms represents a single episode,
- TDD-conformant episodes are shown in green. Non TDD-conformant episodes are transparent.

For the TDD programming session illustrated in Figure 1.4, if Zorro was not used, the developer would falsely claim that he/she complied with TDD. In fact, as it turned out, the developer did not conform to the idealized TDD process all the time. According to Zorro's inference, 65% percent of the episodes in this session are TDD-Conformant, and some episodes are "test-last". In addition to reporting the compliance of TDD, the "TDD Episode Demography" analysis can also be used to look for the development patterns. Episodes are ordered by time when they occurred. The researchers and developers can retrospectively review the development process for training or improvement.

Besides typical analyses such as "TDD Episode Demography", I also implemented a group of telemetry analyses. The Software project telemetry [40, 79] was developed by Qin Zhang in the Collaborative Software Development Lab at the University of Hawaii. It can aggregate metrics data together to perform daily, weekly, or monthly analyses of the software metrics to support in-process software project management and decision makings.

Some goals of this dissertation research are to assist the education, training, improvement, and empirical evaluations of TDD. Software project telemetry is an infrastructure that can be used by Zorro to pursue these goals. In Zorro, I have already defined telemetry streams including "TDD percent of development time", "TDD percent of episodes", "development time ratio of test and production code", "size ratio of test and production code", and so forth. Figure 1.5 is a weekly telemetry chart showing percentage of TDD development and test coverage. From the week of Sep

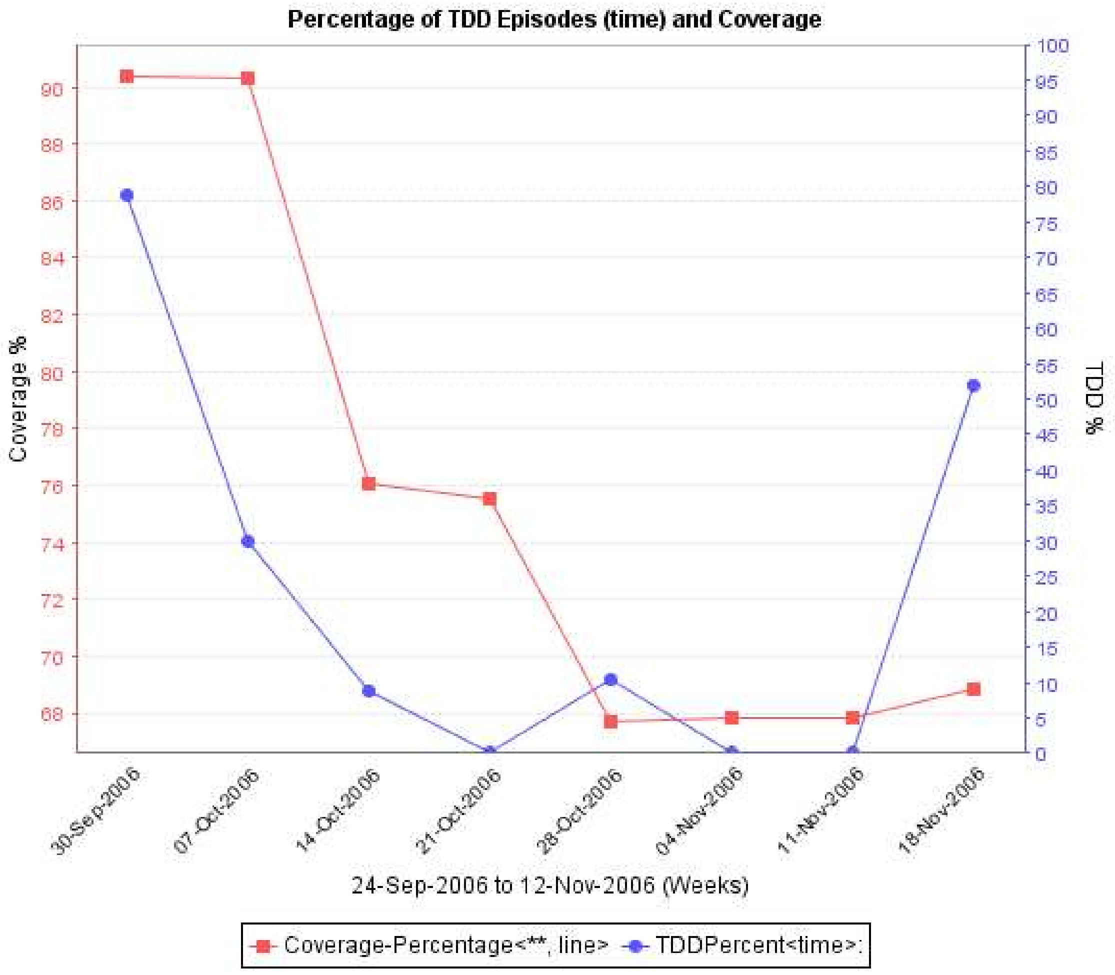


Figure 1.5. Proportion of TDD vs Test Coverage

30, 2006 to the week of Nov 18, 2006, I worked on the Zorro software system and implemented Zorro's web validation interface. I used TDD in my development. Due to the fact that testing web interfaces requires a lot of additional effort, my percent of TDD development dropped down significantly in that period. As a result, the test coverage of Zorro dropped from above 90% to below 70% over the course of eight weeks software development.

The synergy of Zorro and telemetry allows practitioners and researchers to improve the practice and research of TDD with no additional overhead. Both systems are automated based on Hackystat's automatic software metrics collection with sensors attached to the development environment tools.

1.4 Research Statement

In this dissertation, I created the Zorro software system to study the conformance of Test-Driven Development in practice with the aids of the Hackystat framework and the Software Development Stream Analysis (SDSA) framework that I have developed. The design of the Zorro software system is a combination of bottom-up and top-down methods. The development environment tools are instrumented by Hackystat sensors to collect software process and product metrics. A variety of software metrics are abstracted into development activities and then are merged together to form the time-series software development stream, which is then partitioned into small chunks named "episode". This portion is bottom-up. Given a software development method or low-level software process, the process description, guideline, and knowledge can be translated into a set of rules. The rules can be used to evaluate episodes partitioned from the software development stream. This portion is top-down.

The success of Zorro relies on the software metrics collection capability as well as the comprehension of TDD. The software metrics that are related to TDD must be collected. In the long run, it is important for the software development community to reach some kind of consensus on an appropriate definition (or definitions) for TDD. Until there is concrete experience from Zorro, this consensus is not feasible. The development of Zorro is thus iterative and progressive. Figure 1.6 illustrates the time line of this research project including a few milestones. I have conducted three empirical evaluations in my development process of Zorro to investigate whether Zorro can collect enough software metrics and how well it can infer TDD compliance. The Eclipse Screen Recorder (ESR, [17]), an Eclipse plug-in that can record development activities in the Eclipse IDE, was developed to assist the evaluations. ESR can capture Eclipse screen at the rate of 1 frame per second, and thus it provides high fidelity movies for the purpose of validation. The last empirical case study

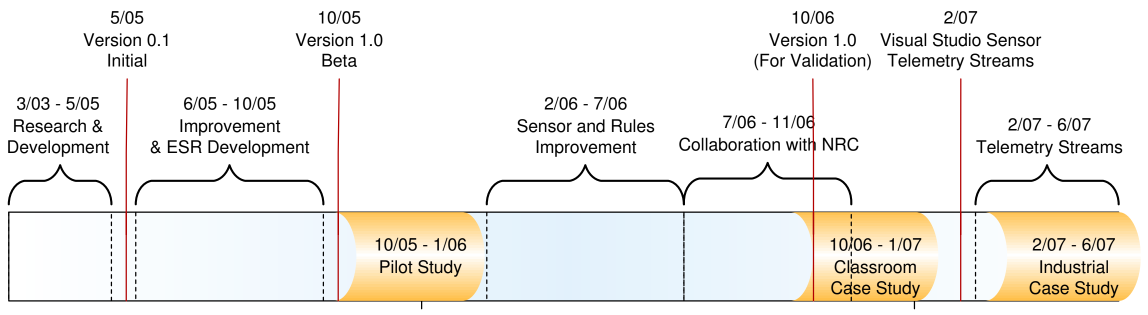


Figure 1.6. Development Timeline of the Zorro Software System

was conducted off-site in Norway to investigate what values Zorro can provide for researchers and project managers.

1.5 Empirical Evaluations

Three longitudinal case studies — a pilot study, a classroom case study, and an industrial case study were conducted to empirically validate the Zorro software system in this dissertation research. The primary goal of these studies was to validate Zorro's software metrics collection as well as TDD inference abilities. The secondary goal was to investigate how useful Zorro would be. An additional goal was to investigate how the metrics collection and inference rules can be improved.

1.5.1 Pilot Study

After researching related work on TDD and stream analysis techniques, I designed and implemented the Zorro software system based upon my observation and analysis of TDD development patterns. By the Spring 2005, I had solved all the difficulties of metrics collection in the Eclipse IDE as a pilot, implemented the SDSA framework, and developed the first set of TDD recognition rules.

I refined the initial version of Zorro in the Summer 2005, and conducted a pilot validation study in the Fall 2005. Six experienced Java programmers participated in this study. Each participant developed a implementation of the stack data structure (Appendix A) using TDD in the Eclipse

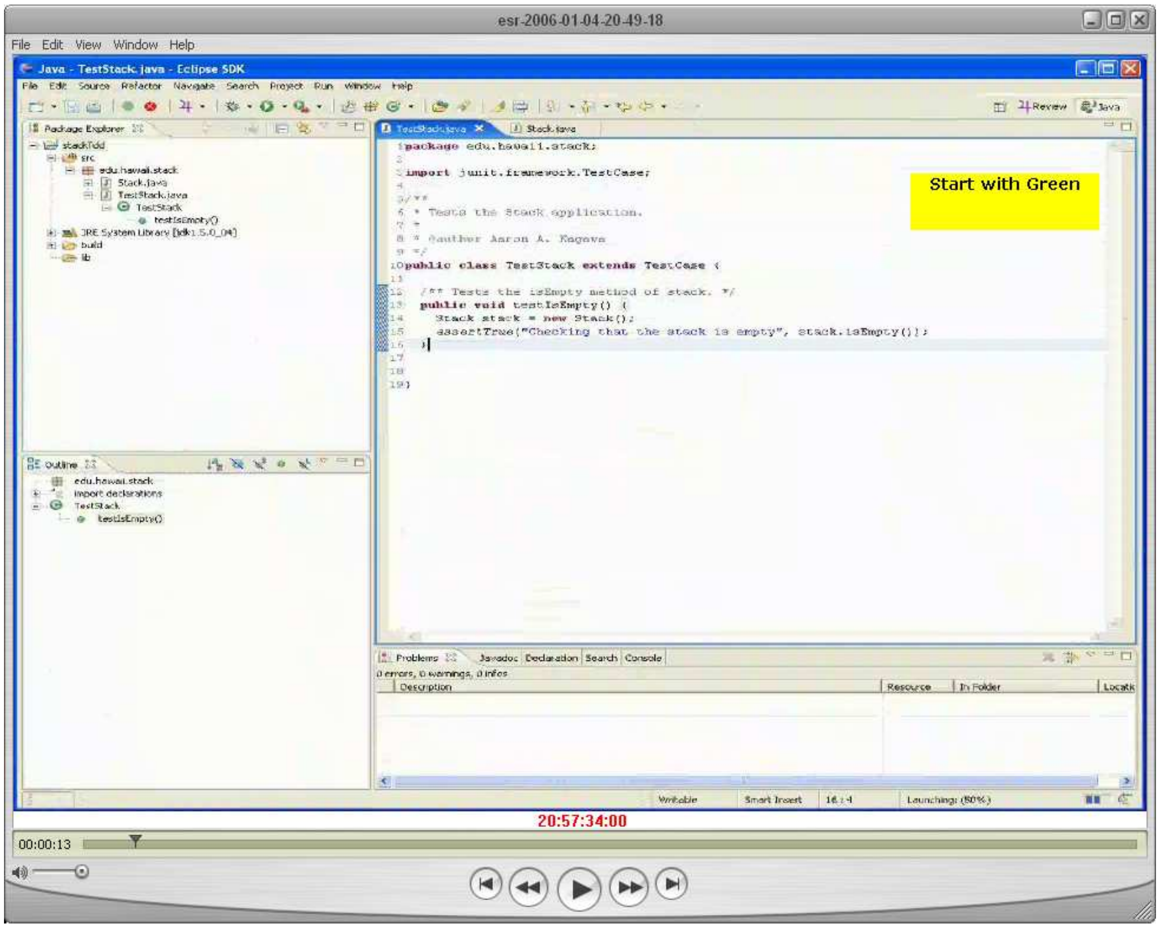
IDE, which was instrumented by the Hackystat Eclipse sensor. I compared Zorro's inference to an independently collected source regarding their development behaviors.

One approach to independent data collection would be to have an observer watching developers as they programmed, taking notes as to what development behavior they are conducting and whether they are pertaining to TDD or not. I considered this but discarded it as unworkable: given the rapidity of development activities in TDD, it would be very hard for an observer to notate all of the TDD-related development activities that can occur literally within seconds of each other. Therefore, I took another approach by developing the Eclipse Screen Recorder [17]. The ESR generates a QuickTime movie containing time-stamped screen shots of the Eclipse Window at the 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 a visual record of development behaviors that can be manually compared to the Zorro analysis using the timestamps for validation purpose. Figure 1.7 is a screen copy showing the recorded development process movie played in the QuickTime Pro software [58].

The participants spent 28 to 66 minutes on the programming task. Zorro partitioned the overall development efforts into 92 episodes, out of which 86 were classifiable; 6 were unclassifiable. It classified 76 out of 86 episodes correctly resulting in classification accuracy rate 88.4%.

The pilot study showed that Zorro is promising at inferring high-level development behaviors with low-level development activities collected as software metrics at least in a simple environment setting. The pilot study also showed that the metrics collection missed some information that led to inference errors.

In the Spring 2006, I improved the Hackystat Eclipse sensor for metrics collection, and refactored Zorro's inference rules. In the Summer 2006, under the suggestion of Dr. Philip Johnson, I went to National Research Council of Canada (C-NRC), where I collaborated with Dr. Hakan Erdogmus, a senior agile process researcher who pioneered the idea of automated TDD conformance inference. The collaborative research at the C-NRC and the follow-up collaboration in the Fall 2006 resulted in a classroom case study and an industrial case study in the Spring 2007.



```
package edu.hawaii.stack;

import junit.framework.TestCase;

/**
 * Tests the Stack application.
 *
 * @author Aaron A. Kagawa
 */
public class TestStack extends TestCase {

  /** Tests the isEmpty method of stack. */
  public void testIsEmpty() {
    Stack stack = new Stack();
    assertTrue("Checking that the stack is empty", stack.isEmpty());
  }


}
```

Figure 1.7. Analysis of QuickTime Video

1.5.2 Classroom Case Study

I, the author of Zorro, compared the recorded movies with Zorro's inference to validate Zorro's metrics collection and TDD development inference in the pilot study. I could be biased both at judging what software metrics are necessary, as well as at inferring development behaviors from the observed activities in the ESR movies. One man's subjective judgment, especially the one from author himself or herself, perhaps is not a valid measure in a case study [78]. This is notably called "construct validity" problem according to [78]. Using multiple data sources, establishing the chain of evidence, and having key informants review draft case study report are three viable tactics that a researcher can use to avoid the construct validity problem. Thus, in the Fall 2006, I developed a web interface to collect participant's comments, the third data source.

In the Fall 2006, I conducted the classroom case study in the software engineering classes at the University of Hawaii. The experimental design of this study is very close to the pilot study. Participants also developed using TDD in the Eclipse IDE with the instrumentations of the Hackystat Eclipse sensor and ESR. The differences were that the Hackystat Eclipse sensor was more robust and the new problem (Bowling Score Keeper at Appendix E) was much harder than the stack data structure (Appendix A).

Eleven students from the software engineering classes voluntarily participated in this study. The participants developed in TDD for 90 minutes, followed by a 10 minute interview and a 20 minutes Zorro inference validation session. In the interview, I asked them questions regarding their opinions on unit testing and Test-Driven Development. A digital voice recorder was used in the interview and in the following validation session for their verbal comments.

The classroom case study data analysis supported the claim that Zorro's metrics collection is as good as ESR, if it is not better. Zorro's TDD compliance inference has two steps. It infers development behaviors in episode first, and then uses the inferred results and context to reason the conformance of TDD. The video analysis validated that Zorro inferred episode development behaviors with 70.1% accuracy and TDD compliance with 89.1% accuracy. The third data source, cross-validation with participant comments, is only slightly different from my video observation analysis.

The participant interview analysis suggested that unit testing is good at yielding high quality software but majority of participants (7 out of 10) admitted that they did not test enough. Perhaps

TDD should be used to improve both developers' confidence and software quality. The data analysis also suggested that TDD is hard to do although half of participants like to develop software using TDD in the future. Therefore, providing a tool such as Zorro to assist practice of TDD has the potential to help beginners. The usefulness evaluation conducted in this study suggested that some of Zorro's TDD analyses are useful for this purpose.

1.5.3 Industrial Case Study

On one hand, I planned the classroom case study for an extended validation after the pilot study. On another hand, I worked with Dr. Philip Johnson and Dr. Hakan Erdogmus to solicit the collaboration with other researchers and practitioners who are also interested in empirical study of TDD. The Zorro demo [80] was developed to demonstrate how Zorro works and what analyses it provides. This effort led to collaboration with Dr. Geir Hanssen and Dr. Tor Erlend Fægri from SINTEF ICT of Norway. They are performing research on the effectiveness of TDD, in contrast to Test-Last Development, the opposite side of TDD. They found that collecting the information about TDD is very hard and Zorro has the potential to provide higher quality information about TDD.

A European software company that provides a packaged software product for marketing and customer surveys [27] is the participant of this industrial case study. However, the development tool is Visual Studio .NET Team Edition and the programming language is C#. We rapidly developed a Zorro compatible Visual Studio sensor for this industrial case study. A Hackystat server is installed on a Windows 2000 server provided by the company. I remotely managed the server for this industrial case study. Out of 20 participants, 12 were in the TDD project and the remaining 8 were in the non-TDD project. Unfortunately, the participation in this study was very low. As to my report written at the end of March 2007, 25% of developers installed the sensor and collected development data, 25% of developers installed the sensor but did not update it as requested, 25% of developers installed the sensor but the sensor did not send any data to the server after a pilot, and the rest 25% of developers did not install the sensor according to my observation of the Hackystat server status. The project manager indicated that Zorro's inferred development behaviors did not agree with what developers actually did, but provided no detailed information. It turned out that it is much harder to have industry participants install the sensor, and it is also hard to relate collected data to actual development behaviors, particularly if the study had to be conducted remotely. However, Dr. Geir

Hanssen and Tor Erlend Fægri have expressed interests to continue using Zorro in the future studies. This case study served as a pilot test only.

1.6 Contributions

My contributions include the SDSA framework, the Zorro software system, and the systematic empirical evaluation of Zorro:

1. **SDSA Framework**

   The Software development is a very complicated process including a series of continuous development activities, yet not independent of each other. The SDSA framework abstracts continuous, interwoven development activities collected by Hackystat sensors in a programming session into a software development stream. The software development stream is a linear, time-series data structure.

   A development stream can be partitioned by SDSA using tokenizers into episodes, another abstract data type representing a micro-iteration of a software process. Tokenizers are expandable and selectable. A different set of tokenizers can be applied according to the studied software process.

   The SDSA framework characterizes development behaviors in episodes using JESS, a rule-based system in Java. A classifier interface is provided in SDSA to flexibly supply inference rules. Moreover, the rules can be changed on the fly.

   In my thesis research, I instantiated the SDSA framework on Test-Driven Development (TDD), and the system resulting from this work is the Zorro software system that can automatically infer the development behaviors and the compliance of TDD. This research work demonstrated that the SDSA framework has the potential to be useful for researching other low-level software processes.

2. **Automated Recognition of TDD with Zorro**

   Zorro recognizes TDD automatically with the software metrics collected by Hackystat sensors in the IDEs. The contributions of Zorro's implementation include the enhanced Eclipse and Visual Studio .NET sensors, a suite of TDD recognition rules, and many useful analyses for understanding TDD in practice.

The Eclipse and Visual Studio .NET Team Edition are two IDEs that are Zorro compatible. The sensors of these two IDEs collect a variety of process metrics such as editing, refactoring, testing, compilation, and so on. Many useful product metrics such as methods, statements, test cases, and assertions are collected too. The progressive changes of software product metrics are very useful information for software engineering research.

The recognition rules are from the descriptions of many well-known TDD practitioners including Beck [6], Doshi [15], and Erdogmus, and my grounded observation of TDD in practice.

Many analyses were developed to display Zorro's inference processes and results, report different aspects of TDD such as episode duration distribution. The TDD telemetry streams were developed to support the in-process decision makings for software project management.

3. **Empirical Evaluations**

   My contribution to the empirical evaluations are:

   (a) ESR [17], a software system that can record Eclipse usage;

   (b) The availability of the Zorro software system for use by other researchers under an open source license;

   (c) The experimental method exemplified by the case studies, which shows how to conduct research on TDD that does not suffer from the process conformance problem;

   (d) The actual results of the case studies, which show (i) the Zorro can identify TDD, that (ii) users found Zorro analyses to be useful in certain cases and not in others, and (iii) that industrial case studies on TDD are more difficult than classroom case studies, as it is more difficult to get developers to install sensors, and more difficult to relate data back to their development.

1.7 Dissertation Structure

This thesis is organized into the following chapters.

- Chapter 1 introduces the TDD challenges and the motivation of this research.
- Chapter 2 presents the related research work.

- Chapter 3 describes the SDSA framework in details.
- Chapter 4 describes the Zorro software system in details.
- Chapter 5 briefs the research questions and methodology of this dissertation.
- Chapter 6 reports the pilot study data analysis in details.
- Chapter 7 reports the classroom case study data analysis in details.
- Chapter 8 reports the industrial case study conducted off-site.
- Chapter 9 synthesizes the results from the empirical case studies, presents the conclusions of this research, and discusses the future work.

Chapter 2

Related Work

Test-Driven Development (TDD), a core practice of Extreme Programming, has been widely adopted by software industry and studied by software engineering researchers. Industry practitioners have put increasing effort into evaluating and understanding TDD in recent years. Many books [6, 3, 51, 47, 30] directly related to TDD have been published. XUnit[59, 77], the foundation of TDD, has been ported to more than 30 languages. Development tools such as Eclipse, NetBeans, and Visual Studio have been enhanced to support unit testing, which makes it easier for practitioners to develop software in TDD. The community of TDD [65, 63] is continuously growing, and some enthusiastic practitioners [48, 2, 62, 69, 57] even write about their personal experiences in their blogs. In addition to this industrial interest, software engineering researchers have begun studying TDD as an enabling software development method. Both pedagogical[50, 18, 53, 19, 44] and industrial [23, 49, 25, 76, 7] evaluations of TDD have been conducted in the last few years.

So far, software engineering researchers have focused most of their energy on the outcomes that applying TDD brings to software products and software developers. However, compared to the claims made by practitioners, research findings of TDD on software quality and developer productivity are mixed. In fact, much of the research work on TDD suffers from the threat of “construct validity” [74] because of the “process conformance” problem. Wang and Erdogmus define process conformance as “the ability and willingness of subjects to follow a prescribed process”. Janzen and Saiedian warn that the inability to accurately characterize process conformance is harmful to TDD research, and that it is so hard to measure the usage of a development method such as TDD that current reports on adoption of TDD are not valid. Surveys are often used to measure the adoption of TDD, but only those who are much in favor or much opposed to it will respond. Janzen and

Saiedian concluded that the combination of popularity of XP, JUnit and Eclipse likely implies a certain degree of adoption of TDD [31]. However, this is a very indirect measure.

Some of research work [11, 34, 45, 74, 75] has been done on software process compliance using development activities and software artifacts collected from the development process. In my dissertation research, I focus on studying the process conformance of low-level software processes and Test-Driven Development in particular.

Janzen claimed that TDD is a kind of software development method, not a process model, and that it has emerged out of a particular set of process models [31]. In contrast, Beck and Cunningham, the pioneers of TDD, put it this way: “test-first coding is not a testing technique but is rather about design.” [5] If TDD is a design technique and it drives the implementation of product code, then classifying it as a software process sounds reasonable. In my research, I have characterized practices such as Test-Driven Development and Personal Software Process (PSP) as low-level software processes. A common characteristic of low-level software processes is that they are defined by many frequent and rapid short-duration activities. Unlike high-level and long duration phases such as “requirement analysis” that might last weeks to months, the activities in low-level software process such as “refactor class Foo to extract interface IFoo” may take only seconds to a few minutes [45].

This chapter begins with a detailed introduction to TDD, followed by a discussion of TDD empirical studies. The research results are mixed because of differences in experiment settings, and the empirical studies suffer from construct validity because of the process conformance problem of TDD. In the second part, I present other related work on automated process conformance in contrast to my research that is built on the automated software metrics collection machinery of Hackystat [35, 37, 36, 38].

2.1 Test-Driven Development: A Short Introduction

“*Test-first coding isn’t new. It’s nearly as old as programming.*”

— Kent Beck

Test-Driven Development[6] is a software development best practice popularized by Extreme Programming [32, 4]. It has two basic rules: “(1) Write new code only if an automated test has failed; (2) Eliminate duplication." Kent Beck, the pioneer of Test-Driven Development, stated that there is an implicit order of programming in TDD [6]:

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. Refactor - Eliminate all the duplication created by merely getting the test to work.

The key characteristic of TDD is "test-first", with which developers should always write a test first according to the requirement, and then implement the functional code to make the test pass. Because a test is always created first to drive the design and implementation, TDD used to be called Test-First Design (TFD) or Test-First Development (TFD) [4]. My opinion is that "test first" is better than "test driven" with respect to describing the order of test and production coding activities. Therefore, in the rest of this document, I will use "test first" when it is necessary to emphasize the order of programming in TDD, otherwise there is no difference between "test first" and "test driven".

Test first is as old as programming, and has been used for decades [76, 4, 5]. Beck recalled that his first programming experience was actually in test-first using output and input tapes [5]. Indebted to the philosophy and popularity of Extreme Programming (XP), Test-Driven Development has emerged as a notable best practice for software development [4, 32].

XP is a light weight software development methodology that is intended for use when confronted by vague and rapidly changing requirements [4]. It is rooted in observations on repeated best practices in software development. The philosophy of XP is to take common-sense principles and practices to an "extreme" level [4, 32]:

- If code reviews are good, we'll review code all the time (Pair Programming).
- If testing is good, everybody will test all the time (unit testing), even the customers (Functional Testing).
- If design is good, we will make it part of everybody's daily business (Refactoring).
- If simplicity is good, we will always leave the system with the simplest design that supports its current functionality (the simplest thing that could possibly work).
- If architecture is important, everybody will work defining and refining the architecture all the time (Metaphor).
- If integration testing is good, then we will integrate and test several times a day (Continuous Integration).

- If short iterations are good, we will make the iteration really, really short — seconds and minutes and hours, not weeks and months and years (Planning Game).

Figure 2.1 illustrates the supporting network of twelve XP practices [4]. XP proposes TDD as

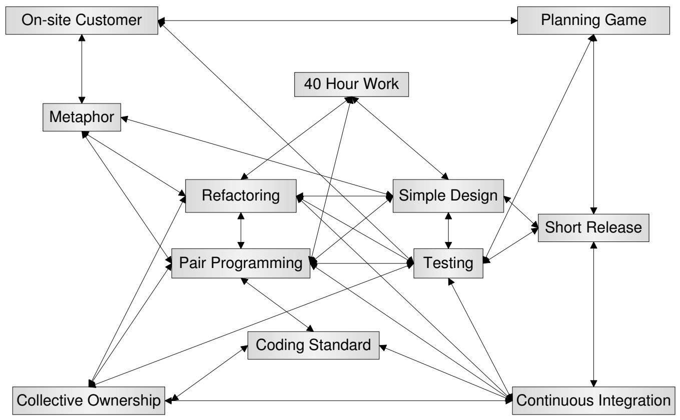


Figure 2.1. Network of Extreme Programming Practices[4]

the glue to hold the process together. TDD helps create a comprehensive suite of unit tests such that developers have the courage to consistently refactor code into a much simpler design.

Though born as a practice of XP, TDD is often thought as an independent software development method. It can be used by practitioners and organizations that do not or partially practice XP [6, 65, 63]. Many TDD training workshops [66, 67, 61, 68, 64, 52] have been provided by experienced practitioners and consulting companies. An informal survey [73] conducted by the Method and Survey magazine found that 46% of the studied software organizations perform unit testing informally, 41% of the studied organizations document their unit test cases, and 14% of the studied organizations use the TDD approach.

2.1.1 Characteristics of TDD

“Never write a line of functional code without a broken test case.”
— Kent Beck

“Test first coding is not a testing technique.”
— Ward Cunningham

Recall that the two basic rules of TDD are to "(1) Write new code only if an automated test has failed; (2) Eliminate duplication." Following the first rule, given a requirement, a TDD developer analyzes it first, and then outlines a To-Do list with a few tasks. The tasks can be very simple but should suffice to enable some development to occur. The developer can start a task that is either easy to do or essential to the problem to be solved [3, 6]. After picking a task, the developer writes a unit test first, and then invokes it. The test invocation might fail because it tests something that does not exist. The failing test drives the implementation of new functional code to make it pass. Often, it is also necessary to run all tests to make sure nothing is broken. The second rule is to eliminate the duplication. If redundancy is introduced in the first step, the developer should refactor either test code or functional code to remove it. It is also necessary to run all tests after refactoring to confirm that nothing is broken. This is an iteration of TDD. A common belief is that TDD iterations should not last more than 10 minutes [16]. After an iteration is over, the developer can cross out the finished task from the To-Do list, and pick another task to begin a new iteration. Additionally, the developer should add new tasks to the To-Do list whenever he or she wants. The characteristics of TDD are :

Test First

Test first is the key characteristic of TDD. Importantly, in TDD, developers should "never write a line of functional code without a broken test case" [5].

Short Iterations

Quickly adding functional code to make test pass is important to TDD. An iteration should last a few seconds to several minutes only. If hours of work is needed to make a test pass, probably the developer should divide the programming task into several sub tasks that can 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 [6].

Rapid Feedback

Unit testing is usually supported by the XUnit[59] framework that is available in most mainstream languages. After new functional code is added, developers can invoke the unit tests to test it right away. The feedback is available within seconds or minutes.

One Ball in the Air at Once

In typical software development, a developer usually carries a baggage with the requirement, system structure design, algorithm, code efficiency, readability and communication with other code etc. Martin Fowler described that the process is like keeping several balls in the air at once (Page 215 in [6]), while the developer only keeps one ball in the air at once and concentrates on that ball properly in TDD. 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.

Always Working Code

The code should be always working in TDD because developers 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.

2.1.2 Benefits of TDD to Software Development

"*I have spent enough time in my career on silly bug-hunting, with TDD those days are gone. Yes, there are still bugs, but they are fewer and far less critical.*" [2]

—Thomas Eyde

The two most notable benefits of TDD are high quality and developer productivity [5, 31] :

High Quality

Probably the most advocated benefit of TDD is the high code quality. Because software quality is hard to measure, practitioners and researchers often use code coverage as the proxy of software quality. The code developed in TDD should be 100% covered since no functional code is created without a unit test.

High Productivity

Functional code and test code are both products. Testing is part of the design process, and it does not take a long time to write a small test. 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. In addition, TDD users claim that the method reduces the overall amount of time spent on debugging, with a resulting increase in overall productivity[76].

2.2 TDD Research Work

Much research work has been conducted on studying important outcomes of TDD such as software quality and developer productivity in recent years. In addition to anecdotal experience reports [24, 49, 76, 44, 18, 7], researchers have run controlled experiments [50, 53, 19] to compare TDD against other development methods such as test last and ad hoc. Depending on whether the test subjects are students or professional developers, the research work can be categorized into academic and industrial studies.

2.2.1 Empirical Evaluation in Academic Settings

Müller and Hanger [50] 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 [18] 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 [44] 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, Ciglarič, Trampuš, and Vidmar [53] designed a controlled experiment 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 that TDD is somewhat different from ITL but the difference is very small.

A more recent study on the effectiveness of TDD conducted by Erdogmus, Morisio and Torchiano [19] used the well-defined test-last and TDD approaches as Pančur did in [53]. This study concluded that TDD programmers wrote more tests per unit of programming effort. More test code tends to increase software quality. Thus, TDD appears to improve the quality of software but TDD group in the study did not achieve better quality on average than test-last group.

2.2.2 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 [24] 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 [49] 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, Maximilien, and Vouk [76] 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, Smith and Miller [25] 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 conducted by Bhat and Nagappan [7] 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.

2.2.3 Discussion of Empirical Evaluation Studies

The research findings of empirical studies are mixed on software quality and developer productivity as shown in Table 2.1. The study conducted in Microsoft [7] and the study conducted in University of Karlsruhe [50] are two extreme cases. In [7], the developers improved software quality up to four times after adopting TDD. In comparison, the TDD group in [50] yielded less reliable programs than the control group. In the following, I will compare the research results according to the differences in experiment settings.

Students vs. Professional Developers

One difference is in population. The studies in academic settings used students as test subjects, while the studies in industrial settings recruited professional developers. Of 10 empirical studies

	Investigator	Study Type	Participants	Software Quality	Developer Productivity
Industrial	George [23]	Controlled Experiment	24	TDD passed 18% more tests	16% more time
	Geras [25]	Controlled Experiment	14	TDD has the edge on quality	No impact
	Maximilien [49]	Case Study	9	50% reduction in defect density	Minimal impact
	Williams[76]	Case Study	9	40% reduction in defect density	No change
	Bhat [7]	Case Study	11	2-4 times reduction in defect density	35% and 15% more time
Academic	Kaufmann [44]	Controlled Experiment	8	N/A	50% improvement
	Edwards [18]	Case Study	59	54% fewer defects	N/A
	Erdogmus [19]	Controlled Experiment	35	No change	Improved productivity
	Müller [50]	Controlled Experiment	19	Less reliable, but better reuse	No change
	Pančur [53]	Controlled Experiment	38	No change	No change

Table 2.1. Research Work of TDD on Software Quality and Developer Productivity

listed in Table 2.1, 5 were conducted in academic settings, and the other 5 were conducted in industrial settings. The studies conducted in industrial settings found evidence of software quality improvements, but often at the cost of decrease of developer productivity. On the contrary, the studies in academic settings often found no software quality improvement but increase of developer productivity.

How come the research findings are so divided among studies conducted within different population? Did professional developers pay more attention to software quality than students? Were students more concerned about productivity than professional developers? A caveat of industrial studies is that they must be beneficial to participants. In [49], Maximilien and Williams proposed TDD as a solution to reduce the defect rate. In [76], Williams, Maximilien and Voulk assigned a dedicated coach to the development team. George and Williams [23] noticed that the control group did not write worthwhile automated tests, partially due to the lack of incentives. Brilliant and Knight [9] note that industry does not perceive a significant benefit from working with academic researchers in many cases. Researchers have to be able to convince industry practitioners of the benefits and provide assistance in order to conduct research in industrial settings. The benefits are important to the participants of industrial studies, otherwise the quality of research would degrade. Geras, Smith and Miller reported that professional developers are hard to recruit when participation is voluntary [25].

Students are generally less experienced at software development when compared to professional software developers. It is unclear whether students are appropriate participants for effectively studying TDD [19]. Müller and Hagner reported that the TDD group produced code with 74% branch converage only, while the control group produced code with 80% branch coverage [50]. Erdogmus, Morisio and Torchiano [19] discussed the causal relationship between research findings and participant skill levels. They found that the students with high skill rate improved productivity more dramatically than the students with low skill rate.

Case Study vs. Controlled Experiment

Another difference between the academic and industry based research is in research methods. Case study and controlled experiment are the two most popular research methods. Of the 10 studies listed in Table 2.1, 6 are controlled experiments and 4 are case studies. Most of the controlled experiments were conducted in academic settings because classroom settings are more amenable to controlled experimentation. Since industry rarely repeats the same project twice, it is hard to have controlled experiments in industrial settings. Three out of five industrial studies were case studies.

In controlled experiments, researchers often isolated test first from TDD to compare it against other development methods such as test last and ad-hoc [50, 53, 19, 23, 25]. Half of the controlled experiments [23, 44, 19] observed productivity improvement in TDD groups, but only two studies found evidence of software quality improvement. On the contrary, the participants in the case studies improved software quality dramatically after adopting TDD, but they also spent more time on development.

Rivalry Development Methods

The third difference of experiment settings is in the development methods that TDD was compared against. Researchers compared TDD against the traditional test-last[44, 25], ad hoc[50, 23], and iterative test last (ITL) [53, 19] methods. TDD did not help to improve software quality when it was compared against ITL. But it helped to improve software quality when it was compared with ad hoc and test last methods. Though TDD developers spent more development time, they also wrote more tests [23, 19] and ran tests more frequently [25].

2.2.4 Process Conformance of TDD

In the previous section, I have discussed the mixed research findings of TDD due to differences in population, research methods and the development methods that TDD was compared to. However, the research findings of studies in the same group were often consistent. For example, all the case studies found evidence of software quality improvement of TDD. Perhaps this phenomenon indicates that there are flaws in the empirical studies of TDD.

Erdogmus, Morisio and Torchiano [19] discussed that a threat to the validity is "process conformance", which is the level of conformance of the subjects to the prescribed techniques. In order to improve process conformance, they informed participants of the importance of following the proper procedures, and conducted a post test survey to filter unconformant data points[19]. Although they first identified process conformance as a threat to the validity of research results of TDD, they were not alone in dealing with it. Müller and Hagner [50] also indicated that their experiment was not technically controlled. During the experiment, they had to ask TDD groups if they were following the test-first process. Pančur, Ciglarič, Trampuš, and Vidmar [53] instrumented Eclipse, the development tool used in their study, to report unit test invocation frequency, results and time taken. A similar instrumentation method was used by Geras, Smith and Miller [25]. However, instrumenting test invocation is a poor measure for process conformance of TDD because developers in test last groups can also run tests frequently. In [23, 76, 49], researchers studied test-first along with pair programming, which served as the process control method. Clearly, none of these methods is reliable at controlling participants' compliance to TDD or the rivalry development methods.

Test first is a key characteristic of TDD, and developers should "never write a line of functional code without a broken test case." [5]. Short iterations, frequent refactoring, one ball in the air at once, and always working code are other characteristics of TDD. Developing software in TDD needs skills, practice, and discipline. On one hand, it is being widely adopted and embraced by software industry. Many developers simply enjoy the "dance" of test first and claim that they are "test-infected", a phrase that is often used by the members of TDD community. On the other hand, the empirical research findings of TDD are often mixed. One of the causes of confused research results is the process conformance problem. In my thesis research, I have focussed on developing an automated machinery to evaluate the TDD conformance. Moreover, this work can be extended to study other low-level software processes as well. In a broader view, my work belongs to the research of automated software process. Some work has been conducted on this direction [11, 10, 33, 34].

2.3 Automated Software Process Research

Software process research is historically top-down. Process programming[42], modeling[8] and simulation[71, 34] are typical research methods for studying software processes. These top-down methods are not usable for studying the process conformance of low-level software processes. They are appropriate when processes are confined and variations are very rare. However, these conditions are not easily met for low-level software processes that define how developers should carry on development activities at the granularity of minutes and hours. Given a low-level software process, the actual process can be very different from the ideal process [14, 55, 37], which in turn leads to the issue of process conformance.

In my research, I used a bottom-up research method. I applied rules on automatically gathered low-level development activities to infer the actual process, and compared the inferred process against the idealized process automatically, enabled by the rule-based system. My work is related to Cook's "Automating Process Discovery and Validation" and Jensen's "Discovering and Modeling Open Source Software Processes" because we all studied the automated evaluation of software processes using low-level development activities and software artifacts.

2.3.1 Automating Process Discovery and Validation

Cook and Wolf [11, 10] 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 in their research. Figure 2.2 illustrates the ISPW 6/7 process they discovered using the Markov Model. The numbered circles in Figure 2.2 are the process states, and the arrows represent the development event data. Although Cook demonstrated that the generated FSM in Figure 2.2 is very close to the actual ISPW 6/7 process, FSM does not look like an ideal technique for automated process discovery and validation. The process FSM can be too complicated to be useful. In [11], the three algorithms RNET, KTAIL and MARKOV generated 15, 20 and 25 states respectively, and the states are interwoven in complicated manners as shown in Figure 2.2. It is very hard to interpret the monolithic state chart without a thorough understanding of Balboa and the adopted software process.

2.3.2 Discovering and Modeling Open Source Software Processes

Jansen and Scacchi [33, 34] 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. But they don't have a working software system except for a prototype implementation of the "requirement and release" process of the open source project NetBeans.

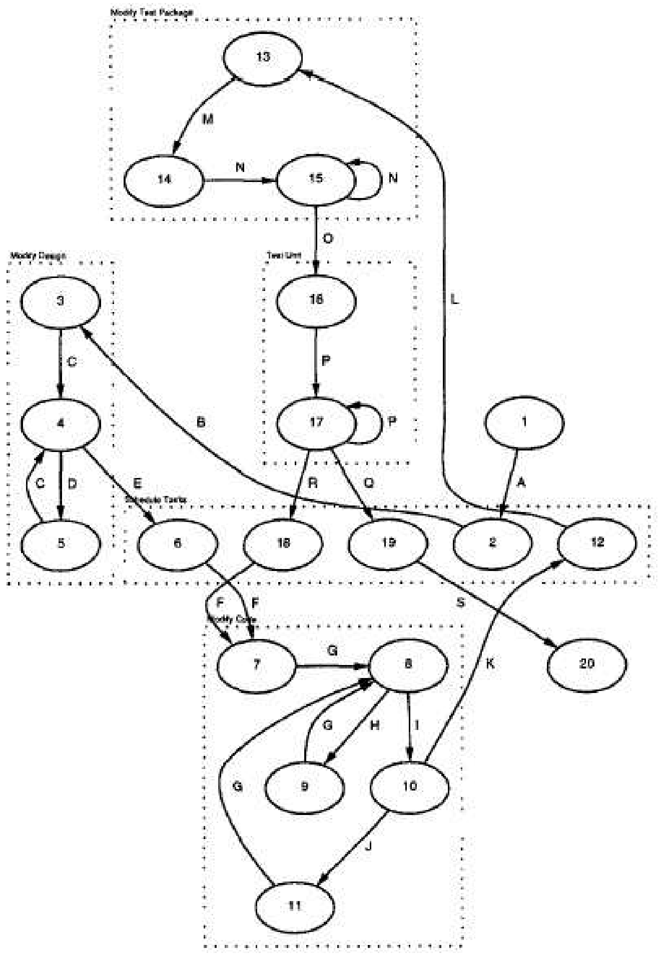


Figure 2.2. Discovery of ISPW 6/7 Process (MARKOV)[11]

2.3.3 Discussion of Automated Process Conformance Research

Automated process conformance is essentially about knowledge discovery and data mining, in which ordered data streams are processed to discover and classify naturally recurring patterns.

Cook and Wolf used artificial neural network with RNET algorithm, MARKOV probability model with Bayesian learning algorithm, and KTAIL algorithm to mine the development event streams for discovering software processes [11]. But generating a monolithic process model represented in FSM does not appear to be a very usable solution for process conformance validation of complicated software processes. In addition, generating FSM is very time consuming according to the performance report [11].

Jansen and Scacchi [33] modeled the requirement and release process of a large open source project using PML. They applied the modeled process to examine the development activities and behaviors. This bottom-up strategy, together with a top-down process meta-modeling, seems more plausible than FSM for dealing with complicated development behaviors.

In my thesis research, after researching the knowledge discovery and data mining algorithms and techniques, I chose a rule-based system to study process conformance of low-level software processes. Instead of asking process experts to inspect the FSM of the executed process as Cook and Wolf did [11], I converted the process knowledge into a set of rules and used them to infer the software development behaviors. My method is very close to Jansen's and Scacchi's approach [33] except that I used rules rather than PML for process descriptions.

An intriguing phenomenon of automated software process research is the scarcity of directly related literature work. Perhaps this is due to the difficulty of collecting the process execution data. Cook and Wolf ignored the problem of data collection in their research and claimed that it is site-specific [11]. Jansen and Scacchi took advantage of rich information on the web and hypothesized that the proposed model can be used for the process deployment, validation and improvement [33].

Fortunately, with the development of sophisticated software metrics collection system such as Hackystat, software development data can be collected automatically and unobtrusively [35, 37, 36, 38]. In my thesis research, based upon low-level development activities collected by Hackystat sensors, I conducted the research of the process conformance of Test-Driven Development.

Wang and Erdogmus [74] argued that the empirical research of TDD suffers from the construct validity problem (as is also the case in some other empirical software engineering research) because it lacks process conformance. They developed a prototype called “TestFirstGauge” to study the process conformance of TDD by mining the in-process log data collected by Hackystat. 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 the indicator of TDD process conformance. This project precedes the Zorro software system[45], and in fact it stimulated our research interest in studying low-level software process conformance. Unlike the prototype implementation of TestFirstGauge in VBA using an Excel spreadsheet, Zorro is integrated into the Hackystat system for automation, reuse, and flexibility using rule-based system [22].

Similarly, Wege [75] 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 will not commit on-going project data at the granularity of seconds, minutes or hours when they develop the software system, making this data collection technique problematic for the purpose of TDD inference. Collecting rapid low-level development activities is a must in order to infer the low-level software process such as TDD automatically.

2.4 Chapter Summary

In this chapter, I introduced Test-Driven Development (TDD), a widely adopted core practice of Extreme Programming. The key characteristic of TDD is to develop software by writing tests first. The software industry has widely adopted TDD and significant efforts have been put into it by practitioners (see Section 2.1). Often, it is claimed that TDD can greatly help to improve software quality and developer productivity. The other claimed benefits of TDD include simple design, programmer confidence, and time saving on maintenance because less time is needed for bug hunting and debugging (see Section 2.1). Much research work has been conducted on TDD about software quality and developer productivity (see Section 2.2), but the research findings are divided. Interestingly, the research results are consistent when the experiment settings were similar. A possible explanation for this phenomenon is process conformance, which is a construct validity threat for the research findings (see Section 2.3).

To address these problems and produce greater consistency in research results, I propose the automated process conformance inference with support from a rule-based system. In this chapter, I compared my research to Cook’s automated process discovery and validation, Jansen’s simulation of open source software project process, Wang’s and Erdogmus’s TestFirstGauge project, and Wege’s automated support for process assessment in TDD (see Section 2.3).

Chapter 3

Software Development Stream Analysis (SDSA)

SDSA is a Hackystat extension that provides a generic framework for organizing various kinds of software metrics received by Hackystat into a form appropriate as input to a rule-based, time-series analysis (Figure 1.1). SDSA supports (1) construction of software development streams, (2) partition of software development streams, and (3) inference of development behaviors.

The SDSA framework supports completely automated analysis, once configured. First, the data collection is automated because SDSA uses software process metrics collected by Hackystat sensors. The sensors collect in-process software metrics automatically and unobtrusively. Second, the construction and partitioning of the software development stream is automated. SDSA reduces software process metrics into development activities for construction of a software development stream, which is then partitioned into episodes. Once a development stream has been partitioned into a sequence of episodes, SDSA can analyze the development behaviors in these episodes and classify them according to whatever process is under analysis. Finally, the inference process is also automated because SDSA uses JESS[22], a rule-based system in Java. Developers can specify part or all of the studied process as a set of rules.

The SDSA framework is designed to be customized for a specific software process of interest. Developers can selectively choose the event streams to merge in the software development stream construction phase (see Section 3.2.1). For a software process, developers can use different tokenizers to partition the development streams (see Section 3.2.2). Also, different rules can be supplied for development behaviors inference (see Section 3.2.3).

This chapter begins with an introduction to the Hackystat framework on which the SDSA framework is built, followed by a description of the SDSA framework itself.

3.1 Hackystat

Hackystat is an open source framework for automated collection and analysis of software engineering process and product metrics. Hackystat supports unobtrusive data collection via specialized "sensors" that are attached to development environment tools. These sensors 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.

Hackystat is tool, environment, and process independent. It does not presume a specific operating system platform, a specific integrated development environment, or a specific software process. It is designed to be extensible. It provides not only generic services such as software metrics collection, persistence and retrieval, and project definition management for conglomerating discrete software metrics from both an individual and other project members, but also an extension mechanism where new modules (sensors or applications) can be added. Applications of the Hackystat framework in addition to SDSA include in-process project management [40], high performance computing [41], and software engineering education [39].

3.1.1 Software Metrics Collection, Persistence, and Retrieval

When developers are programming in a development environment tool with its sensor installed and enabled, the sensor will collect both the process and product metrics unobtrusively. The sensors then send them via SOAP to a web server that hosts Hackystat. The architecture of Hackystat is client-server. The *"clients"* can be development environment such as Emacs, Eclipse, and Microsoft Visual Studio. The *"server"* is the framework itself and its extended applications. On the server-side Hackystat handles metrics data persistence automatically. It stores them in XML formats and implements a self-managed caching mechanism to retrieve them when requested by the applications.

3.1.2 Extension Mechanism

The Hackystat system provides an extension mechanism to support new functionalities including sensor data types (metrics), sensors, applications, documentation, and so forth. Each functionality needs to specify a configuration file in XML saying what it is about and what part of Hackystat it will extend. For instance, if you are developing a new sensor for the Java development environment tool NetBeans, you will specify a sensor definition file such as netbeans.sensor.def.xml. It will include the sensor name and what metrics it collects. The further information on how to implement an extended functionality can be found at the Hackystat home page:*http://www.hackystat.org*.

3.2 SDSA Framework

SDSA is an application extending the Hackystat framework. It organizes various kinds of process metrics data into a time-series software development stream and conducts rule-based analyses. SDSA is data oriented. Figure 3.1 illustrates its data model and work flow.

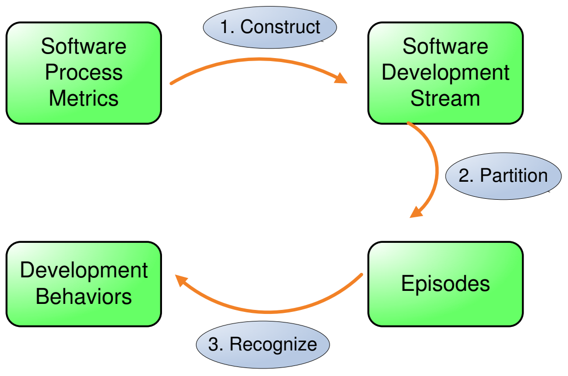


Figure 3.1. Data Model and Work Flow of the SDSA Framework

SDSA models a software development process as a software development stream, which is a time-series data structure consisting of continuous software development activities collected by Hackystat sensors. The development stream is then partitioned into episodes delimited by boundary conditions. Each episode is a data model that represents an atomic components of the software process of interest. Eventually, the goal of SDSA is to recognize the development behaviors embodied within each episode.

The data model breaks down the SDSA framework into three components(Figure 3.1): (1) a software development stream construction subsystem, (2) a software development stream partition subsystem, and (3) a development behavior inference subsystem.

3.2.1 Software Development Stream Construction

SDSA begins with the development stream construction subsystem that translates variety kinds of software process and product metrics into a development stream. Figure 3.2 illustrates the process of software development stream construction.

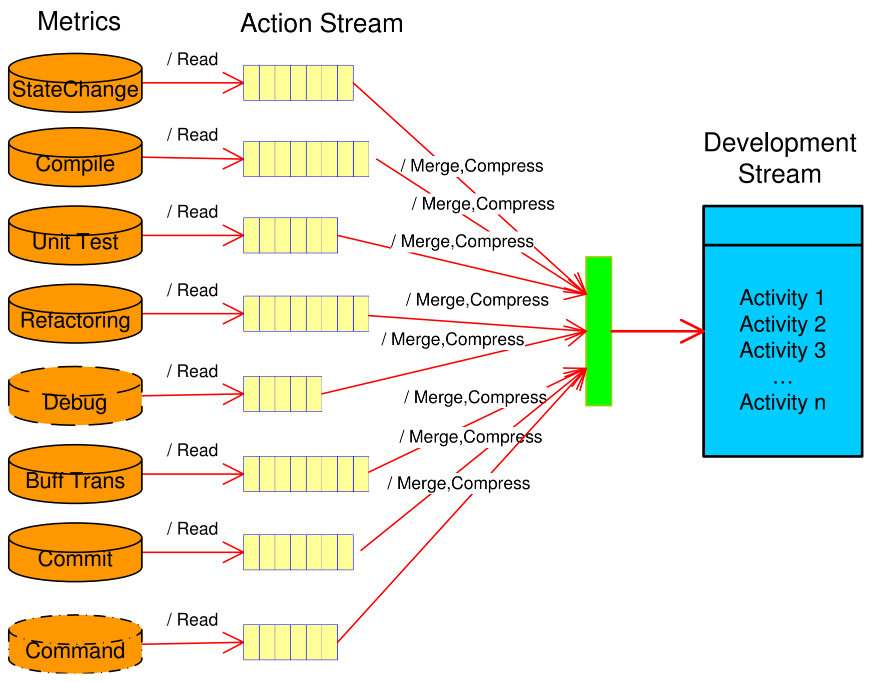


Figure 3.2. Software Development Stream Construction

Hackystat sensors collect the software process and product metrics and abstract them as the structured “sensor data type”. A sensor data entry represents a development activity or an in-process product metric change that has been caused by a development activity. Because sensor data mixes software process and product metrics, SDSA must restructure them as development actions in order to infer the development behaviors. The development action is SDSA's internal representation of the software development activities that are collected by Hackystat sensors.

Development Action

Figure 3.3 is the class diagram of variety of development actions SDSA abstracted. All the development actions are subtypes of the "Action" type. Each action has a clock attribute indicating when it occurs and a duration attribute indicating how much time it takes.

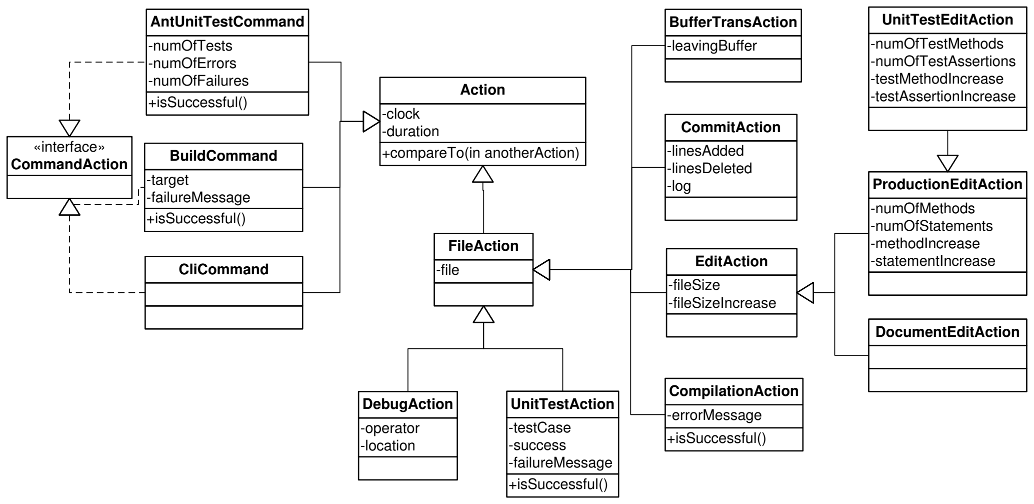


Figure 3.3. Class Diagram of Development Actions

After reading the software metrics of a sensor data type, SDSA transforms them into development actions, and then forms a time-series action stream.

Action Stream

Figure 3.4 illustrates the class diagram of action streams. The "DataStream" is the super class of all action stream classes. Each action stream has the capability of transforming a type of sensor data into a time-series action stream.

In addition to converting sensor data into development actions, an action stream can implement a callback function to compress itself. This function can group a set of continuous sensor data that

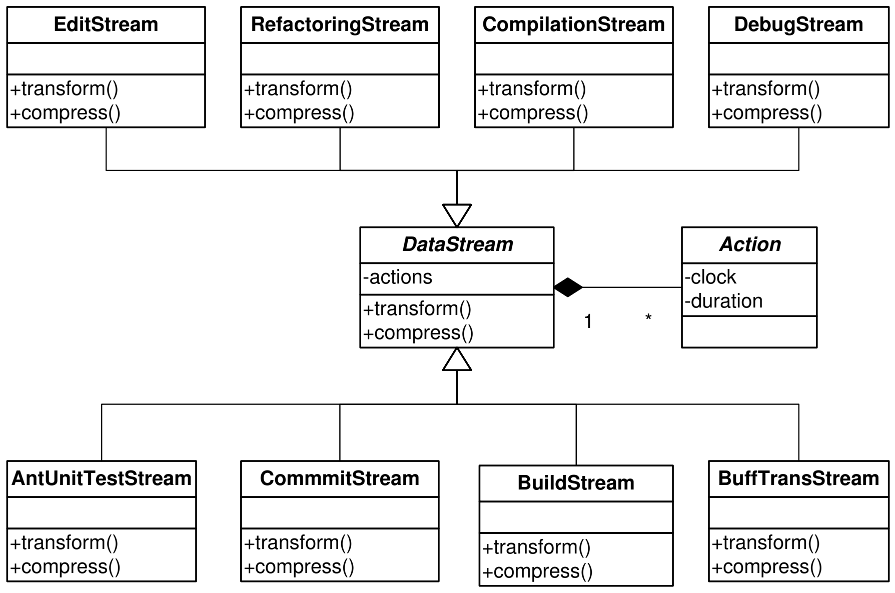


Figure 3.4. Class Diagram of Action Streams

represents a single development action. Using unit test invocation in the Eclipse IDE as an example, the Eclipse sensor collects and sends 3 "UnitTest" sensor data entries if the invoked unit test has 3 test cases. Because they are the consequence of a unit test invocation, the UnitTestStream can implement a compress function to group them into a single "UnitTest" development action.

Development Stream

Many kinds of action streams from a developer over a time period can merge together to form a software development stream as illustrated in Figure 3.2. The "DevelopmentStream" is an object representing a software development stream. It is the container of action streams. The following code demonstrates the usage of "DevelopmentStream" class. Table 3.1 is an example that shows a software development stream's internal representations.

```
DevelopmentStream stream =
    new DevelopmentStream(project, user, startDay, endDay);
stream.addSubstream(new EditStream(user));
stream.addSubstream(new BuffTransStream(user));
stream.addSubstream(new RefactoringStream(user));
stream.addSubstream(new UnitTestStream(user));
stream.addSubstream(new CompilationStream(user));
stream.assemble();
```

Table 3.1. An Excerpt of a Software Development Stream

Time	File	Event	Metrics
12:35:29	TestBowlingGame.java	ADD METHOD	void TestGameCreation()
12:37:47	TestBowlingGame.java	TEST EDIT	120sec MI=+1(3),SI=+1(4),TI=+1(3),AI=0(1)
12:38:07	BowlingGame.java	ADD CLASS	BowlingGame.java
12:38:08	BowlingGame.java	BUFFTRANS	FROM TestBowlingGame.java
12:38:13	TestBowlingGame.java	BUFFTRANS	FROM BowlingGame.java
12:39:17	TestBowlingGame.java	TEST EDIT	0sec MI=0(3),SI=-1(3),TI=0(3),AI=0(1)
12:39:17	TestBowlingGame.java	COMPILE	The constructor Frame() is undefined
12:39:28	BowlingGame.java	ADD METHOD	BowlingGame(Frame)
12:39:30	BowlingGame.java	BUFFTRANS	FROM TestBowlingGame.java
12:39:35	TestBowlingGame.java	BUFFTRANS	FROM BowlingGame.java
12:39:48	BowlingGame.java	PRODUCT EDIT	0sec MI=+1(1),SI=0(0),FI=+124(124)
12:39:50	BowlingGame.java	BUFFTRANS	FROM TestBowlingGame.java

3.2.2 Software Development Stream Partition

The second component of SDSA is the software development stream partition subsystem. The software development stream is a time-series data structure that may have hundreds to thousands development activities. Though applying machine learning algorithm is plausible [11], I proposed a mechanism to partition the development streams into episodes using the boundary conditions (Figure 3.5).

Larman and Basili[46] claim that the software development is iterative and incremental. Thus, tokenizing development streams not only simplifies mining large volumes of data, but also provides an approach to comparing actual software development processes to software process theories. Figure 3.5 illustrates the design of the software development stream partition algorithm. Tokenizers, which partition software development streams using boundary conditions, can be chained together. Currently, four tokenizers have been developed in SDSA. Figure 3.6 illustrates their class diagram. Of course, developers can implement and add new tokenizers if necessary.

- **Commit Tokenizer** partitions development streams by source code repository commit actions.
- **Command Tokenizer** partitions development streams by commands invoked in a shell window.

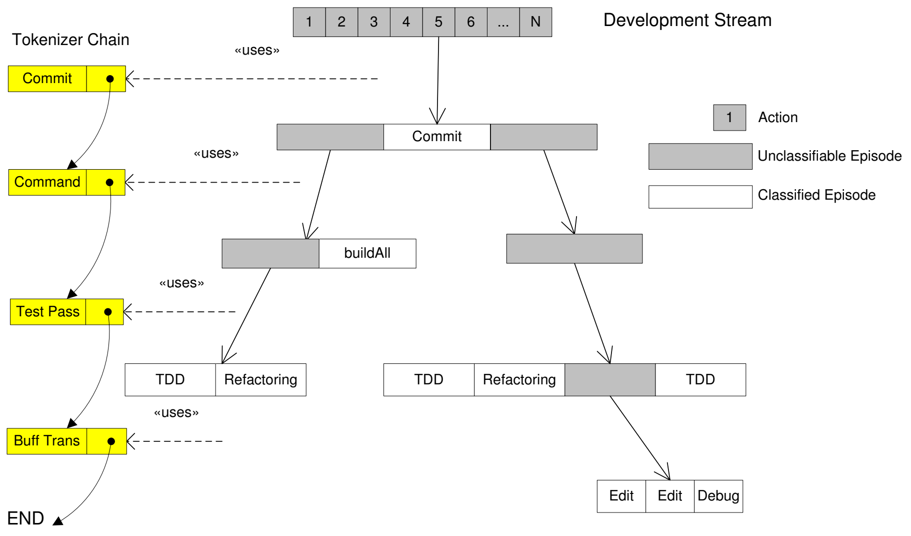


Figure 3.5. Partition of Development Stream

- **Test Pass Tokenizer** partitions development streams by successful unit test invocations.
- **Buffer Transition Tokenizer** partitions development streams by buffer changes in an IDE.

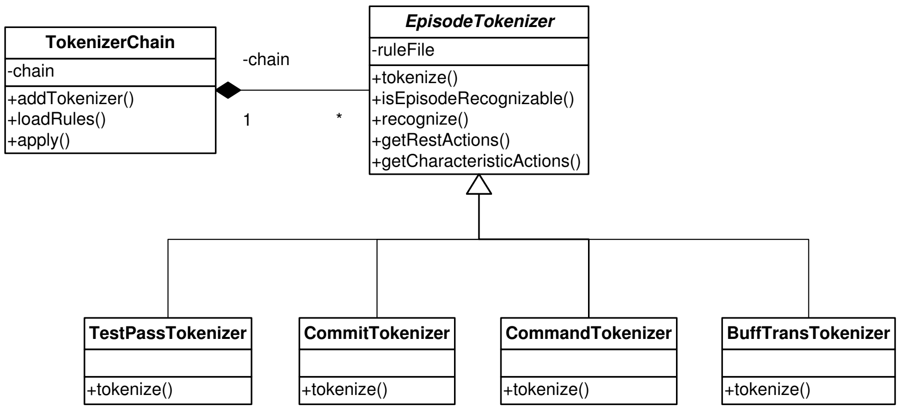


Figure 3.6. Class Structure of SDSA Tokenizers

Although it is possible, it is not necessary to have multiple tokenizers in a single run. In my dissertation research, I found that the “test pass” tokenizer is sufficient for the inference of Test-Driven Development behaviors.

3.2.3 Development Behaviors Inference

SDSA infers development behaviors using JESS [22], a rule-based system. Figure 3.7 illustrates how SDSA interacts with JESS to infer development behaviors from development actions in an episode.

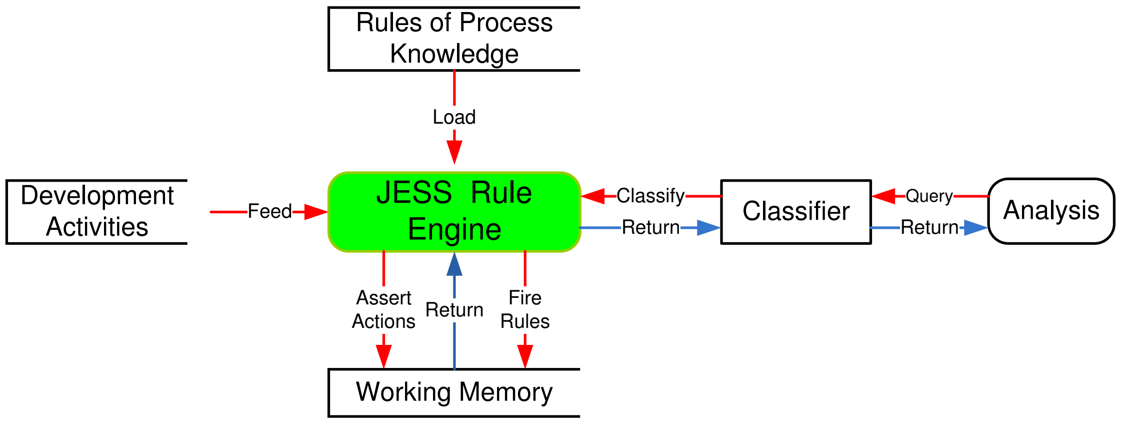
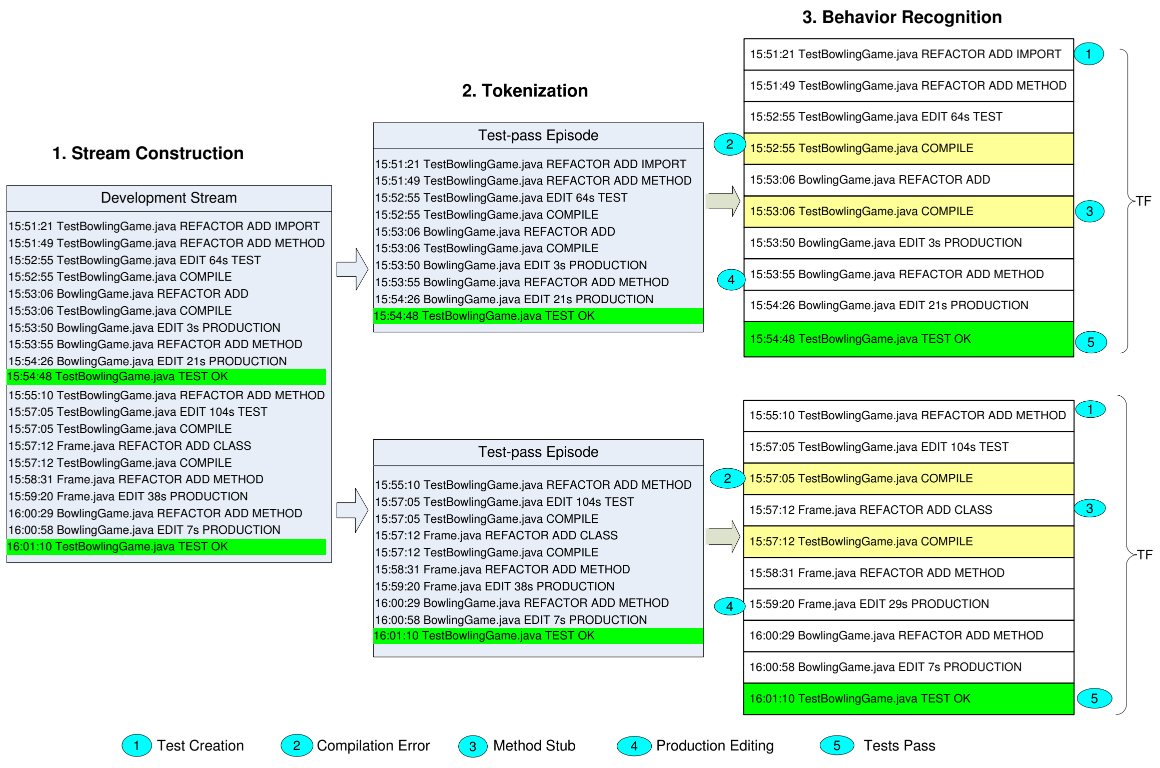


Figure 3.7. Developer Behavior Inference

The inference process is a mix of top-down and bottom-up methods. For the process of interest, developers can convert the process knowledge into a set of rules, which are then fed into JESS's working memory. JESS can infer development behavior after all development actions in an episode are asserted in JESS as facts. Inferred results can be queried by applications via a classifier, the interface between JESS and SDSA.

3.2.4 An Example

Now that we have shown how SDSA uses software process and product metrics for development behavior inference, I will use an example to demonstrate it. Figure 3.8 illustrates the usage of SDSA for inferring development behaviors. From 15:51:21 to 16:01:10, a developer implemented two user stories of the bowling game in Test-Driven Development. We used Hackystat to instrument the development process and collected software process metrics including refactoring, editing, compilation and test invocations. Following the development stream construction method described in Section 3.2.1, they were read and converted into development actions for construction of the development stream. Part 1 of Figure 3.8 illustrates the internal structure of the development stream that



```
Development Stream
15:51:21 TestBowlingGame.java REFACTOR ADD IMPORT
15:51:49 TestBowlingGame.java REFACTOR ADD METHOD
15:52:55 TestBowlingGame.java EDIT 64s TEST
15:52:55 TestBowlingGame.java COMPILE
15:53:06 BowlingGame.java REFACTOR ADD
15:53:06 TestBowlingGame.java COMPILE
15:53:50 BowlingGame.java EDIT 3s PRODUCTION
15:53:55 BowlingGame.java REFACTOR ADD METHOD
15:54:26 BowlingGame.java EDIT 21s PRODUCTION
15:54:48 TestBowlingGame.java TEST OK
15:55:10 TestBowlingGame.java REFACTOR ADD METHOD
15:57:05 TestBowlingGame.java EDIT 104s TEST
15:57:05 TestBowlingGame.java COMPILE
15:57:12 Frame.java REFACTOR ADD CLASS
15:57:12 TestBowlingGame.java COMPILE
15:58:31 Frame.java REFACTOR ADD METHOD
15:59:20 Frame.java EDIT 38s PRODUCTION
16:00:29 BowlingGame.java REFACTOR ADD METHOD
16:00:58 BowlingGame.java EDIT 7s PRODUCTION
16:01:10 TestBowlingGame.java TEST OK
```

```
Test-pass Episode
15:51:21 TestBowlingGame.java REFACTOR ADD IMPORT
15:51:49 TestBowlingGame.java REFACTOR ADD METHOD
15:52:55 TestBowlingGame.java EDIT 64s TEST
15:52:55 TestBowlingGame.java COMPILE
15:53:06 BowlingGame.java REFACTOR ADD
15:53:06 TestBowlingGame.java COMPILE
15:53:50 BowlingGame.java EDIT 3s PRODUCTION
15:53:55 BowlingGame.java REFACTOR ADD METHOD
15:54:26 BowlingGame.java EDIT 21s PRODUCTION
15:54:48 TestBowlingGame.java TEST OK
```

```
Test-pass Episode
15:55:10 TestBowlingGame.java REFACTOR ADD METHOD
15:57:05 TestBowlingGame.java EDIT 104s TEST
15:57:05 TestBowlingGame.java COMPILE
15:57:12 Frame.java REFACTOR ADD CLASS
15:57:12 TestBowlingGame.java COMPILE
15:58:31 Frame.java REFACTOR ADD METHOD
15:59:20 Frame.java EDIT 38s PRODUCTION
16:00:29 BowlingGame.java REFACTOR ADD METHOD
16:00:58 BowlingGame.java EDIT 7s PRODUCTION
16:01:10 TestBowlingGame.java TEST OK
```

```
15:51:21 TestBowlingGame.java REFACTOR ADD IMPORT     1
15:51:49 TestBowlingGame.java REFACTOR ADD METHOD
15:52:55 TestBowlingGame.java EDIT 64s TEST
15:52:55 TestBowlingGame.java COMPILE                 2
15:53:06 BowlingGame.java REFACTOR ADD
15:53:06 TestBowlingGame.java COMPILE                 3
15:53:50 BowlingGame.java EDIT 3s PRODUCTION
15:53:55 BowlingGame.java REFACTOR ADD METHOD         4
15:54:26 BowlingGame.java EDIT 21s PRODUCTION
15:54:48 TestBowlingGame.java TEST OK                 5
```

```
15:55:10 TestBowlingGame.java REFACTOR ADD METHOD     1
15:57:05 TestBowlingGame.java EDIT 104s TEST
15:57:05 TestBowlingGame.java COMPILE                 2
15:57:12 Frame.java REFACTOR ADD CLASS                3
15:57:12 TestBowlingGame.java COMPILE
15:58:31 Frame.java REFACTOR ADD METHOD
15:59:20 Frame.java EDIT 29s PRODUCTION               4
16:00:29 BowlingGame.java REFACTOR ADD METHOD
16:00:58 BowlingGame.java EDIT 7s PRODUCTION
16:01:10 TestBowlingGame.java TEST OK                 5
```

Figure 3.8. An SDSA Example

has 21 development actions. Among them, two are successful test invocations that are painted in green background. Then, in part 2 of Figure 3.8, SDSA's "Test Pass Tokenizer" (See Section 3.2.2) was used to partition the development stream into two episodes. Further, development actions in the first episode were fed into JESS to evaluate using an interface provided by SDSA. In part 3 of Figure 3.8, the inference rules detected a sequence of development activities that were in the order of (1) test creation, (2) compilation failures on test code, (3) method stub of production code, (4) production code editing, and (5) successful test invocation. Thus the development behavior in the first episode was recognized as "TF", a short name of Test-First. Similarly, the second episode was also recognized as "TF" by the inference rules.

3.3 Chapter Summary

In this chapter, I first introduced Hackystat, a generic framework for software metrics collection and analyses, which makes it possible to design the SDSA framework for studying low-level software processes. SDSA has three sub-processes: (1) software development stream construction, (2) software development stream partition, and (3) development behavior inference. SDSA is configurable and extensible. I concluded this chapter with an example in Section 3.2.4 using a portion of a development stream developed in TDD.

Chapter 4

Zorro Implementation

With the capabilities provided by Hackystat and SDSA, as part of my dissertation research, I implemented the Zorro software system for the automation of Test-Driven Development (TDD) behavior inference. As illustrated in Figure 1.1, Zorro is a specialization of the SDSA framework for TDD, supported by Hackystat. Zorro not only uses Hackystat's generic services such as sensor data collection and persistence, but also contributes new functionalities to Hackystat. Zorro defines several analyses and telemetry streams to study TDD using activities collected in development environment tools. This chapter starts with Zorro's extensions to the Hackystat and SDSA frameworks, followed by a collection of Zorro's TDD analyses.

4.1 Extensions to Hackystat's Data Collection

Zorro has special requirements for Hackystat sensors that are responsible for collecting software metrics. In order to partition development streams and recognize TDD development behaviors, certain software development activities must be collected. Table 4.1 contains a list of the required development activities including edit, compile, test, switch file, and refactor. The "DevEvent" is a sensor data type defined in Hackystat to represent these kinds of development activities using the combination of "Type" and "Subtype" attributes. In addition, DevEvent defines a special attribute named "PropertyMap" to hold the required metrics, which are listed in the "Product or Process Metrics" column of Table 4.1. Note that DevEvent does not have the "Kind" attribute as you can see in Table 4.1. It is for the sake of clarification for sensor developers to determine what metrics are mandatory. For instance, if a test is edited, the "current-test-method" and "current-test-assertion" metrics must be collected in addition to other metrics required for editing activities on production code.

Table 4.1. Sensor Data Types Required by Zorro

Development Activity	DevEvent		Product or Process Metrics	
	Type	Subtype	Kind	Names
Edit	Edit	StateChange and Save	Document	current-size
			Production	current-size, class-name, current-methods, current-statements
			Test	current-size, class-name, current-methods, current-statements, current-test-methods, current-test-assertions
Compile	Build	Compile		success, error
Test	Test	UnitTest		test-name, success, test-count, test-indice, elapsed-time, run-time
Switch File	Edit	BufferTransition		from-buff-name, to-buff-name
Refactor	Edit	ProgramUnit	Unary	op (add or remove), unit-name, unit-type, language
			Binary	op (rename or move), unit-type, from-unit-name, to-unit-name, language

These development activities are language and IDE independent. I wrote a sensor for Eclipse to gather these activities and more recently wrote a similar sensor for Visual Studio .NET. It should be possible to enhance a wide range of IDEs to collect necessary development activities for Zorro's automated TDD behavior inference.

4.2 Extensions to SDSA

Zorro is a specialization of SDSA for automated inference of development behaviors of TDD. Figure 4.1 illustrates how Zorro specializes SDSA. First, Zorro requires certain types of development activities, as described in Section 4.1. Thus, the development stream in Zorro is a special type of software development stream, which collects specific development activities useful for TDD recognition. Second, Zorro uses the "test pass" tokenizer defined in the SDSA framework to partition the development stream. Last, a special set of rules are defined in Zorro to infer TDD development behaviors. In this section, I will describe Zorro's specialization of SDSA in detail.

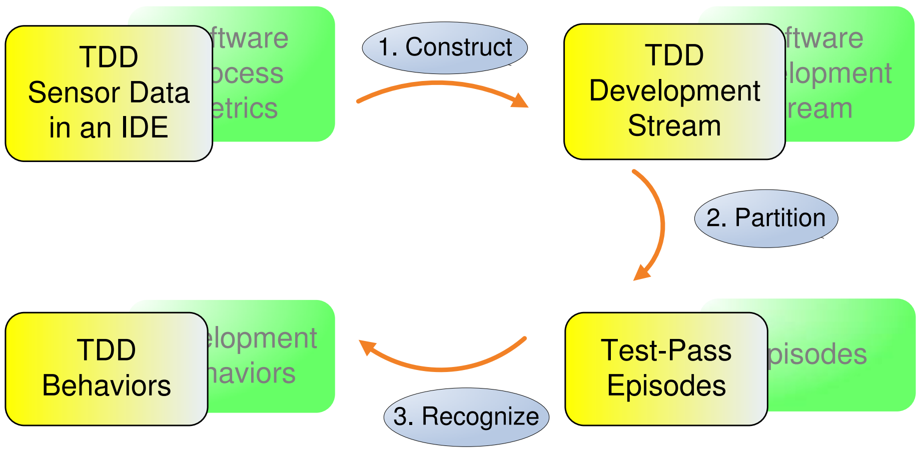


Figure 4.1. Zorro's Extensions to the SDSA Framework

4.2.1 Zorro Development Stream

Edit, compile, test, switch file, and refactor are five types of development activities required by Zorro. Though other development activities such as command line invocations can be conducted by developers, Zorro does not include them in the development stream because they are not useful for recognition of TDD. The following code shows how Zorro constructs a development stream using SDSA.

```
DevelopmentStream stream =
    new DevelopmentStream(project, user, startDay, endDay);
stream.addSubstream(new EditStream(user));
stream.addSubstream(new BuffTransStream(user));
stream.addSubstream(new RefactoringStream(user));
stream.addSubstream(new UnitTestStream(user));
stream.addSubstream(new CompilationStream(user));
stream.assemble();
```

This piece of code instantiates a "DevelopmentStream" for a given project, a project member, and a specified time period indicated by the start day and the end day. It assembles five action streams together to create the TDD development stream.

4.2.2 TDD Development Stream Partition

Zorro uses the “test pass” tokenizer defined in the SDSA framework (see Section 3.2.1). As a result, a development stream is partitioned into a sequence of “test pass” episodes all of which end with successful unit test invocations. Figure 3.8 illustrates this process.

Before we move on, let’s pause a while and think about the methods we are using. The development stream in Zorro contains five types of development activities that occur in an IDE when a developer is programming. These activities are not TDD specific. Moreover, the “test pass” tokenizer is used to divide the TDD development stream into “test pass” episodes. Again, the “test pass” tokenizer is not TDD specific and a developer might invoke tests no matter what development methods he/she is using. Given this, how can we recognize development behaviors in TDD? Can we use a dedicated tokenizer for TDD such as a “TDD Iteration” tokenizer?

Recall that the order of programming in TDD is “Red/Green/Refactor”. If a developer programs in TDD, tests should be invoked one or more times. Given a task from the To-Do list (see Section 2.1), a developer quickly writes a test and then invokes it. He or she will get a red bar if the test fails. After implementing functional code for the task, he or she will invoke the test again to see it pass. Therefore, when the “test pass” tokenizer is used, we will get an episode containing the “Red/Green” development portion of a TDD cycle. If there is any redundancy in the code, the developer will refactor it and then invoke the test again. As a result, we will get another episode containing the “Refactoring” portion of a TDD cycle. If we can successfully recognize the “test first” behavior in the first episode and the “refactoring” behavior in the second episode, then we will be able to recognize TDD development behavior using the divide-conquer method. This shows that the “test pass” tokenizer is sufficient for recognition of development behaviors in TDD. The first question is answered.

Can we have a “TDD Iteration” tokenizer? The answer is probably no. First, detecting refactoring development behaviors is hard. It is hard to tell whether a developer is adding new code or refactoring for simpleness and clarification. Second, as we have discussed in Chapter 1, developers may or may not conform to TDD, the Red/Green/Refactor cycle may not exist sometimes. Fortunately, the test pass tokenizer is sufficient for us to find the boundary of TDD’s Red/Green/Refactor cycles as we just showed in last paragraph.

4.2.3 Inference of TDD Development Behavior

Zorro also extends SDSA with a set of rules that enable instances of test pass episodes to be classified as one of 22 episode types. Table 4.2 lists these episode types, their definition in terms of their internal patterns of development activities, 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 end with a "Test Pass" event, since that is the episode boundary condition. UN-2 is provided as a way to classify a development stream where there is no unit testing at all.

Test First

"Test First" can be seen as the Red/Green portion of a TDD cycle. In a "Test First" episode, a test method or assertion statement is created first, followed by production code editing. Depending on compilation and test invocation results, an episode can be one of the four types: TF-1, TF-2, TF-3, or TF-4, as illustrated in Table 4.2.

Refactoring

Refactoring behaviors in reality are a bit complicated to recognize automatically because developers can refactor code in many different ways. A simple scenario of refactoring is to change a method implementation without changing its input and output. To accomplish this, the developer can edit code directly or abstract a new method from it for reuse. Directly matching the patterns of refactoring behaviors is hard, but we can observe them indirectly. First of all, there should be no test created. Second, code being refactored should not increase in size. If it does, the increase should be relatively small. With these two principles in mind, we can define rules to infer refactoring behaviors. Table 4.2 lists five kinds of refactoring episodes: RF-1, RF-2, RF-3, RF-4, and RF-5. Among them, RF-1 and RF-2 are refactoring on test, RF-3 and RF-4 are refactoring on production code, and RF-5 defines mixed refactoring on both test and production code.

Table 4.2. Zorro episode types, definitions, and TDD conformance

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 $\leq$ 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 editing →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

Test Addition

If you are a TDD developer, you probably have never heard of “Test Addition” since Beck did not explicitly define it in [6]. However, you may occasionally add a test that does not drive implementation of new production code, thus “Test Addition” exists even if you do not realize it. It is fundamentally different from the refactoring behavior on test code because test methods and assertion statements are added. Zorro defines two types of “Test Addition” behaviors: TA-1 and TA-2.

Regression

It is always a good habit to run existing tests before adding any new code into the system, especially at the beginning. This development behavior is called the regression test. The tests may fail due to problems such as incorrect environment settings, so the regression behavior has two categories: RG-1 and RG-2.

Code Production

Recall that developers might not develop software following the pattern of Red/Green/Refactor all the time, as we discussed in Chapter 1. Also, some developers may use other software development methods, such as Test Last Programming. For example, it is possible that a big chunk of production code can be inserted into the system without corresponding tests. Although tests are invoked in the end, the development behavior is actually about writing production code. Zorro therefore defines “Code Production” to classify this variety of development behaviors. Depending on the size increase of production code, “Code Production” has three types: CP-1 (statement increase only), CP-2 (small code increase), and CP-3 (big production code increase).

Test Last

“Test Last” is not a development behavior of TDD, but it is useful to define it. Researchers have compared TDD to Test Last Programming or Iterative Test Last [19, 53]. If Zorro differentiates TDD from “Test Last”, it can greatly help empirical researchers to study TDD in practice. “Test

First” is opposed to “Test Last” where a test is created after production code implementation. “Test Last” has two types: TL-1 and TL-2, however, the difference between the two types is very small.

Long

Development behaviors in long episodes such as ones with too many development activities or ones that last a very long time (over 30 minutes) are very hard to recognize. Therefore, Zorro does not infer development behaviors of long episodes. This is acceptable because iterations of TDD should be just seconds or minutes in duration[5]. If an episode is very long, it is very likely not to be TDD conformant.

Unknown

The last type of development behavior is “Unknown”. UN-2 is defined to classify development streams that do not have any successful unit test invocation. UN-1 is defined to incorporate the situation that the rule set is insufficient, which is very rare but we can not exclude it.

4.2.4 TDD Conformance

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. Table 4.2 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, and 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 Production” 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 TDD 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 TDD non-conformant episodes, then these ambiguous episodes types are classified as TDD non-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). Figure 4.2 illustrates this scenario. In Figure 4.2, a rectangle with letters represents an episode, and it is painted with green background if it is TDD conformant according to the heuristic.

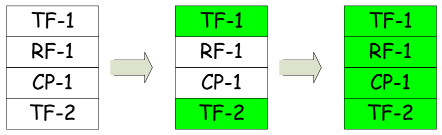


Figure 4.2. Episode Sequence Example A

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 TDD non-conformant episode types (TL-1 and TL-2). This scenario is illustrated in Figure 4.3 where an episode is painted with red background if it is TDD non-conformant.

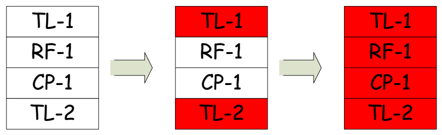


Figure 4.3. Episode Sequence Example B

Now consider a sequence like: [TL-1, RF-1, CP-1, TF-1] illustrated in Figure 4.4. Here, the two ambiguous episodes (RF-1 and CP-1) are surrounded on one side by an unambiguously TDD conformant episode (TL-1) and on the other side by an unambiguously TDD non-conformant episode (TF-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 TDD non-conformant. Figure 4.4 illustrates both of heuristics. Again, an episode is painted with green background if it is TDD conformant, otherwise it is painted with red background. Both of heuristics are supported by Zorro. In default, the optimistic one is used, but Zorro allows developers to switch to the pessimistic heuristic.

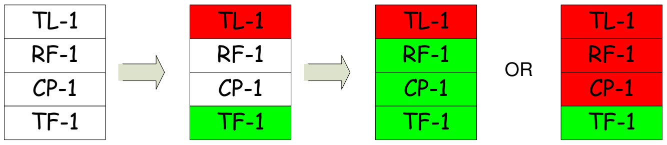
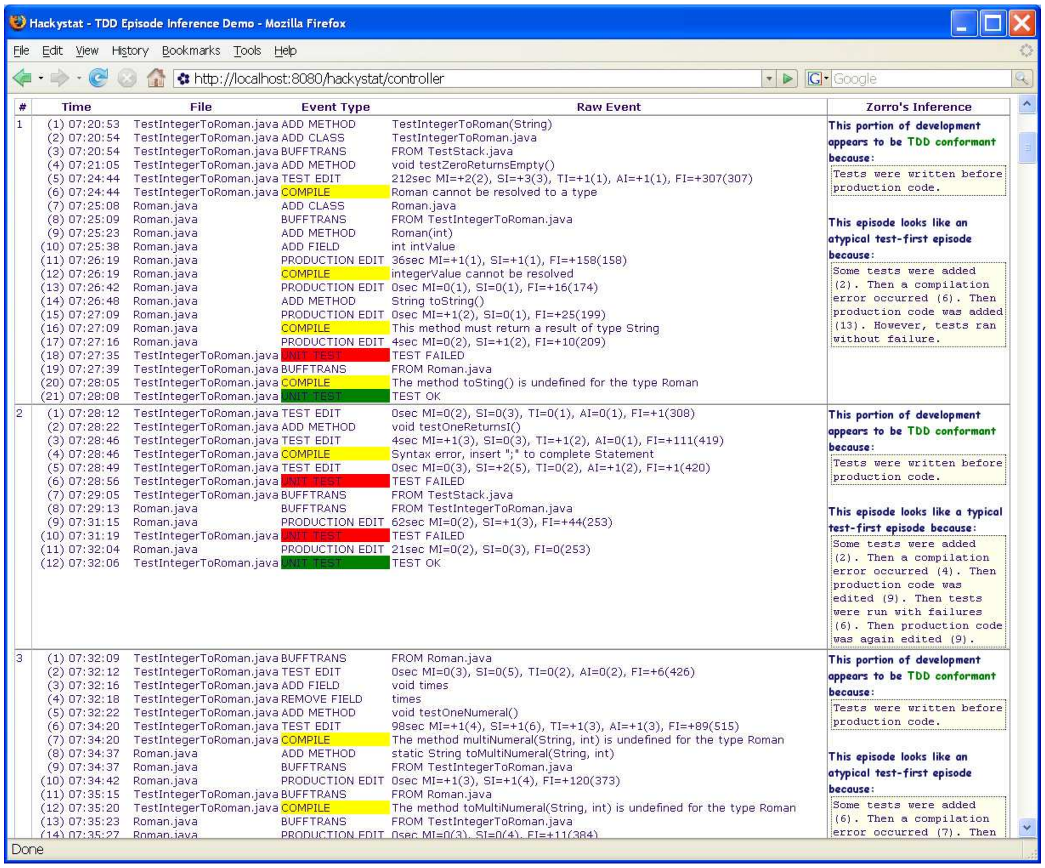


Figure 4.4. Optimistic and Pessimistic Heuristic Algorithms

4.2.5 Zorro's TDD Episode Inference

With episode categorization and TDD conformance heuristics, both of which are supported by the rule-based system, Zorro can automate the recognition of TDD. I conducted the "TDD Episode Inference" analysis using a real TDD development stream to demonstrate the Zorro's inference result. An experienced developer solved the Roman numeral conversion problem (see Appendix C)


Hackystat - TDD Episode Inference Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/hackystat/controller
Google
Done

#	Time	File	Event Type	Raw Event	Zorro's Inference
1	(1) 07:20:53	TestIntegerToRoman.java	ADD METHOD	TestIntegerToRoman(String)	This portion of development appears to be TDD conformant because: Tests were written before production code. This episode looks like an atypical test-first episode because: Some tests were added (2). Then a compilation error occurred (6). Then production code was added (13). However, tests ran without failure.
	(2) 07:20:54	TestIntegerToRoman.java	ADD CLASS	TestIntegerToRoman.java	
	(3) 07:20:54	TestIntegerToRoman.java	BUFFTRANS	FROM TestStack.java	
	(4) 07:21:05	TestIntegerToRoman.java	ADD METHOD	void testZeroReturnsEmpty()	
	(5) 07:24:44	TestIntegerToRoman.java	TEST EDIT	212sec MI=+2(2), SI=+3(3), TI=+1(1), AI=+1(1), FI=+307(307)	
	(6) 07:24:44	TestIntegerToRoman.java	COMPILE	Roman cannot be resolved to a type	
	(7) 07:25:08	Roman.java	ADD CLASS	Roman.java	
	(8) 07:25:09	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(9) 07:25:23	Roman.java	ADD METHOD	Roman(int)	
	(10) 07:25:38	Roman.java	ADD FIELD	int intValue	
	(11) 07:26:19	Roman.java	PRODUCTION EDIT	36sec MI=+1(1), SI=+1(1), FI=+158(158)	
	(12) 07:26:19	Roman.java	COMPILE	integerValue cannot be resolved	
	(13) 07:26:42	Roman.java	PRODUCTION EDIT	0sec MI=0(1), SI=0(1), FI=+16(174)	
	(14) 07:26:48	Roman.java	ADD METHOD	String toString()	
	(15) 07:27:09	Roman.java	PRODUCTION EDIT	0sec MI=+1(2), SI=0(1), FI=+25(199)	
	(16) 07:27:09	Roman.java	COMPILE	This method must return a result of type String	
	(17) 07:27:16	Roman.java	PRODUCTION EDIT	4sec MI=0(2), SI=+1(2), FI=+10(209)	
	(18) 07:27:35	TestIntegerToRoman.java	UNIT TEST	TEST FAILED	
	(19) 07:27:39	TestIntegerToRoman.java	BUFFTRANS	FROM Roman.java	
	(20) 07:28:05	TestIntegerToRoman.java	COMPILE	The method toSting() is undefined for the type Roman	
	(21) 07:28:08	TestIntegerToRoman.java	UNIT TEST	TEST OK	
2	(1) 07:28:12	TestIntegerToRoman.java	TEST EDIT	0sec MI=0(2), SI=0(3), TI=0(1), AI=0(1), FI=+1(308)	This portion of development appears to be TDD conformant because: Tests were written before production code. This episode looks like a typical test-first episode because: Some tests were added (2). Then a compilation error occurred (4). Then production code was edited (9). Then tests were run with failures (6). Then production code was again edited (9).
	(2) 07:28:22	TestIntegerToRoman.java	ADD METHOD	void testOneReturnsI()	
	(3) 07:28:46	TestIntegerToRoman.java	TEST EDIT	4sec MI=+1(3), SI=0(3), TI=+1(2), AI=0(1), FI=+111(419)	
	(4) 07:28:46	TestIntegerToRoman.java	COMPILE	Syntax error, insert ";" to complete Statement	
	(5) 07:28:49	TestIntegerToRoman.java	TEST EDIT	0sec MI=0(3), SI=+2(5), TI=0(2), AI=+1(2), FI=+1(420)	
	(6) 07:28:56	TestIntegerToRoman.java	UNIT TEST	TEST FAILED	
	(7) 07:29:05	TestIntegerToRoman.java	BUFFTRANS	FROM TestStack.java	
	(8) 07:29:13	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(9) 07:31:15	Roman.java	PRODUCTION EDIT	62sec MI=0(2), SI=+1(3), FI=+44(253)	
	(10) 07:31:19	TestIntegerToRoman.java	UNIT TEST	TEST FAILED	
	(11) 07:32:04	Roman.java	PRODUCTION EDIT	21sec MI=0(2), SI=0(3), FI=0(253)	
	(12) 07:32:06	TestIntegerToRoman.java	UNIT TEST	TEST OK	
3	(1) 07:32:09	TestIntegerToRoman.java	BUFFTRANS	FROM Roman.java	This portion of development appears to be TDD conformant because: Tests were written before production code. This episode looks like an atypical test-first episode because: Some tests were added (6). Then a compilation error occurred (7). Then
	(2) 07:32:12	TestIntegerToRoman.java	TEST EDIT	0sec MI=0(3), SI=0(5), TI=0(2), AI=0(2), FI=+6(426)	
	(3) 07:32:16	TestIntegerToRoman.java	ADD FIELD	void times	
	(4) 07:32:18	TestIntegerToRoman.java	REMOVE FIELD	times	
	(5) 07:32:22	TestIntegerToRoman.java	ADD METHOD	void testOneNumeral()	
	(6) 07:34:20	TestIntegerToRoman.java	TEST EDIT	98sec MI=+1(4), SI=+1(6), TI=+1(3), AI=+1(3), FI=+89(515)	
	(7) 07:34:20	TestIntegerToRoman.java	COMPILE	The method multiNumeral(String, int) is undefined for the type Roman	
	(8) 07:34:37	Roman.java	ADD METHOD	static String toMultiNumeral(String, int)	
	(9) 07:34:37	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(10) 07:34:42	Roman.java	PRODUCTION EDIT	0sec MI=+1(3), SI=+1(4), FI=+120(373)	
	(11) 07:35:15	TestIntegerToRoman.java	BUFFTRANS	FROM Roman.java	
	(12) 07:35:20	TestIntegerToRoman.java	COMPILE	The method toMultiNumeral(String, int) is undefined for the type Roman	
	(13) 07:35:23	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(14) 07:35:27	Roman.java	PRODUCTION EDIT	0sec MI=0(3), SI=0(4), FI=+11(384)	

Figure 4.5. Demo of TDD Conformance Inference

using TDD in the Eclipse IDE. The Hackystat Eclipse sensor was installed to instrument the development process for collecting development activities. Zorro partitioned his development stream into 16 episodes, and inferred his development behaviors. The screen-shot in Figure 4.5 shows the first three episodes with inferred development behaviors. They are all types of “Test First” and conformant to TDD according to Zorro’s inference.

This analysis is useful at demonstrating to new users how Zorro infers development behaviors. Another use of this analysis is to validate the correctness of Zorro’s inference results. I will show how I enhanced this analysis for Zorro’s validation in Chapter 7. Next, I will introduce Zorro’s extensions to Hackystat’s functionalities.

4.3 Extensions to Hackystat’s Functionalities

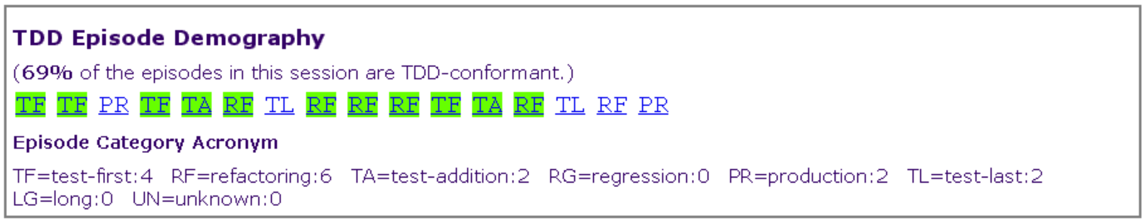
TDD conformance is very important according to the discussion of related work in Chapter 2, and therefore I implemented the Zorro software system to provide explicit support for process conformance. Besides TDD conformance, with the capability of automated inference of TDD behaviors, more useful analyses that once were very hard can be conducted now. In the course of my research, I implemented TDD analyses including “Episode Demography”, “T/P Effort Ratio”, and “Episode Duration Distribution” etc., for studying TDD (see Section 4.3.1). Moreover, with the aid of Software project telemetry infrastructure, I implemented TDD telemetry reducers to assist research and management of software development in TDD (see Section 4.3.2).

4.3.1 TDD Analyses

The following Zorro analyses have been developed to study software development in TDD and other development methods that include unit testing.

Episode Demography

The episode demography analysis provides an overview of a programming session as shown in Figure 4.6. In Figure 4.6, each small box with a two-letter acronym represents an episode. The legend explains the links between two-letter episode acronyms and episode types. The background


TDD Episode Demography

(**69%** of the episodes in this session are TDD-conformant.)

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

Episode Category Acronym

TF=test-first:4 RF=refactoring:6 TA=test-addition:2 RG=regression:0 PR=production:2 TL=test-last:2
LG=long:0 UN=unknown:0

Figure 4.6. Episode Demography Analysis

color of an episode tells its TDD conformance. If the episode is TDD conformant, the background is green; otherwise it is transparent. Episodes are clickable in the actual analysis. Clicking on an episode takes users to another analysis that is similar to the one illustrated in Figure 4.5.

Some useful information can be obtained using the episode demography analysis. For instance, the example presented in Figure 4.6 shows that this programming session has 17 episodes, of which 69% are TDD conformant. Four episodes are “Test First”, six are “Refactoring”, and two are “Test Last”. This information can be used to improve TDD conformance. If a higher degree of TDD conformance is wanted, developers can study what are the legitimate TDD episodes to improve their compliance with TDD. Moreover, this analysis can also be used to find development patterns. For instance, in Figure 4.6, we can observe two patterns (TF)(TA)+ and (TA)(RF)+. The pattern (TF)(TA)+ means that a “Test First” episode is followed by one or more “Test Addition” episodes. The (TA)(RF)+ means that a “Test Addition” episode is followed by one or more “Refactoring” episodes.

T/P Effort Ratio

Unlike “Episode Demography”, the “T/P Effort Ratio” analysis is not directly derived from Zorro’s inference. In Section 4.1, we discussed that Zorro extends Hackystat’s data collection. One extension is that it requires numbers of test methods and assertion statements. With this information, we can easily tell if a developer is working on test code at a given time. The T/P Effort Ratio stands for the effort a developer spent on test code, compared to the effort spent on production code. Figure 4.7 shows an example of the T/P Effort Ratio analysis using the same TDD programming session used in Figure 4.6. In Figure 4.7, the horizontal axis is the elapsed development time in minutes, and the vertical axis is the ratio of effort spent on test code to effort spent on production code. The

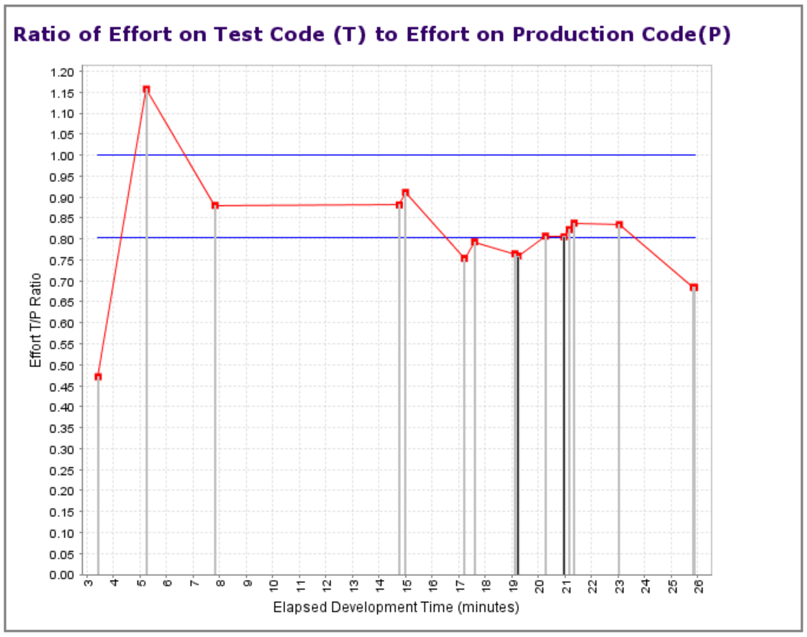


Figure 4.7. Test Effort vs. Production Effort

T/P ratio over 1.0 indicates more effort on writing test code than on writing production code. The vertical bars are episode borders, thus, the span between bars can represent episode durations.

The example illustrated in Figure 4.7 shows that the effort spent on testing code is consistent over the course of this TDD development session. Approximately, on average, effort on testing code is about 80% of effort on production code. This analysis can be applied to not only TDD but also other development methods that include unit testing. An interesting use of it would be to compare T/P effort ratio differences between TDD and Test Last Programming.

T/P Size Ratio

Same as the T/P Effort Ratio analysis, the T/P Size Ratio analysis is also based upon Zorro's extension to Hackystat's data collection. In Table 4.1, we have shown that the Zorro compatible sensor collects size information (current-size, mostly line of code) for both of test and production code. With this information, we can compute the incremental changes of test code size, production code size, and the ratio of the two. Figure 4.8 shows an example of the T/P Size Ratio analysis using the same TDD programing session. The interpretation to this analysis is exactly the same to Figure

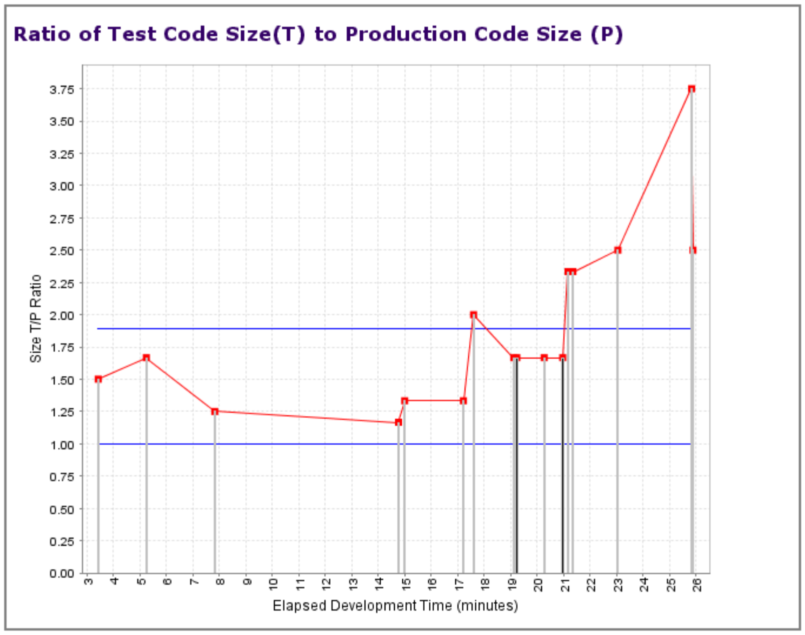


Figure 4.8. Test Size vs. Production Size

4.7, except that each value in Figure 4.8 is the ratio of test code size to production code size.

The example illustrated in Figure 4.8 shows that test code is always more than production code. Again, this analysis is applicable to any development methods where unit testing is practiced.

Episode Duration

The Episode Duration analysis is inspired by TDD's characteristically short durations that I discussed in Section 2.1. How frequently unit tests are invoked is an alternative indicator of TDD conformance [74]. Figure 4.9 shows an example of the Episode Duration analysis. The interpretation of this analysis is very straightforward. The horizontal axis has the two-letter acronym for each episode. The vertical axis is the episode duration in minutes. As we can see in Figure 4.9, with a little hill climbing at the beginning, tests were invoked frequently in this TDD programming session.

Similarly, this analysis is applicable to any development methods where unit testing is practiced. The durations in Figure 4.9 are mostly short, which supports the short duration claim made about TDD iterations. In practice, I rarely observed episodes that were longer than 30 minutes in duration.

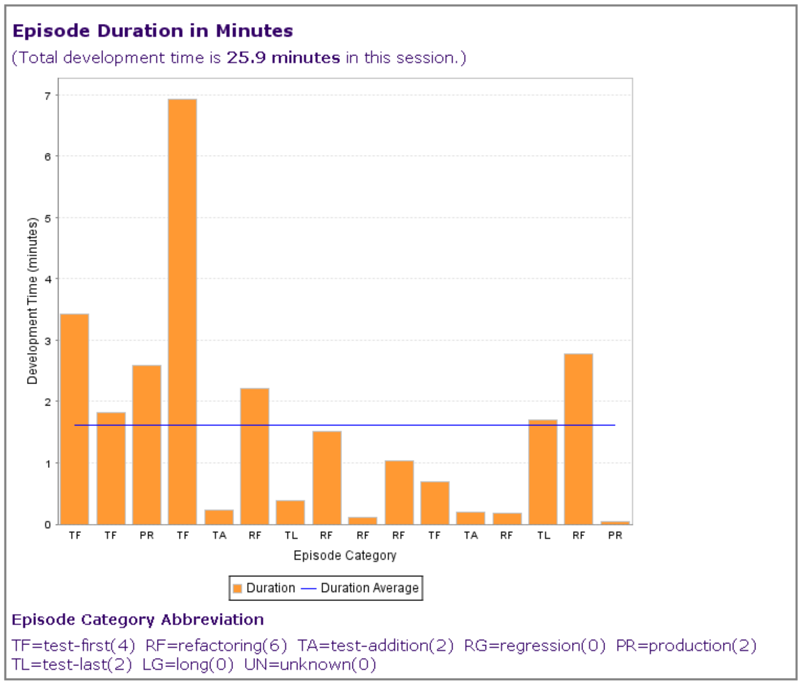


Figure 4.9. Episode Duration

In that case, the conformance to TDD is likely not stringent any more. A tabular report (Table 4.3) is also available for comparing episode duration differences among episode categories.

Table 4.3. Duration Average by Episode Category

Category	Average Duration (minutes)
TF	3.2
RF	1.3
TA	0.2
RG	0
PR	1.3
TL	1
LG	0
UN	0

Episode Duration Bin

The Episode Duration analysis lines up episodes with their durations, which is useful at displaying progressive changes. An alternative presentation is to arrange episodes with close durations into a bin. Figure 4.10 is an example showing the Episode Duration Bin analysis. In Figure 4.10, the

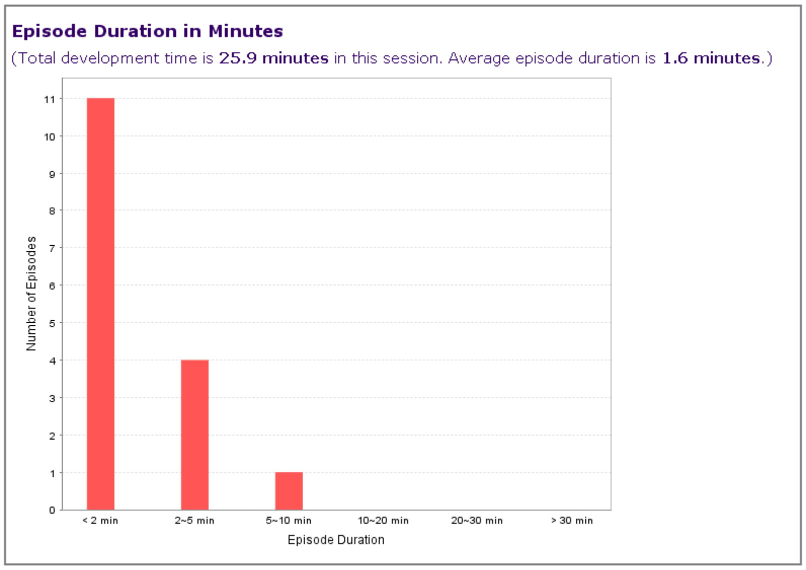


Figure 4.10. Episode Duration Bin

horizontal axis has a set of bins of episode durations, and the vertical axis is the number of episodes.

Similar to the Episode Duration analysis, this analysis can be used to verify TDD's short duration characteristic. An ideal TDD development session should have many short episodes with no or very few long episodes. If too many long episodes are observed, perhaps developers should take actions to develop and run tests more frequently. An episode duration distribution table (see Table 4.4) is attached to this analysis.

Table 4.4. Episode Duration Distribution by Category

Category	<2 min	2˜5 min	5˜10 min	10˜20 min	20 ˜30 min	>30 min	Total
TF	2	1	1				4
RF	4	2					6
TA	2						2
RG							0
PR	1	1					2
TL	2						2
LG							0
UN							0
Total	11	4	1	0	0	0	16

4.3.2 TDD Telemetry Streams

The analyses in Section 4.3.1 leverage Zorro's recognition of TDD development behaviors. They can be used to understand and improve TDD development for individuals. In order to support project management and improvement, I implemented a group of TDD telemetry streams in my research. Because Zorro abstracts software metrics into low-level development behaviors, the synergy of Zorro and Software telemetry allows better management of software projects developed in TDD. In this section, I will introduce Software telemetry, and present TDD telemetry analyses.

Software Project Telemetry and TDD Telemtry Reducers

The Software project telemetry [40, 79] was developed by Qin Zhang in the Collaborative Software Development Lab at the University of Hawaii. It is an infrastructure to aggregate metrics data together to perform daily, weekly, or monthly analyses to support in-process software project management and decision makings. Software telemetry abstracts software metrics into streams, charts and reports. The telemetry stream contains a series of time-stamped events in the daily, weekly, or monthly granularity. The telemetry charts and reports provide visualization of telemetry streams. Detecting changes and covariance in the trend of telemetry streams enables an incremental, visible, and experimental approach to manage software projects [79].

The telemetry reducer is the extension point of Software telemetry, which can be used to extend the existing telemetry stream base. The following lists TDD telemetry reducers supplied in Zorro.

- **TDDPercent Reducer**:
  Computes a single telemetry stream for percentage of TDD development time to overall development time. Alternatively, the percentage can also be the number of TDD compliant episodes to the number of total episodes.

- **TDDProductionDevTime Reducer**:
  Computes a single telemetry stream of development time on production code. Though its name suggests that this reducer is for TDD, it can actually be applied to other development methods as well.

- **TDDTestDevTime Reducer**:
  Computes a single telemetry stream of development time on test code. Same as TDDPro-

ductionDevTime, it can be applied to development methods other than TDD if unit testing is used.

- **TDDDevTime Reducer**:
  Computes a single telemetry stream of total development time. Note that Software telemetry already defines a DevTime, which is different from TDDDevTime. The TDDDevTime can be thought as the summation of development time on production, test, and others such as XML configuration files.

- **EpisodeAverageDuration Reducer**:
  Computes a single telemetry stream of average episode duration. It can be applied to TDD and other development methods. It indicates how frequently unit test is invoked.

- **TDDMemberProductionDevTime Reducer**:
  Computes multiple telemetry streams for development time on production code, one telemetry stream for each project member.

- **TDDMemberTestDevTime Reducer**:
  Computes multiple telemetry streams for development time on test code, one telemetry stream for each project member.

- **TDDMemberDevTime Reducer**:
  Computes multiple telemetry streams for total development time, one telemetry stream for each project member.

- **MemberEpisodeAverageDuration Reducer**:
  Computes multiple telemetry streams for average episode durations, one telemetry stream for each project member.

TDD Telemetry Analyses

The availability of TDD telemetry reducers enables developers to invoke telemetry analyses using Zorro's recognized low-level development behaviors.

Recall that a benefit of TDD is that the code developed in TDD should be 100% covered because no functional code is created without a unit test (see Section 2.1). So it would be interesting to study this claim. Since Zorro has the TDDPercent Reducer, we can define a telemetry stream reporting percentages of TDD development and correlate it with the test coverage stream that is already included in Software telemetry(Page 59 [79]). The following code is a definition of the telemetry stream and chart for this investigation.

```
streams TDDPercent(type) = {
  "Percentage of Test-Driven Development",
  TDDPercent(type) * 100
};

chart TDD-Coverage-Percentage-Chart() = {
  "Percentage of TDD Episodes (time) and Coverage",
  (TDDPercent("time"), percentageYAxis("TDD \%")),
  (Coverage-Percentage("**", "line"),
   percentageYAxis("Coverage \%"))
};
```

Figure 4.11 is a weekly telemetry chart showing percentage of TDD development and test coverage. From the week of Sep 30, 2006 to the week of Nov 18, 2006, I worked on the Zorro software

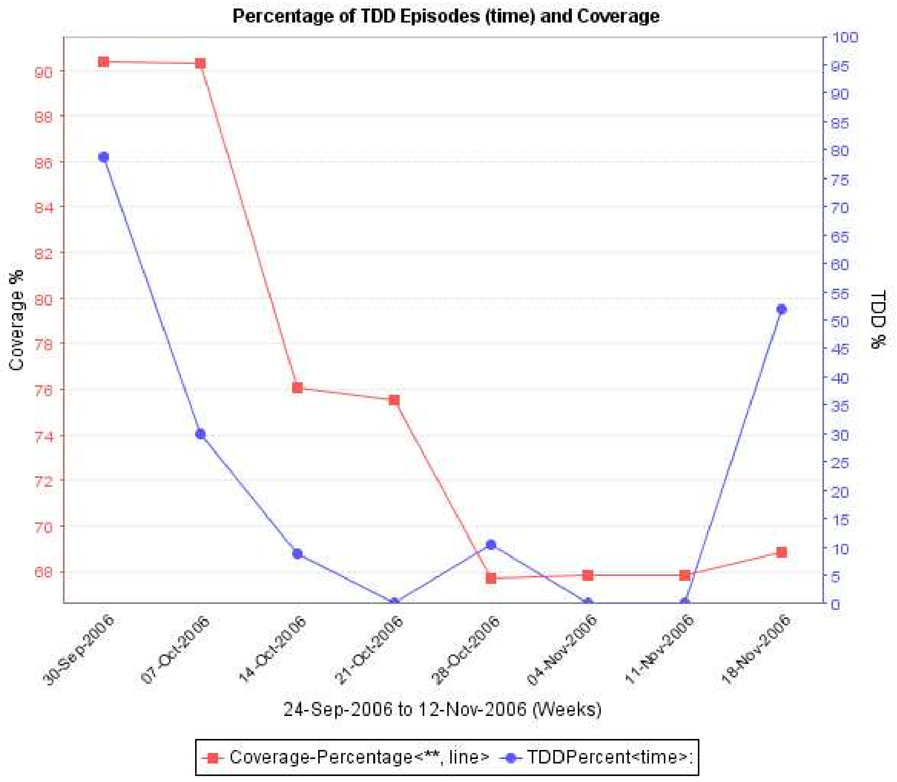


Figure 4.11. TDD Percentage and Testing Coverage

system and implemented Zorro's web validation interface. Due to the fact that testing web interfaces requires a lot of additional effort, my conformance to TDD dropped down significantly in that period. As a result, the test coverage dropped from 90% to below 70% over the course of eight weeks software development. Though we can not directly verify the claim of 100% test coverage characteristic of TDD, indeed the analysis shows that deviating from TDD caused decrease of test coverage. In turn, this indicates that TDD can help to improve test coverage.

4.4 Chapter Summary

In this chapter, I introduced Zorro's implementation with the support of the Hackystat and SDSA frameworks. Zorro extends Hackystat's data collection, specializes SDSA's low-level development behaviors inference for TDD, and supplies new functionalities to Hackystat. Zorro's development behaviors and conformance of TDD were detailed in this chapter, along with an introduction to TDD analyses and telemetry reducers implemented in Zorro.

Chapter 5

Research Questions and Methodology

The long-term goal of my research is to improve the consistency of empirical evaluation research on low-level software processes. As a step in this direction, I designed and developed the Software Development Stream Analysis (SDSA) framework (Chapter 3) that can infer development behaviors using automatically collected in-process software metrics. I used the SDSA framework to specify a low-level software process called Test-Driven Development (TDD) and implemented Zorro, a software system that can recognize development behaviors and measure the developer's process conformance to TDD. If Zorro, a specification of the SDSA framework, can be experimentally validated, I will have provided evidence that SDSA is an enabling technique for recognizing at least certain types of low-level development activities.

In order to validate Zorro, I designed a series of case studies including a pilot study,a classroom case study and an industrial case study. The purpose of the pilot and classroom validation studies was to validate Zorro's data collection and TDD behavioral inference in controlled environments. The purpose of the industrial case study was to explore how Zorro can be used by external researchers. Section 5.1 introduces the central research questions of my thesis research. Section 5.2 presents my research methods including participant observation, interview and survey.

5.1 Research Questions

The central research questions of the Zorro validation studies were:

- Q1: Can Zorro automate the recognition of TDD behaviors using automatically collected software metrics?

  This question can be further divided into three sub-questions:

  - Can Zorro collect software metrics correctly?
  - Does Zorro collect the necessary software metrics?
  - Can Zorro infer TDD behaviors correctly based upon automatically collected software metrics?

- Q2: How useful is Zorro?

  This question is hard to answer but it can be divided into three sub-questions based upon users' roles.

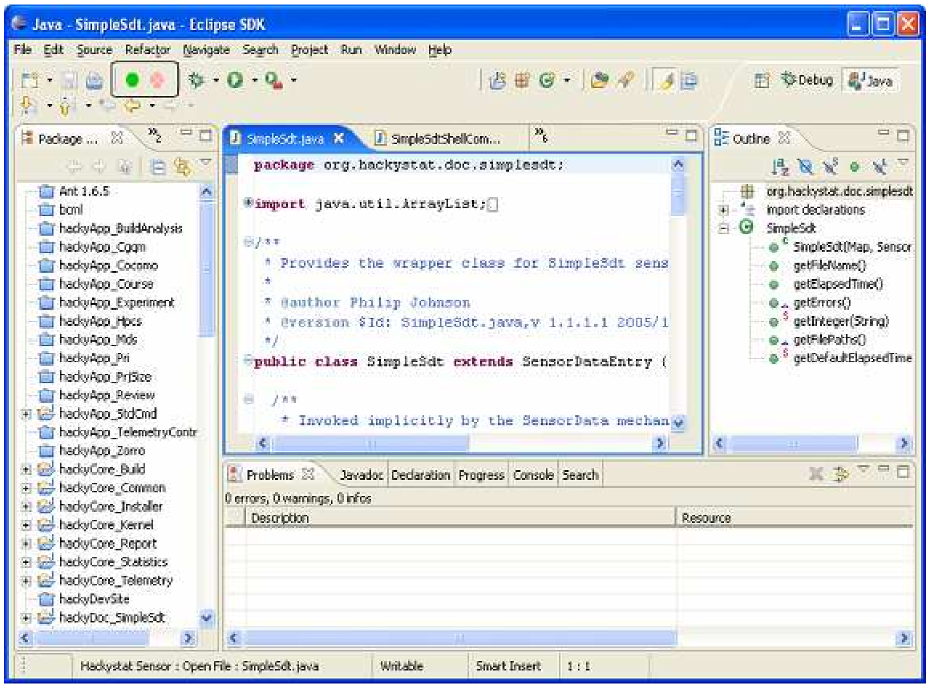
  - For beginners, can Zorro help them improve their compliance to TDD?
  - For experienced TDD practitioners, will Zorro help them improve their TDD practice by providing them with new insight into their TDD development behaviors?
  - For researchers, can Zorro help them reach legitimate research conclusions on TDD experiments by providing them with TDD process conformance information?

5.2 Research Methodology

Answering the above research questions requires a "mixed methods" research methodology according to [12]. Question Q1 can be investigated using the participant observation. In order to investigate the research question Q2, we need to collect users' feedback or interview them.

To use the participant observation method, we can observe participants in the field, take notes about their development activities and behaviors to provide an independent data source for validating Zorro's data collection and TDD behaviors inference. However, this manual participant observation method [12] does not work for studying low-level software development activities that could occur in seconds and minutes. It is simply too demanding for the observer to record all of the necessary information about a low level software process, where significant events can occur every few seconds. Thus, I developed a recording tool called "Eclipse Screen Recorder" (ESR) to assist participant observation. ESR is an Eclipse plugin that can generate a QuickTime movie containing time-stamped screen shots of the Eclipse window at regular intervals. With ESR, observers will not need to worry about missing rapid development activities.

Figure 5.1 illustrates the Eclipse window with ESR installed. The two pictured buttons on the toolbar menu are defined by ESR. Pressing the green button can start the recording process and make the red button become enabled. The recorder will stop if either the red button is pressed or Eclipse is closed. Figure 1.7 in Chapter 1 illustrates the recorded movie being played by QuickTime.



```
package org.hackystat.doc.simplesdt;

import java.util.ArrayList;

/**
 * Provides the wrapper class for SimpleSdt sens
 *
 * @author Philip Johnson
 * @version $Id: SimpleSdt.java,v 1.1.1.1 2005/1
 */
public class SimpleSdt extends SensorDataEntry {

  /**
   * Invoked implicitly by the SensorData mechan
```

```
0 errors, 0 warnings, 0 infos
```

Figure 5.1. Eclipse Screen Recorder

The embedded timeline at bottom of videos (See Figure 1.7) can be used to synchronize videos with development activities collected by Zorro for validation. ESR allows customization of recording rate (0.5-2 frames per second), picture quality, and resolution. One frame/second was found to be sufficient for validation, generating file sizes of approximately 7-8MB per hour of video.

The development of Zorro was iterative and incremental. So was the validation of Zorro. I have conducted three case studies — a pilot study, a classroom case study, and an industrial case study. Next, I will introduce the research methods I used in these case studies.

5.2.1 Pilot Study

I designed and implemented the Zorro software system based upon descriptions in books [6, 3, 51, 47, 30] and my observation of TDD in practice. By Spring 2005, I had enhanced the Hackystat Eclipse sensor to collect necessary development activities, designed the SDSA framework and implemented the Zorro software system to infer TDD development behaviors automatically. In the following Fall semester, I conducted a pilot Zorro validation case study.

The purpose of the pilot study was to test ESR as a tool to provide independent evidence for validating Zorro, as well as to validate Zorro's data collection and TDD behavior inference. The research method was participant observation using ESR as the data collection tool. I played recorded QuickTime videos to observe participants' TDD development activities and behaviors for Zorro validation.

5.2.2 Classroom Case Study

The pilot study showed that participant observation with ESR is a suitable research method for conducting Zorro validation study. Though Zorro was not perfect at collecting necessary software metrics and inferring TDD behaviors, the pilot demonstrated that it was promising. Based on research results from the pilot study, I improved Zorro's data collection, enhanced Zorro's TDD behaviors inference, added heuristics on process conformance of TDD, and provided many TDD analyses and telemetry reducers (See Chapter 4). With these improvements, I conducted an extended validation study on Zorro in the software engineering classes at the University of Hawaii in Fall 2006.

The purpose of the classroom case study was to validate Zorro in a controlled environment and investigate its usefulness for TDD beginners. The research method of the validation was also participant observation. Participants developed software using TDD in Eclipse with instrumentation including the Hackystat Eclipse sensor and ESR. ESR served as the data collection tool for participant observation and the recorded videos were analyzed to validate Zorro.

A caveat with the pilot study was that I, the author of the Zorro software system, analyzed the recorded videos for Zorro validation. This creates a "construct validity" threat with regard to this experiment design because ESR was the only source of evidence and my analysis could be biased.

Therefore, in order to increase the construct validity of classroom case study, multiple sources of evidence were used according to suggestions from Yin in [78] (Page 34). Participants' comments were the third source of information I used to cross-validate video analysis results. Prior to the study, I developed a Zorro validation wizard in web pages to let participants comment.

To explore whether Zorro can help TDD beginners, I interviewed participants on their TDD development experiences. After the interview, each participant used the Zorro validation wizard to analyze his/her TDD development behaviors. Meanwhile, an embedded survey was conducted to evaluate Zorro's usefulness. To improve quality of the survey, I asked participants to justify their answers verbally.

5.2.3 Industry Case Study

In Spring 2007, I conducted an industry case study to test Zorro's usefulness to TDD researchers. Using Zorro as a tool to assess TDD process conformance, Dr. Geir Hanssen and Dr. Tor Erlend Fægri from SINTEF ICT of Norway conducted a TDD vs. non-TDD comparison study in a Norwegian software company.

The purpose of this study was to explore how Zorro can be used by researchers to increase quality of TDD's evaluation research. I deployed Zorro in the software company, assisted the project manager and the researcher on analyzing process conformance data, and conducted a survey at the end of the study.

5.3 Chapter Summary

This chapter introduced my research statement and Zorro's validation case studies. The central research questions were: (1) Can Zorro automate the recognition of TDD behaviors using automatically collected software metrics? and (2) How useful Zorro is? To address these questions, I have conducted a pilot study, a classroom case study and an industry case study in which I used research methods such as participant observation, interviews and surveys. In order to improve the quality of data collection for participant observation, I developed ESR, a tool that can record the Eclipse window in high fidelity. Both the pilot study and the classroom case study used ESR as a data collection tool for Zorro validation.

Chapter 6

Pilot Study

I employed the case study research strategy [78] to empirically study Zorro's TDD development behaviors inference. In order to validate Zorro, I conducted a pilot study at the University of Hawaii in Fall 2005. This study shows that the participant observation research method using ESR is an acceptable research method for Zorro validation and it also shows that Zorro can accurately recognize development behaviors of TDD in a simple project development setting.

6.1 Purpose of the Study

The pilot study served as a dry-run test to the chosen case study research strategy. As mentioned in Chapter 5, the purpose of the pilot study was to test ESR, as well as to validate Zorro's data collection and behaviors inference of TDD.

6.2 Research Questions

The specific research questions for the pilot study were:

- Q1a: Is ESR a suitable tool for Zorro validation study?
- Q1b: Does Zorro collect enough low-level development activities to infer developer's TDD behaviors?
- Q1c: Does Zorro's inference of TDD agree with analyses based upon participant observation?

Note that these specific research questions corresponded to the main research questions Q1: *Can Zorro automate the recognition of TDD behaviors using automatically collected software metrics?*

6.3 Experiment Design

The pilot study was largely a one-shot case study in which participants were exposed to TDD. Creswell [12] suggests the following notation for it:

Group A X —— O

where group A represents a group of participants, X represents an exposure of a group to an experimental variable or event, and O represents an observation or measurement on an instrument. In this study, TDD was the treatment and Zorro was the instrument.

6.3.1 Participants

In this study, I recruited seven experienced Java developers who were familiar with unit testing.

6.3.2 Materials

The programming problem was the stack data structure. I gave the developers user stories which described the activities they were to perform in order to implement the stack.. Eclipse was the development tool. In addition, I also provided Doshi's TDD Rhythm [16] and TDD Quick Reference Guide [15] as supporting materials. In order to improve participants' commitment to TDD, a To-Do list was supplied as the reference.

6.3.3 Instrument

I instrumented participants' TDD development processes with the Hackystat Eclipse sensor and ESR. Participants were required to install the Hackystat Eclipse sensor and send the collected development activities to a remote Hackystat server. They also installed ESR which recorded their Eclipse windows as they participated in this study.

6.3.4 Procedure

Since the pilot study configuration was very simple, participants were given the option to either work on a lab computer or on their own computers at home. For those who opted to work at home, I provided detailed step-by-step instructions to them.

1. Setup

   Prior to the study I confirmed that the lab computer had the following software installed:

   - JDK
   - Eclipse IDE
   - Hackystat Eclipse Sensor [26]
   - Eclipse Screen Recorder [17]

   When participants chose to work at home on their own computers, I asked them to configure these software before participating in this study.

2. Introduction to TDD

   When participants did not have prior knowledge of TDD, I briefly introduced TDD to them using Beck's simple abstraction of TDD: Red/Green/Refactor.

3. Development in TDD

   Stack is a well-known data structure that works according to the Last-In-First-Out (LIFO) principle. The participants developed a program to implement the Stack data structure using TDD. I provided them with three documents: the graphic illustration of TDD rhythm, the TDD reference guide, and the user stories of stack with a To-Do list. They are available at Appendix A. In this step, both the Hackystat Eclipse sensor and ESR were turned on.

4. Data Collection

   The Hackystat Eclipse sensor collected and sent development activities to a remote Hackystat server. I asked participants' permissions to access their data for this study. For videos recorded by ESR, I asked participants to send them to me via email attachments.

6.4 Threats to Validity

There were several threats to construct validity. One of them was that some participants did not know TDD prior to the study. For these participants, I gave them a brief tutorial at the beginning of the study, and provided a graphic illustration of the rhythm [16] and the reference guide [15] of TDD. Another threat was the process conformance of TDD. To minimize its effects, I used Stack, a simple and well-known data structure, and provided user stories with a To-Do list to help participants comply with TDD.

With regard to validity of data collection, I used unobtrusive data collection utilities: Hackystat Eclipse Sensor and ESR. Both tools required a little overhead from participants [39, 35] at the beginning and at the end of the study.

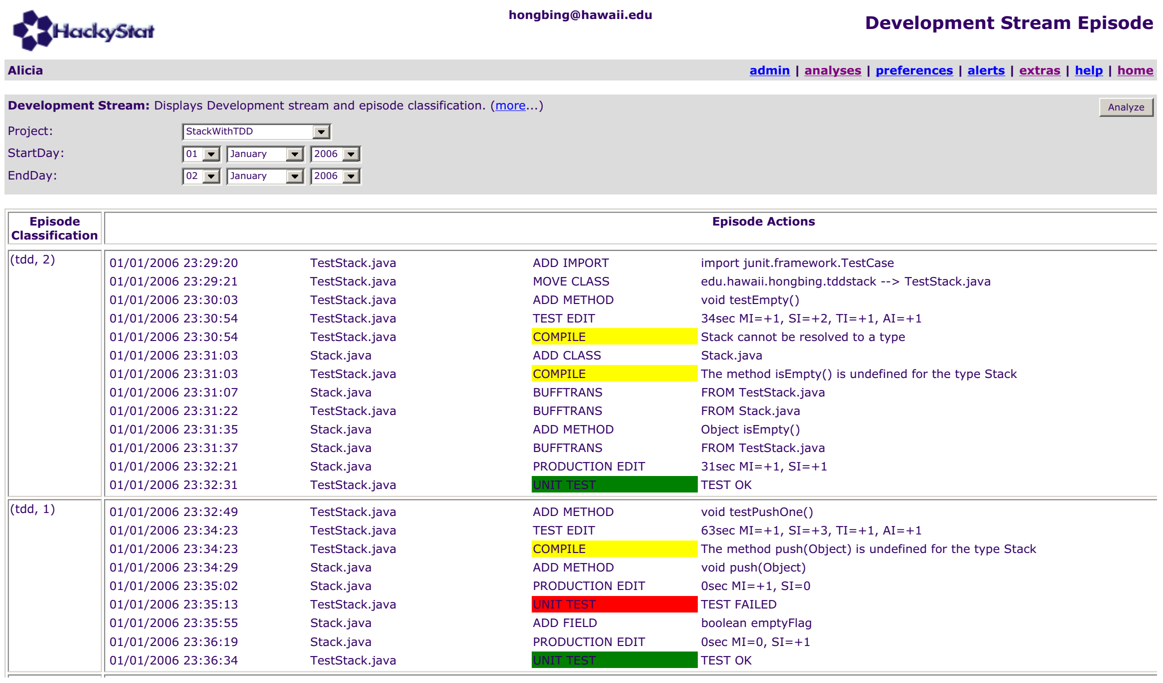
There were two external validity threats in this study. The first one was the simplicity and stringency of TDD. In the pilot study, I interpreted TDD as strictly as Beck suggested in [5, 6] and Doshi recommended in [16, 15]. The second one was that only 7 developers participated in this study. To address both of them, I conducted an extended validation study and an industrial case study in my thesis research following this pilot study.

6.5 Data Analyses

I collected two sources of data about TDD development in this study. The Hackystat Eclipse sensor collected low-level development activities that were used by Zorro to recognize participants' TDD development behaviors. The development process videos recorded by ESR were the second source of data that served as participant observation. I used the observational results from videos recorded by ESR to validate Zorro in data analyses.

6.5.1 Infer Development Behaviors with Zorro

To use Zorro, I defined a Hackystat project for each participant and then conducted the "TDD Development Stream" analysis provided by Zorro. Figure 6.1 illustrates the inferred results using my own data. Recall that Zorro partitions development streams using the "Test-Pass" tokenizer as described in Chapter 4, which yielded a sequence of "Test-Pass" episodes as shown in Figure 6.1.


hongbing@hawaii.edu

Development Stream Episode

Alicia

admin | analyses | preferences | alerts | extras | help | home

Development Stream: Displays Development stream and episode classification. (more...)

Analyze

Project:	StackWithTDD
StartDay:	01 January 2006
EndDay:	02 January 2006

Episode Classification	Episode Actions			
(tdd, 2)	01/01/2006 23:29:20	TestStack.java	ADD IMPORT	import junit.framework.TestCase
	01/01/2006 23:29:21	TestStack.java	MOVE CLASS	edu.hawaii.hongbing.tddstack --> TestStack.java
	01/01/2006 23:30:03	TestStack.java	ADD METHOD	void testEmpty()
	01/01/2006 23:30:54	TestStack.java	TEST EDIT	34sec MI=+1, SI=+2, TI=+1, AI=+1
	01/01/2006 23:30:54	TestStack.java	COMPILE	Stack cannot be resolved to a type
	01/01/2006 23:31:03	Stack.java	ADD CLASS	Stack.java
	01/01/2006 23:31:03	TestStack.java	COMPILE	The method isEmpty() is undefined for the type Stack
	01/01/2006 23:31:07	Stack.java	BUFFTRANS	FROM TestStack.java
	01/01/2006 23:31:22	TestStack.java	BUFFTRANS	FROM Stack.java
	01/01/2006 23:31:35	Stack.java	ADD METHOD	Object isEmpty()
	01/01/2006 23:31:37	Stack.java	BUFFTRANS	FROM TestStack.java
	01/01/2006 23:32:21	Stack.java	PRODUCTION EDIT	31sec MI=+1, SI=+1
	01/01/2006 23:32:31	TestStack.java	UNIT TEST	TEST OK
(tdd, 1)	01/01/2006 23:32:49	TestStack.java	ADD METHOD	void testPushOne()
	01/01/2006 23:34:23	TestStack.java	TEST EDIT	63sec MI=+1, SI=+3, TI=+1, AI=+1
	01/01/2006 23:34:23	TestStack.java	COMPILE	The method push(Object) is undefined for the type Stack
	01/01/2006 23:34:29	Stack.java	ADD METHOD	void push(Object)
	01/01/2006 23:35:02	Stack.java	PRODUCTION EDIT	0sec MI=+1, SI=0
	01/01/2006 23:35:13	TestStack.java	UNIT TEST	TEST FAILED
	01/01/2006 23:35:55	Stack.java	ADD FIELD	boolean emptyFlag
	01/01/2006 23:36:19	Stack.java	PRODUCTION EDIT	0sec MI=0, SI=+1
	01/01/2006 23:36:34	TestStack.java	UNIT TEST	TEST OK

Figure 6.1. TDD Development Stream Analysis

The “Episode Actions” column on the right displays episodes’ internal structures. Corresponding to the Red/Green/Refactor metaphor of TDD, Zorro highlights test failures in red and test passes in green backgrounds. In addition, compilation errors were highlighted in a yellow background. The “Episode Classification” column on the right presents development behaviors inferred by Zorro.

The inferred results were summarized in Table 6.1. For each participant, it lists the development duration and number of total, TDD, Refactoring, Test-Last and Unclassified episodes. According to

Table 6.1. Zorro’s Inference Results for Pilot Study

Subject ID	Duration	Episode	TDD	Refactoring	Test-Last	Unclassified
1	44:53	15	6	1	7	1
2	28:17	13	5	0	8	0
3	48:00	14	9	0	5	0
4	66:32	14	5	1	8	0
5	43:14	16	3	1	7	5
6	45:57	11	4	0	7	0
7	32:40	9	4	1	3	0
Total		92	36	4	45	6

Table 6.1, participants spent 28-45 minutes to implement Stack using TDD and yielded 92 episodes.

Zorro recognized 86 of them, which accounts for 93.6% of all episodes. Interestingly, among 6 unrecognizable episodes, 5 of them were from one participant only. It was also notable that participants almost never refactored, and they did “Test-Last” half of the time (in the unit of episode number). Here “Test-Last” means that participants write test code after production code has been implemented, which is the opposite side of TDD.

In the pilot study Zorro inferred TDD behaviors as “TDD”, “Refactoring” or “Test-Last”. This development behaviors classification reflected Beck’s [6] simple abstraction of TDD. Further research found that this classification can not represent actual TDD developments, and thus a more sophisticated TDD development behaviors classification schema was developed as described in Chapter 4.

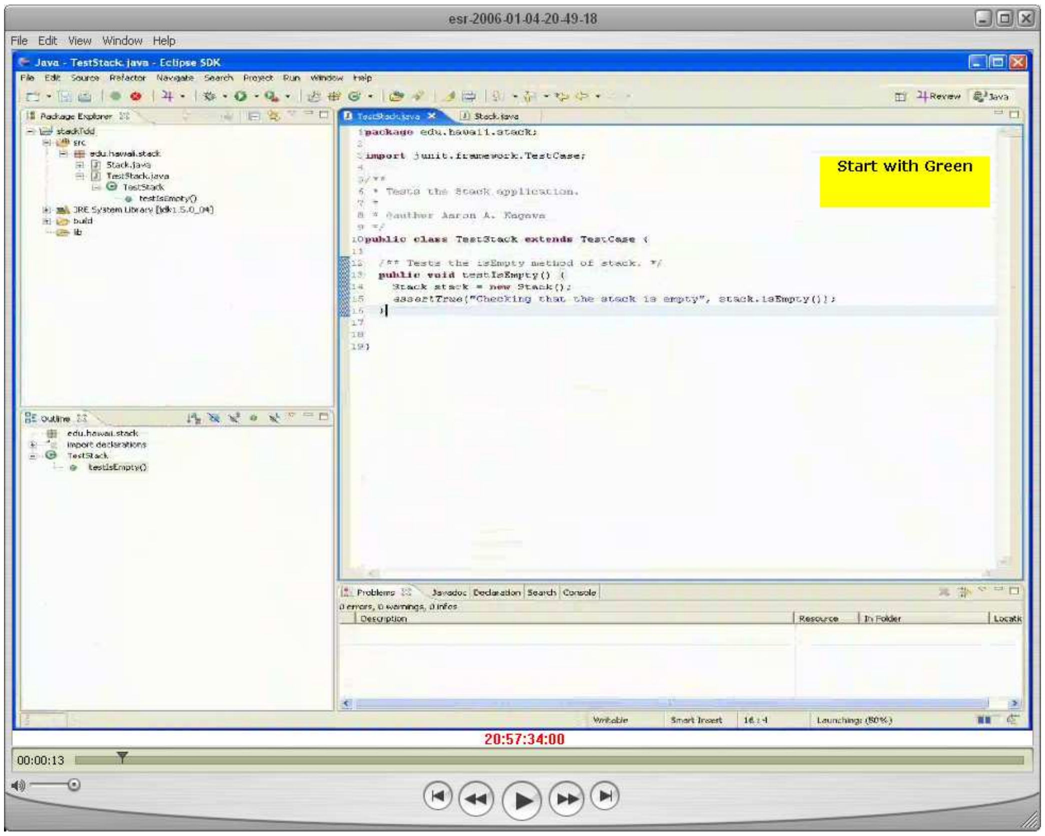
6.5.2 Participant Observation

An ESR video contains time-stamped Eclipse windows that were captured at the rate of one frame per second. I played and watched the recorded videos as a form of participant observation. Figure 6.2 is a screen-shot showing an ESR video that I played using QuickTime[58]. At the time when Figure 6.2 was captured, the participant who produced the video had just finished a TDD iteration ending with a successful unit test invocation. The video also includes a time-line at the bottom indicating the time when the window was captured. Note that the comment at the top-right corner in Figure 6.2 was not part of the video. I added this remark in my observation analysis.

When I observed a TDD-realted activity in the ESR videos, I recorded it into an Excel spreadsheet for bookkeeping (See Figure 6.3). Each activity has a start time, an end time, an abstract, and additional annotations. As seen in Figure 6.3, activities such as compilations, failed test invocations and successful test invocations were highlighted in contrast to Zorro’s TDD Development Stream analysis results as shown in Figure 6.1.

6.5.3 Validation of Zorro’s Data Collection

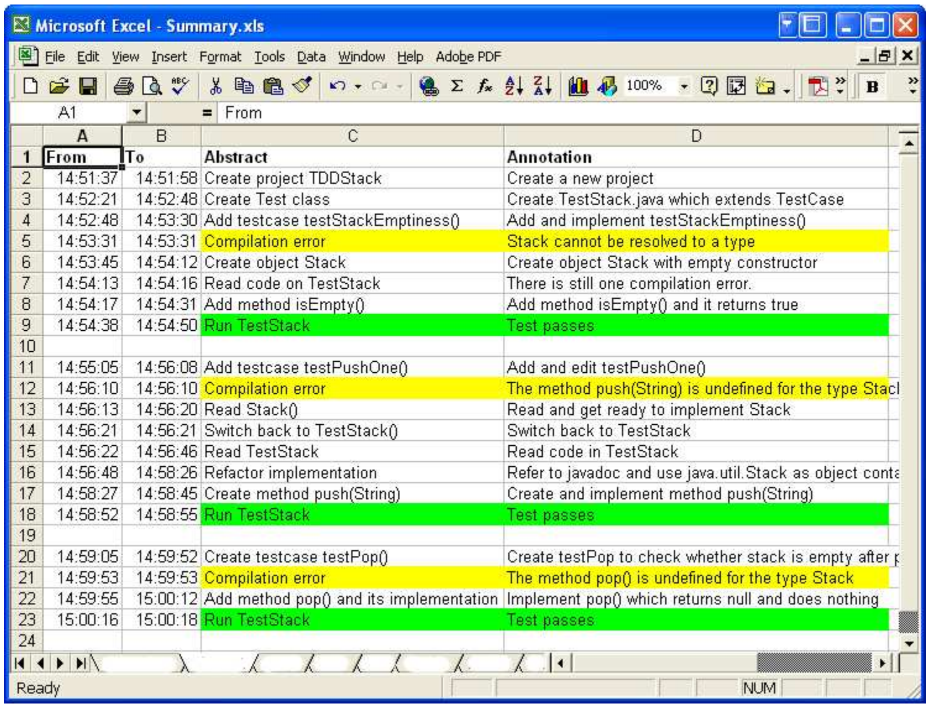
I used activities observed from the ESR videos to validate Zorro’s data collection. The comparison between activities recorded by ESR and activities collected by Zorro allowed me to perform a partial validation of Zorro by determining which activities were missed by Zorro and which activ-



```
package edu.hawaii.stack;

import junit.framework.TestCase;

/**
 * Tests the Stack application.
 *
 * @author Aaron A. Kagawa
 */
public class TestStack extends TestCase {

  /** Tests the isEmpty method of stack. */
  public void testIsEmpty() {
    Stack stack = new Stack();
    assertTrue("Checking that the stack is empty", stack.isEmpty());
  }


}
```

```
0 errors, 0 warnings, 0 infos
```

Figure 6.2. ESR Video


Microsoft Excel - Summary.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
A1
From
Ready
NUM

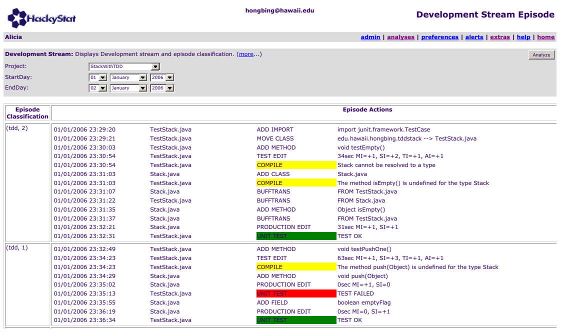
	A	B	C	D
1	From	To	Abstract	Annotation
2	14:51:37	14:51:58	Create project TDDStack	Create a new project
3	14:52:21	14:52:48	Create Test class	Create TestStack.java which extends TestCase
4	14:52:48	14:53:30	Add testcase testStackEmptiness()	Add and implement testStackEmptiness()
5	14:53:31	14:53:31	Compilation error	Stack cannot be resolved to a type
6	14:53:45	14:54:12	Create object Stack	Create object Stack with empty constructor
7	14:54:13	14:54:16	Read code on TestStack	There is still one compilation error.
8	14:54:17	14:54:31	Add method isEmpty()	Add method isEmpty() and it returns true
9	14:54:38	14:54:50	Run TestStack	Test passes
10				
11	14:55:05	14:56:08	Add testcase testPushOne()	Add and edit testPushOne()
12	14:56:10	14:56:10	Compilation error	The method push(String) is undefined for the type Stacl
13	14:56:13	14:56:20	Read Stack()	Read and get ready to implement Stack
14	14:56:21	14:56:21	Switch back to TestStack()	Switch back to TestStack
15	14:56:22	14:56:46	Read TestStack	Read code in TestStack
16	14:56:48	14:58:26	Refactor implementation	Refer to javadoc and use java.util.Stack as object conta
17	14:58:27	14:58:45	Create method push(String)	Create and implement method push(String)
18	14:58:52	14:58:55	Run TestStack	Test passes
19				
20	14:59:05	14:59:52	Create testcase testPop()	Create testPop to check whether stack is empty after p
21	14:59:53	14:59:53	Compilation error	The method pop() is undefined for the type Stack
22	14:59:55	15:00:12	Add method pop() and its implementation	Implement pop() which returns null and does nothing
23	15:00:16	15:00:18	Run TestStack	Test passes
24				

Figure 6.3. Observation Results in Excel

ities were not correctly collected by Zorro. This section introduces the analysis method, followed by analysis results.

Validation Analysis Method

After watching each participant's video, I compared the activities observed by me to activities collected by Zorro and presented by the TDD Development Stream analysis. Figure 6.4 illustrates the comparison for a participant. The sub-figure on the left is a screen-shot showing activities collected by Zorro and the sub-figure on the right is the Excel spreadsheet with development activities I observed for a programming period. Using this comparison method, I validated Zorro's data collection for all of the development streams produced by pilot study participants. The next section presents this analysis result.



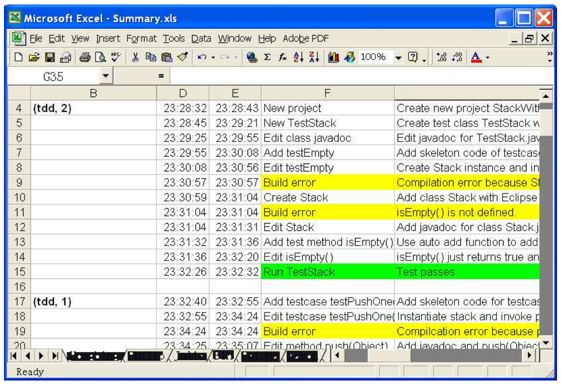
HackyStat hongbing@hawaii.edu **Development Stream Episode**

Alicia admin | analyses | preferences | alerts | extras | help | home

Development Stream: Displays Development stream and episode classification. (more...)

Project: StackWithTDD
StartDay: 01 January 2006
EndDay: 02 January 2006

Episode Classification	Episode Actions			
(tdd, 2)	01/01/2006 23:29:20	TestStack.java	ADD IMPORT	import junit.framework.TestCase
	01/01/2006 23:29:21	TestStack.java	MOVE CLASS	edu.hawaii.hongbing.tddstack --> TestStack.java
	01/01/2006 23:30:03	TestStack.java	ADD METHOD	void testEmpty()
	01/01/2006 23:30:54	TestStack.java	TEST EDIT	34sec MI=+1, SI=+2, TI=+1, AI=+1
	01/01/2006 23:30:54	TestStack.java	COMPILE	Stack cannot be resolved to a type
	01/01/2006 23:31:03	Stack.java	ADD CLASS	Stack.java
	01/01/2006 23:31:03	TestStack.java	COMPILE	The method isEmpty() is undefined for the type Stack
	01/01/2006 23:31:07	Stack.java	BUFFTRANS	FROM TestStack.java
	01/01/2006 23:31:22	TestStack.java	BUFFTRANS	FROM Stack.java
	01/01/2006 23:31:35	Stack.java	ADD METHOD	Object isEmpty()
	01/01/2006 23:31:37	Stack.java	BUFFTRANS	FROM TestStack.java
	01/01/2006 23:32:21	Stack.java	PRODUCTION EDIT	31sec MI=+1, SI=+1
	01/01/2006 23:32:31	TestStack.java	UNIT TEST	TEST OK
(tdd, 1)	01/01/2006 23:32:49	TestStack.java	ADD METHOD	void testPushOne()
	01/01/2006 23:34:23	TestStack.java	TEST EDIT	63sec MI=+1, SI=+3, TI=+1, AI=+1
	01/01/2006 23:34:23	TestStack.java	COMPILE	The method push(Object) is undefined for the type Stack
	01/01/2006 23:34:29	Stack.java	ADD METHOD	void push(Object)
	01/01/2006 23:35:02	Stack.java	PRODUCTION EDIT	0sec MI=+1, SI=0
	01/01/2006 23:35:13	TestStack.java	UNIT TEST	TEST FAILED
	01/01/2006 23:35:55	Stack.java	ADD FIELD	boolean emptyFlag
	01/01/2006 23:36:19	Stack.java	PRODUCTION EDIT	0sec MI=0, SI=+1
	01/01/2006 23:36:34	TestStack.java	UNIT TEST	TEST OK



Microsoft Excel - Summary.xls

G35 =

	B	D	E	F	
4	(tdd, 2)	23:28:32	23:28:43	New project	Create new project StackWit
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 S
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

[redacted]

Ready

Figure 6.4. Comparison of Development Activities between Zorro and ESR

Validation Results

Overall Zorro was capable of capturing development activities. Compared to activities observed from ESR videos, Zorro collected almost all of them, and thus it was pointless to literally report number of missed and incorrectly collected activities. Instead, I decided to summarize types of missed and incorrectly collected activities that affected Zorro's inference. The following is a summary of problems I found with regard to Zorro's data collection:

- **Problem 1**: Insignificant editing work.

Severity:	*High*
Reason:	*Editing work did not change object metrics such as statements and methods. Or developers quickly edited code, which resulted in one state change event only.*
Result:	*Episodes were misclassified since editing activities were not captured.*
Resolution:	*Change the implementation of file edit sub stream in SDSA to look for file size change as well.*
Affected:	*6 episodes.*

- **Problem 2**: Missed compilation errors to test code.

Severity:	*Medium*
Reason:	*Changes to production code caused compilation errors to inactive test code.*
Results:	*Episode were misclassified.*
Resolution:	*Fix the Hackystat Eclipse sensor to report compilation errors on inactive files as well.*
Affected:	*2 episodes*

- **Problem 3**: Two unit test invocations were grouped together or one unit test invocation was divided into two continuous episodes.

Severity:	*Medium*
Reason:	*Eclipse sensor collected multiple data entries for one test invocation.*
Results:	*Two or more episodes were grouped together or divided resulting that they were not classified correctly.*
Resolution:	*Tag unit tests with run time to group multiple unit test entries belong to one test invocation together.*
Affected:	*3 episodes*

Of the three problems listed above, one was high and the other two were medium in severity. In total, they affected 11 out of 92 episodes.

6.5.4 Validation of Zorro's TDD Behaviors Inference

As seen in Figure 6.4 and Table 6.1, Zorro inferred development behaviors as "TDD", "Refactoring" or "Test-Last". For each behavior category, Zorro also defined sub-types as seen in Figure 6.4, but I will not discuss them in this analysis because sub-types are for Zorro's internal uses only. In this analysis, I compared development behaviors I observed in ESR videos to development behaviors inferred by Zorro.

Validation Analysis Method

In Section 6.5.2, I introduced how I did participant observation using videos. After observing low-level development activities, I played the videos again to derive development behaviors of TDD. The remark in a yellow background at the top-right corner of Figure 6.2 was a visual presentation of my derivation of development behaviors. In this data analysis, similar to what I have done in Section 6.5.3, I compared the development behaviors derived from ESR videos to development behaviors inferred by Zorro for validation.

Validation Results

Table 6.2 is a summary of the validation results. For each participant, it lists number of total, classified and wrongly classified episodes along with the percentage of classification errors. According to Table 6.2, 11.6% of episodes were wrongly classified. On the other hand, it indicated that Zorro inferred TDD behaviors correctly for 88.4% of episodes.

Table 6.2. TDD Development Behavior Validation

Subject ID	Episode	Classified	Wrongly Classified	Percentage
1	15	14	2	13.3%
2	13	13	3	23.3%
3	14	14	1	7.1%
4	14	14	1	7.1%
5	16	11	1	9.1%
6	11	11	1	9.1%
7	9	9	1	12.5%
Total	92	86	10	11.6%

In addition, the validation analysis helped me to discover an inference problem for very long episodes that occurred when developers did not invoke unit tests frequently. It is described in the following:

- **Problem 4**: An episode had too many activities.

Severity:	*Low*
Reason:	*Participants did not invoke unit testing frequently enough.*
Results:	*Episodes were misclassified.*
Resolution:	*Introduce long episode type and avoid inferring episode with too many activities.*
Affected:	*2 episodes*

This behavior is clearly a violation to TDD's short-duration characteristic, but Zorro did not have an episode behavior category for it when I conducted the pilot study. Later I included this behavior in the new development behavior classification schema in current version of Zorro (See Chapter 4).

6.6 Research Conclusions

The above analyses show that the one-shot case study research strategy is useful for Zorro validation. ESR, an Eclipse screen recording tool, is capable of recording incremental small changes made by participants for the purpose of participant observation. Although ESR caused a small delay when it was initialized, participants did not notice much delay in the pilot study. With ESR videos, I was able to validate both Zorro's data collection and TDD behaviors inference. Thus, ESR is suitable for Zorro validation studies and the research question Q1a is answered.

Using videos recorded by ESR for participant observation, I found that Zorro had 3 types of solvable data collection problems. The validation analysis of Zorro's data collection shows that Zorro can collect sufficient low-level development activities accurately for the purpose of TDD development behavior inference. This supports the research question Q1b.

Only two out of 93 episodes were incorrectly inferred by Zorro because they had too many activities in them. Other than this, Zorro's inference rules recognized TDD development behaviors correctly when low-level development activities were sufficient. Thus this pilot study provides the supporting evidence to research question Q1c.

6.7 Discussion and Zorro Improvements

In a simple environment setting, I validated that Zorro worked well at collecting low-level development activities and inferring developer's TDD behaviors. However, as described above, this pilot study also identified several areas that could be improved.

6.7.1 Data Collection

Section 6.5.3 addressed three data collection problems that prevented Zorro from inferring TDD behaviors correctly. Following the pilot study, I fixed them in the current version of Zorro.

6.7.2 TDD Behaviors Classification

The validation analysis in Section 6.5.4 showed that applying inference rules on episodes that had too many development activities caused inference errors. More interestingly, about 50% of episodes were "Test-Last" in the pilot study. There are several possibilities that could explain this phenomenon. One possibility could be that the programming problem (Stack) is too simple and developers did not need to fail tests first to have the correct implementation. Another possibility could be that Beck's concise summary of TDD is just too simple, while real TDD development is much more complicated than what he described. For instance, a developer can add a new test that does not fail initially because the functional code works well even without any change. This development behaviors is none of "TDD", "Refactoring" and "Test-Last". Therefore, after the pilot study, I developed a much sophisticated TDD behaviors classification schema as seen in Table 4.2 that can best describe real development behaviors. The long episode behavior is part of this schema.

6.7.3 Process Conformance Inference

In the pilot study, I did not directly calculate process conformance of TDD. Instead, I used development behaviors to describe process conformance: "TDD" and "Refactoring" behaviors were TDD conformant automatically while "Test-Last" was not. According to this simple measurement and Table 6.1, in the pilot study, only less than 50% (40 out of 92 episodes) of episodes were TDD conformant, which was very contrary to what I had anticipated before the study. Further research indicated that this simple measurement was very limited.

With the introduction of the new episode classification schema, I defined a more sophisticated two-step model for process conformance of TDD (See Figure 6.5) using heuristics. The first step is to infer development behaviors in episodes and then look up episode context to determine their process conformance. There are three lists in Figure 6.5. The left-most one is a list of development

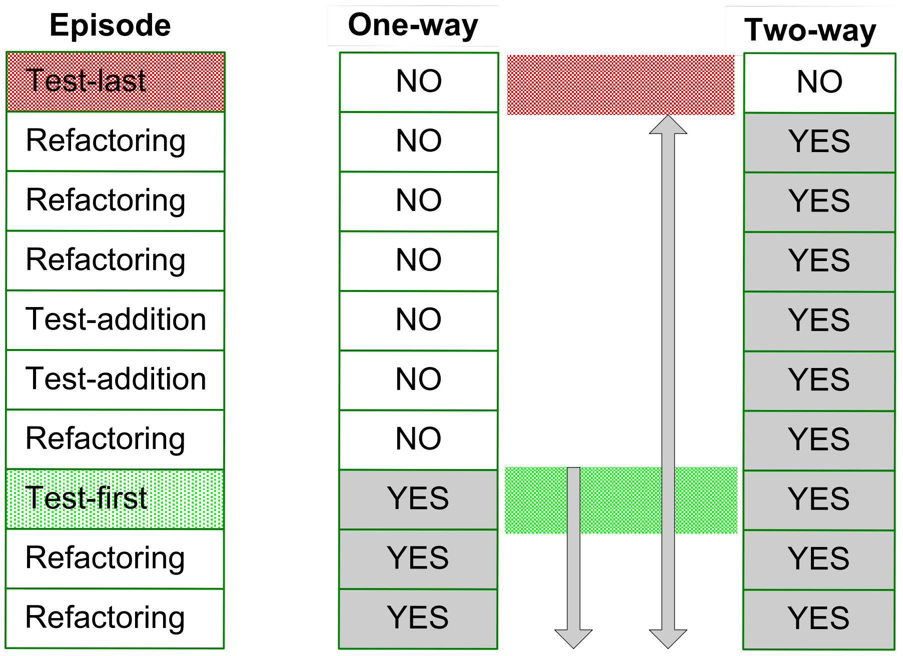


Figure 6.5. Heuristic Algorithms of TDD Conformance

behaviors recognized by Zorro's inference rules. As their names indicate, the episodes can be "test-first", "test-addition", "refactoring", or "test-last" etc. The one-way and two-way TDD heuristic algorithms are on the right side of Figure 6.5. The one-way algorithm uses look-forward approach to determine whether an episode is TDD conformant, while the two-way heuristic algorithm uses both look-forward and look-backward approaches. Figure 6.5 indicates this difference using a single-head arrow and a double-head arrow. I implemented these heuristic rules with the support of JESS [22]. Section 4.2.4 in Chapter 4 has a detailed description to this heuristic. Our preliminary work suggests that the two-way heuristic algorithm can understand real world situations better than the one-way algorithm.

6.8 Chapter Summary

This chapter introduced the pilot study, a test to investigate how to validate Zorro using participant observation supported by ESR. The study showed that Zorro's data collection and TDD behaviors inference can be validated by analyzing development videos recorded by ESR. This study also identified several problems that helped to improve Zorro. With these improvements, I conducted an extended classroom case study following the pilot study in my thesis research.

Chapter 7

Classroom Case Study

Following the pilot study and collaborative research with Dr. Hakan Erdogmus[1], I improved Zorro and conducted an extended validation study in the software engineering classes at the University of Hawaii in Fall 2006. This study shows that Zorro can collect low-level development activities and infer TDD development behaviors accurately. This study also provides evidence that Zorro is helpful for beginners who do not have much prior experience with the TDD practice.

7.1 Purpose of the Study

The purpose of this study was to validate Zorro using the case study research method tested by the pilot study (Chapter 6) and investigate how useful Zorro is for TDD beginners.

7.2 Research Questions

The specific research questions for the classroom case study were:

- Q2a: Does Zorro collect software development activities accurately enough for episode partitioning and TDD behavioral inference?

[1]Dr. Hakan Erdogmus is a Senior Research Officer in the NRC-IIT Software Engineering Group (http://iit-iti.nrc-cnrc.gc.ca/personnel/erdogmus_hakan_e.html). He has interests in Software Economics, Agile Software Development, and Software Process Measurement and Awareness.

- Q2b: Does Zorro's inference of TDD behaviors agree with analyses based upon participant observation?
- Q2c: Does Zorro's inference of TDD behaviors agree with what participants believe to be their TDD behaviors?
- Q2d: Does Zorro provide useful information for beginners to understand TDD and improve their TDD development?

Research questions Q2a, Q2b, and Q2c are three specific research questions for the thesis research question Q1, "*Can Zorro automate the recognition of TDD behaviors using automatically collected software metrics?*". The research question Q2d corresponds to the thesis research question Q2, "*How useful is Zorro?*"

7.3 Experiment Design

To address the above research questions, I used the one-shot case study research method as in the pilot study. TDD was the treatment. We taught TDD to participants and asked them to develop software using TDD in this study.

7.3.1 Participants

TDD beginners were the targeted population. I recruited participants from a senior-level software engineering class and a graduate-level software engineering class, both of which were taught by Professor Philip Johnson in Fall 2006 at the University of Hawaii. There were 16 students in two classes and 11 of them agreed to participate in this study.

7.3.2 Materials

I conducted this study in the Collaborative Software Development Laboratory (CSDL) at the University of Hawaii. Three Windows-based lab computers were used, one of them acted as a Hackystat server and the other two were used by participants to develop software for this study. I configured two development computers with necessary software including JDK, Eclipse, the Hackystat Eclipse Sensor, QuickTime and ESR. Bowling Score Keeper, a famous problem that has been widely used in empirical evaluation research of TDD[23, 19], was the programming task. Since participants were TDD beginners, I designed user stories in plain text (Appendix E) to help them develop the program without needing prior experience with the game. Since user stories are ordered from the easiest one to the hardest one, they can also be used as the "To Do" list for TDD development.

7.3.3 Instruments

I instrumented the development processes with the Hackystat Eclipse sensor and ESR. Prior to the study, I created a Hackystat key for each participant and configured the Hackystat Eclipse sensor for him/her. At the beginning of the study, I asked participants to enable ESR so that it would record their development processes. Participants validated Zorro's TDD behavioral inference using the "TDD Episode Validation" (Figure 7.2) analysis. The "Web Validation Wizard" was another instrumentation tool that I used to collect participant's evaluation. Additionally I interviewed participants in the study and recorded the interviews using a digital voice recorder.

7.3.4 Procedure

This experiment included a TDD lecture and a 2-hour lab session.

1. TDD Lecture

   In both software engineering classes, Professor Philip Johnson gave the same TDD lecture to students. The lecture included following contents:

   - Introduction to TDD
     - Two principles of TDD from [6]
     - Red/green/refactor pattern of TDD
     - TDD Rhythm [16]
     - TDD vs. Unit Testing
     - A TDD example: implementing stack by writing test first

- Why TDD?
  - Developer gets quick feedback.
  - TDD improves software quality.
  - TDD promotes simple design.
  - Microsoft has successful story on TDD [7]
  - Test-Driven Development proves useful at Google[72]
- About TDD
  - TDD may not be appropriate for everybody.
  - TDD is about design.
  - Some studies show that TDD improves software quality.
  - TDD may reduce productivity.
  - TDD references including testdriven.com, mailing list and blogs.
- Reading and programming assignments
  - Page 1-20 of Beck's book "Test-Driven Development by Example" [6]
  - TDD Quick Reference [15]
  - Practice TDD on Roman Numeral Problem (Appendix C)

After the lecture, students practiced TDD by working on an in-class assignment, the Roman Numeral Conversion (Appendix C). Then they were asked to voluntarily participate in this study. The study was conducted in CSDL under my supervision. The 2-hour lab session included 90-minute of development using TDD, Episode Validation, Participant Interview and Zorro Usefulness Evaluation.

At the beginning of the study, I introduced the purpose and content of this research study to participants, followed by the consent form signing. Then I gave them user stories of the Bowling Score Keeper problem (Appendix E) and explained that they only needed to spend 90 minutes on the programming task. Following a five minute introduction, I asked them to start up Eclipse, enable the ESR recorder and begin programming.

2. TDD Development (90 minutes)

   Each participant developed a program to compute bowling game scores in TDD following the provided user stories. I suggested that they use the user stories as a To-Do list to help them develop in TDD. The time limit was 90 minutes. It was acceptable if they did not finish all user stories by the end within the 90 minutes.

3. Validating Zorro's Inference about Their TDD Behaviors (15 minutes)

   After 90 minutes of development in TDD, participants exited Eclipse, which forced the Eclipse sensor to send all remaining development activities to the Hackystat server and stopped the ESR recorder. Following a five minutes coffee break, I then directed them to login into the Hackystat server and asked them to invoke the "TDD Episode Validation" (Figure 7.2) analysis in which they provided their feedback to Zorro's inference results. As seen in Figure 7.2, they commented on Zorro's inference of development behaviors and compliance to TDD.

4. Interview (5 minutes)

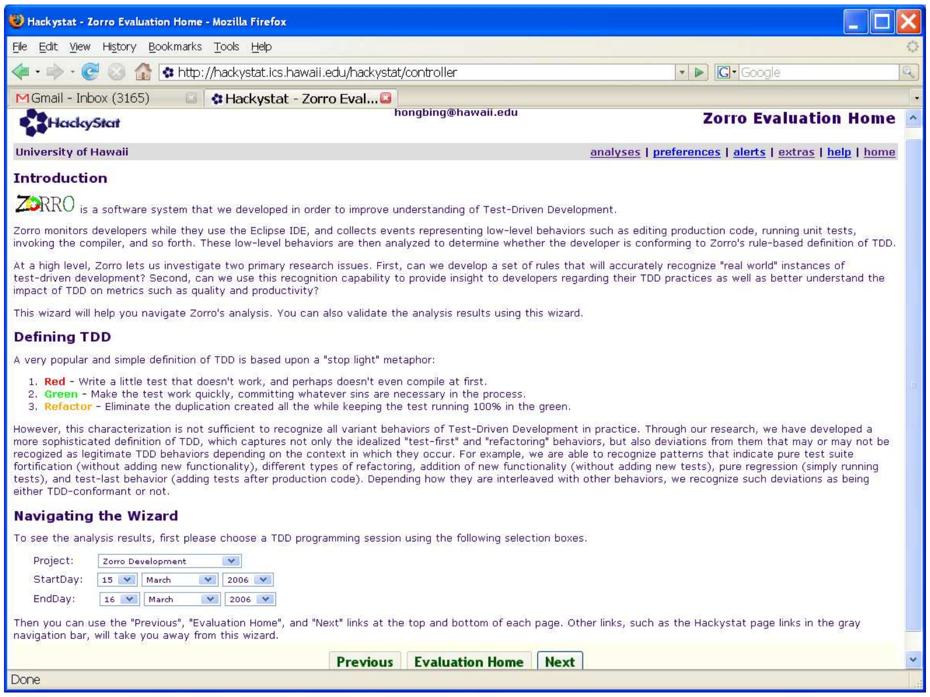
   I conducted an interview with each participant after he/she finished reviewing Zorro's inference of their development behaviors. The interview lasted 5 minutes. In the interview, I asked participants questions regarding their development experiences on software engineering best practices, unit testing, TDD and software quality.

5. Zorro Usefulness Evaluation (10 minutes)

   At the end of the lab session, participants conducted several analyses using the "Evaluation Wizard" (Figure 7.1) provided by Zorro to evaluate its usefulness. Participants used the wizard to review five analyses that were created based on Zorro's inference: Episode Demography, T/P Effort Ratio, T/P Size Ratio, Episode Duration and Episode Duration Bin.

7.3.5 Data Collection

Similarly to the pilot study, I collected development activities using the Hackystat Eclipse sensor and development process videos using ESR. Development activities were saved on the Hackystat server, and ESR videos were saved on the lab PCs. To collect participants' validation comments, Zorro provides the "TDD Episode Validation" analysis (Figure 7.2), which is an extension to the "TDD Development Stream" analysis (Figure 6.1) discussed in Section 6.5. In addition to presenting inference results, this analysis also supplies Zorro's reasoning process and allows participants to give feedback, which was saved to the Hackystat server. Participants conducted Zorro usefulness evaluation using the "Evaluation Wizard" analysis (Figure 7.1) that saved the evaluation results to the Hackystat server too.


Hackystat - Zorro Evaluation Home - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Gmail - Inbox (3165)
Hackystat - Zorro Eval...
Done

HackyStat hongbing@hawaii.edu **Zorro Evaluation Home**

University of Hawaii analyses | preferences | alerts | extras | help | home

Introduction

ZORRO is a software system that we developed in order to improve understanding of Test-Driven Development.

Zorro monitors developers while they use the Eclipse IDE, and collects events representing low-level behaviors such as editing production code, running unit tests, invoking the compiler, and so forth. These low-level behaviors are then analyzed to determine whether the developer is conforming to Zorro's rule-based definition of TDD.

At a high level, Zorro lets us investigate two primary research issues. First, can we develop a set of rules that will accurately recognize "real world" instances of test-driven development? Second, can we use this recognition capability to provide insight to developers regarding their TDD practices as well as better understand the impact of TDD on metrics such as quality and productivity?

This wizard will help you navigate Zorro's analysis. You can also validate the analysis results using this wizard.

Defining TDD

A very popular and simple definition of TDD is based upon a "stop light" metaphor:

1. Red - Write a little test that doesn't work, and perhaps doesn't even compile at first.
2. Green - Make the test work quickly, committing whatever sins are necessary in the process.
3. Refactor - Eliminate the duplication created all the while keeping the test running 100% in the green.

However, this characterization is not sufficient to recognize all variant behaviors of Test-Driven Development in practice. Through our research, we have developed a more sophisticated definition of TDD, which captures not only the idealized "test-first" and "refactoring" behaviors, but also deviations from them that may or may not be recognized as legitimate TDD behaviors depending on the context in which they occur. For example, we are able to recognize patterns that indicate pure test suite fortification (without adding new functionality), different types of refactoring, addition of new functionality (without adding new tests), pure regression (simply running tests), and test-last behavior (adding tests after production code). Depending how they are interleaved with other behaviors, we recognize such deviations as being either TDD-conformant or not.

Navigating the Wizard

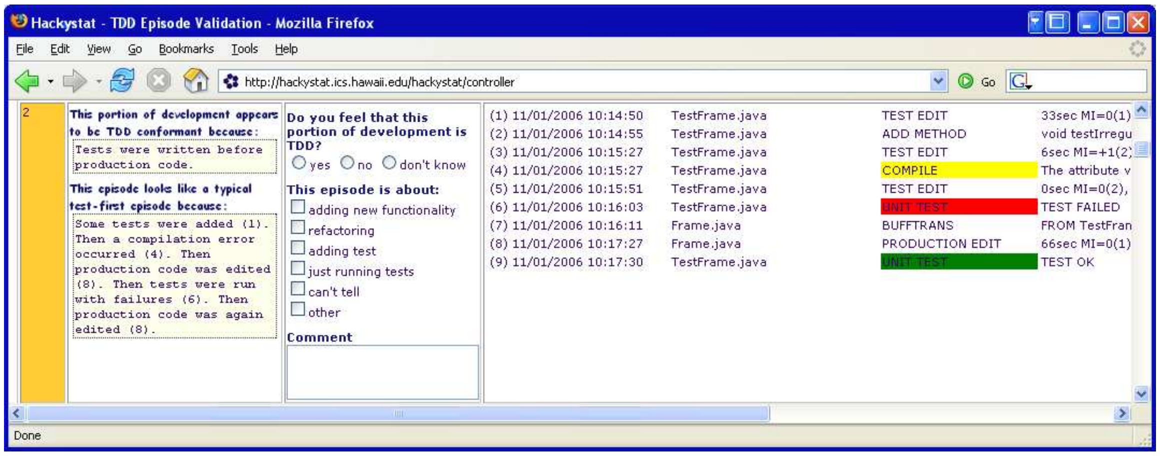
To see the analysis results, first please choose a TDD programming session using the following selection boxes.

Project:	Zorro Development		
StartDay:	15	March	2006
EndDay:	16	March	2006

Then you can use the "Previous", "Evaluation Home", and "Next" links at the top and bottom of each page. Other links, such as the Hackystat page links in the gray navigation bar, will take you away from this wizard.

Previous | Evaluation Home | Next

Figure 7.1. Zorro Evaluation Wizard


Hackystat - TDD Episode Validation - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Go
Done

2

This portion of development appears to be TDD conformant because:

Tests were written before production code.

This episode looks like a typical test-first episode because:

Some tests were added (1). Then a compilation error occurred (4). Then production code was edited (8). Then tests were run with failures (6). Then production code was again edited (8).

Do you feel that this portion of development is TDD?

○ yes ○ no ○ don't know

This episode is about:

☐ adding new functionality
☐ refactoring
☐ adding test
☐ just running tests
☐ can't tell
☐ other

Comment

(1) 11/01/2006 10:14:50	TestFrame.java	TEST EDIT	33sec MI=0(1)
(2) 11/01/2006 10:14:55	TestFrame.java	ADD METHOD	void testIrregu
(3) 11/01/2006 10:15:27	TestFrame.java	TEST EDIT	6sec MI=+1(2)
(4) 11/01/2006 10:15:27	TestFrame.java	COMPILE	The attribute v
(5) 11/01/2006 10:15:51	TestFrame.java	TEST EDIT	0sec MI=0(2),
(6) 11/01/2006 10:16:03	TestFrame.java	UNIT TEST	TEST FAILED
(7) 11/01/2006 10:16:11	Frame.java	BUFFTRANS	FROM TestFran
(8) 11/01/2006 10:17:27	Frame.java	PRODUCTION EDIT	66sec MI=0(1)
(9) 11/01/2006 10:17:30	TestFrame.java	UNIT TEST	TEST OK

Figure 7.2. TDD Episode Validation

7.4 Threats to Validty

A construct validity problem existed in the pilot study. I, the author of Zorro, compared the recorded movies with Zorro's inference to validate Zorro's metrics collection and TDD development behavioral inference. I could be biased both at judging what software metrics are necessary, as well as at inferring development behaviors from the observed activities in the recorded movies. One person's subjective judgment, especially the one from the author, perhaps is not a valid measure in case studies [78]. Therefore, I employed the additional evidence of participant comments to improve the construct validity. The participant comments were analyzed to cross-validate my video analysis. Compared to the pilot study, this additional source of evidence helped to solidify research conclusions, but note that a caveat did exist because the student participants could have given feedback that favored Zorro's behavioral inference.

It is challenging to validate Zorro because participants might not want to be instrumented by the sensors, not to mention having their development processes recorded with ESR. In my study, I used two lab PCs and carefully stated that participants were not evaluated based upon their performances, and they gained extra credit as long as they participated in this study. Moreover, participants' identities were not disclosed in any written documents. The consent form I used is available at the Appendix D.

In term of external validity, only 11 students participated in this study. The sample size is small, which makes it difficult for me to generalize these research findings. Also, students in the software engineering classes at the University of Hawaii might not be representative of all TDD beginners since professional developers can also be TDD beginners. To improve external validity, replication studies need to be conducted in other organizations. For this purpose, I made all my research materials available to public.

7.5 Data Analysis Methods for An Individual Participant

I processed every participant's data and drew research conclusions by putting analysis results of all participants together. This section introduces the analysis methods for an individual participant using the first participant's data.

7.5.1 Participant Observation and Validation of Data Collection

The first step was to observe the first participant by playing his development video recorded by ESR. This observation helped to validate Zorro's software metrics collection for him.

Analysis Method

Similarly to the pilot study, I observed low-level development activities by playing the recorded ESR video from the first participant. Then I logged what I observed into an Excel spreadsheet for bookkeeping (See Figure 6.3 in Chapter 6). Recall that SDSA constructs the development stream using software metrics collected by sensors (Section 3.2.1), so we can also construct another development stream using development activities that occurred in the participant's development process and that I observed in the recorded development video. To validate Zorro's data collection, I simply compared two development streams that were illustrated in Figure 6.4 in Chapter 6.

Analysis Result

The first participant finished 7 of the 13 user stories in 90 minutes. I observed that he conducted 153 development activities, which were then divided into 10 episodes. For each episode, Table 7.1 lists both the number of activities collected by Zorro and the number of activities I observed. The difference between the two numbers is in the "Difference" column. Descriptive analysis results are available at the bottom of this table.

Note that ESR captures the Eclipse window once per second, so it should capture almost everything that happens in a programming session. However, according to Table 7.1, activities collected by Zorro outnumbered activities I observed in the recorded video of the same development process conducted by the first participant. Moreover, the differences in activity numbers were significant according to the descriptive analysis. Zorro collected 19.8 activities per episode, which were 4.5 more than the number of activities per episode I observed in the video.

Table 7.1. Number of Development Activities

Episode	Activities (Zorro)	Activities (Video)	Difference
1	4	4	0
2	6	5	+1
3	13	11	+2
4	19	15	+4
5	23	19	+4
6	14	9	+5
7	46	35	+11
8	22	15	+7
9	5	5	0
10	46	35	+11
Total	198	153	
Mean			4.5
Median			4
STDEV			4.1

7.5.2 Validation of TDD Behaviors and TDD Compliance Inference

The second step was to validate Zorro's inference of TDD development behaviors and TDD compliance.

Analysis Method

In Section 7.5.1, I introduced how I observed development activities in the recorded video. In this step, I played the video again to observe TDD development behaviors of the first participant. Following this observation, I analyzed the TDD compliance of each episode. Similarly to what I did in Section 7.5.1, I validated Zorro's inference results of TDD behaviors and TDD compliance by comparing what Zorro inferred to what I observed.

Analysis Result

I list the analysis result in Table 7.2 in which Zorro's inference results are on the left and my observation analysis results are on the right. Recall that Zorro infers development behaviors in episodes first, and then infers TDD compliance based upon development context using heuristic algorithms (Section 4.2.4). Therefore, I present Zorro's inference results in two columns in Table

7.2: one is for TDD development behaviors and the other one is for TDD compliance. In order to compare Zorro's inference results to my observation results, I also list the observation results in two columns: one is for development behaviors and the other one is for TDD compliance.

Table 7.2. Comparison between Zorro Inference and Video Observation

Index	Zorro		Observation	
	Behavior	Is TDD?	Behavior	Is TDD?
1	test-addition	Yes	test-addition	Yes
2	refactoring	Yes	refactoring	Yes
3	refactoring	Yes	refactoring	Yes
4	test-first	Yes	test-first	Yes
5	test-first	Yes	test-first	Yes
6	test-first	Yes	test-first	Yes
7	test-first	Yes	test-first	Yes
8	test-first	Yes	test-first	Yes
9	test-addition	Yes	test-addition	Yes
10	unknown	No	test-first	Yes

From Table 7.2, we can see that participant observation results are identical to Zorro's inference results except for the last episode. Due to the 90-minute time constraint, the first participant did not finish the last user story he worked on. Thus the last episode did not end with any successful unit test invocation, which prevented Zorro from inferring the development behavior in it. In Chapter 4, we introduced that Zorro classifies an episode as "Unknown" if it does not end with a successful unit test invocation. This is often the situation at the end of a programming session.

7.5.3 Cross-validation of TDD Behaviors and TDD Compliance Inference

As the author of Zorro, my observation of development behaviors could be biased (Section 7.4). This bias would decrease construct validity [78] if participant observation were the only data analysis method. To improve construct validity, I used an additional source of data – participants' validation comments that were acquired through the "TDD Episode Validation" analysis (Section 7.3.5).

Analysis Method

Similarly to the participant observation analysis in the previous step, I compared the first participant's validation comments episode by episode to Zorro's inference results for cross-validation.

Analysis Result

In Table 7.3, I listed the first participant's comments along with both Zorro's inference and my video observation results. Note that the participant's comments on development behaviors are different from what Zorro inferred and what I observed in the ESR video because participants described their development behaviors by selecting them from the following list:

- adding new functionality,
- refactoring,
- adding test,
- just running tests,
- can't tell,
- other

for each episode. Because of this, it is impossible to directly compare what the participant agreed upon to be his development behaviors to what Zorro inferred and what I observed. Thus, I will introduce a mapping schema to make them comparable in Section 7.6 after processing all participants' data.

In term of TDD compliance, the first participant believed that his development process was 100% TDD compliant, which is slightly different from what Zorro inferred but conformant to what I observed in the video. As I discusssed in Section 7.5.2, this difference is caused by Zorro's stringent requirement that an episode must end with successful unit test invocations.

Index	Zorro Inference		Video Observation		Participant Comment	
	Behavior	Is TDD?	Behavior	Is TDD?	Behavior	Is TDD?
1	test-addition	Yes	test-addition	Yes	adding test	Yes
2	refactoring	Yes	test-first	Yes	refactoring	Yes
3	refactoring	Yes	refactoring	Yes	refactoring	Yes
4	test-first	Yes	test-first	Yes	adding new functionality, adding test	Yes
5	test-first	Yes	test-first	Yes	refactoring	Yes
6	test-first	Yes	test-first	Yes	adding new functionality, adding test	Yes
7	test-first	Yes	test-first	Yes	adding new functionality, refactoring, adding test	Yes
8	test-first	Yes	test-first	Yes	adding new functionality, adding test	Yes
9	test-addition	Yes	test-addition	Yes	adding test	Yes
10	unknown	No	test-first	Yes	adding new functionality, refactoring, adding test	Yes

Table 7.3. Participants’ Comments on their Development Behaviors

7.5.4 Participant Interview Analysis using the Coding Method

I analyzed the first participant's interview data to study his opinions on unit testing and TDD using the coding analysis method.

Analysis Method

Coding is a data analysis method that can generate a description of setting or people as well as categories or themes[12, 13]. For interview conducted in this study, I coded participants into different categories according to their opinions on software development, unit testing, and TDD.

Analysis Result

Table 7.4 summarizes the interview questions and answers from the first participant.

Table 7.4. List of Interview Questions and Answers

Interview Question	Participant's Response
Unit testing experience	Several years.
Prior unit testing strategy	Write test after production iteratively.
How much unit testing	Not all the time.
TDD's impact on unit testing	TDD is messy and leads to wrong design. TDD is better if there is good design first.
Comfortableness of TDD	Hard, especially when a refactoring activity caused previous tests failed (regression test failure).
Full-scale use of TDD	Do not want to do so until I get accustomed to it. Likes the idea of TDD.

In the interview, the first participant stated that unit testing was a practice that was required in his prior software development. In his opinion, iteratively implementing test cases afterward makes more sense than writing test first. Additionally, TDD may lead to wrong design and messy code if there is no good design first. Therefore, implementing software in TDD is hard and it takes time to get accustomed to it. With these answers, I coded a category named"somewhat in favor of unit testing but not in favor of TDD".

7.5.5 Reporting Usefulness of Zorro's Analyses

With the navigation of the "Zorro Evaluation Wizard", participants conducted five Zorro's analyses on their own TDD development conducted in this study. After reviewing each analysis, they evaluated its usefulness. I generated the usefulness evaluation results in this step.

Analysis Method

The evaluation of usefulness varies from "Strongly Disagree" to "Strongly Agree", which are quantified into values from 1 to 5 (Table 7.5). In addition to the scale of usefulness, participants

Table 7.5. Table of Usefulness Scale

Strengthen	Scale
Strongly Disagree	1
Disagree	2
Neutral	3
Agree	4
Strongly Agree	5

also checked areas that were helpful after reviewing each analysis. Table 7.6 lists all possible useful areas that are encoded to UA-1, UA-2, and UA-3 etc.

Table 7.6. Table of Useful Areas

Code	Useful Area
UA-1	Acquiring awareness of my programming patterns
UA-2	Learning TDD
UA-3	Mastering TDD
UA-4	Monitoring my pace
UA-5	Improving my programming skills
UA-6	Discovering the situations in which TDD is useful
UA-7	Discovering the situations in which TDD is applicable
UA-8	Gauging how much testing I am doing
UA-9	Other

Analysis Result

Table 7.7 is a summary of the first participant's evaluation on Zorro's usefulness. Note that he only evaluated 4 of 5 analyses because the "Effort T/P Ratio" analysis had a bug at that time, which was fixed after he finished participating in this study.

Table 7.7. The First Participant's Usefulness Evaluation

Analysis Name	Scale	Areas								
		UA-1	UA-2	UA-3	UA-4	UA-5	UA-6	UA-7	UA-8	UA-9
Demography Analysis	3		X			X		X		
Effort T/P Ratio	N/A									
Size T/P Ratio	2				X					
Duration	3				X					
Duration Histogram	2				X					

The first participant thought that Zorro's analyses were somewhat useful to him. According to his evaluation, the "TDD Episode Demography" analysis was most useful for learning TDD whereas other analyses were only good at showing his development pace.

7.6 Classroom Study Data Analysis Results

In the data analysis, I assigned letters 'A', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', and 'T' to 11 participants as their identifications. Using the data analysis methods introduced in the previous section (Section 7.5), I analyzed data collected from all participants and presented results in the following.

7.6.1 An unexpected phenomenon and participant grouping

First, I will introduce an unexpected phenomenon I discovered in the data analysis. In Chapter 2, we have discussed that TDD is iterative and incremental. The rhythm of TDD is[6]:

1. Quickly add a test.
2. Run all the tests and see the new one fail.
3. Make a little change.

4. Run all tests and see them all succeed.

5. Refactor to remove duplication.

Based upon the rhythm of TDD, I designed Zorro to partition a TDD development stream over a time period into episodes using successful test invocations as tokens (Chapter 4). Also in Chapter 4, I have shown that the "Test-Pass" tokenizer is sufficient for identifying TDD iterations. However, in this study, I found an unexpected phenomenon that made the partitioning of several episodes problematic. The cause is that in the step 2 of the TDD rhythm, "run all the tests and see the new one fail" does not occur as expected when compilation errors exist. Although Eclipse, the IDE I used in this study, prompted a warning message in this case and allowed developers to cancel test invocations, some developers opted not to. As a result, sometimes test invocations succeeded unexpectedly regardless of compilation errors. In consequence, Zorro failed to partition some episodes, which led to development stream partitioning and behavioral inference errors.

To investigate how this phenomenon affected Zorro, I divided participants into groups G1 and G2 based upon their test invocation behaviors.

- **G1** Participants who canceled a test invocation when compilation error(s) existed.
- **G2** Participants who continued a test invocation regardless of the existence of compilation error(s).

Among 11 participants, 4 of them are in group G1 and 7 of them are in group G2 (see Table 7.8). Because development stream partitioning and behavioral inference errors only occurred to participants in group G2, I will term this phenomenon as G2-DevBehavior in the rest of this document. I will also discuss how Zorro can be improved if we can avoid this problem.

Table 7.8. Participant Groups

Group	Participants
G1:	K, L, O, and R
G2:	A, M, N, P, Q, S, and T

7.6.2 Validation of Data Collection

Using the participant observation research method discussed in Section 7.5.1, I validated Zorro's data collection for all participants.

Analysis Result

Table 7.9 presents numbers of development activities for each participant. The first column includes participant IDs. The second column has numbers of episodes that were partitioned by Zorro. For each participant, I listed his/her development activities per episode in columns 3 and 4. The number of activities collected by Zorro is in column 3 and the number of activities I observed is in column 4. The rest two columns have mean and median values of episode activity number differences.

Table 7.9. Summary of Development Activities

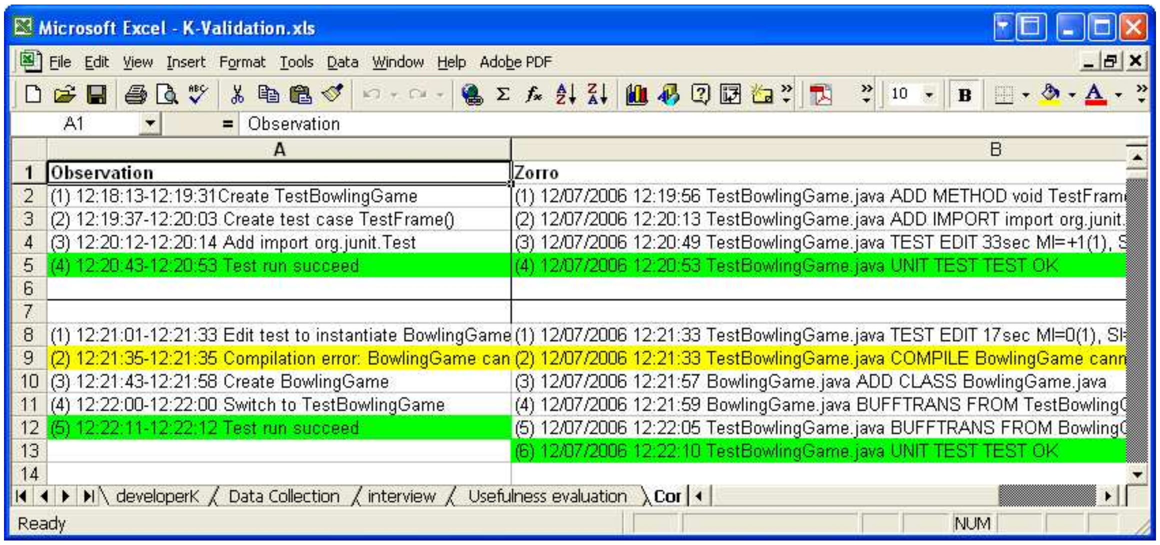
ID	Episodes	Activities per Episode		Activity Difference	
		Zorro	Video Observation	Mean	Median
A	19	12.7	11.9	0.8	0
K	10	19.8	15.3	4.5	4
L	8	32.5	30.4	2.1	2
N	9	20.4	18.2	2.2	3
O	16	15.3	13.6	1.7	0.5
P	18	13.7	11.7	2.1	1.5
Q	21	9.8	9.6	0.5	0
R	14	12.6	11.2	1.4	0
S	9	16.3	12.7	3.7	2
T	13	15.3	13.2	2.1	1
Mean	13.8	16.8	14.8	2.1	1.4

Note that I excluded data from participant 'M' in Table 7.9 that helped me find a bug in the Eclipse sensor. The bug was caused by the G2-DevBehavior I discussed in the previous section (Section 7.6.1). At the time when G2-DevBehavior occurred, a run time exception was thrown but the Eclipse sensor did not handle it gracefully such that some development activities from participant 'M' were missing. Though I improved the Eclipse sensor after finding this bug, some data from participant 'M' were permanently lost. So in the rest of this document, I will not include his data, which reduced the number of effective participants to 10.

Discussion

According to analysis results presented in Table 7.9, Zorro is capable of collecting development activities. On average, Zorro collected more development activities (16.8 per episode) than what I observed (14.8 per episode). Both the mean and median values of episode activity number differences are positive.

In addition to computing differences of development activities, I also did further investigation to find what caused them by comparing development activities collected by Zorro to development activities I observed.


Microsoft Excel - K-Validation.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
A1 = Observation
developerK
Data Collection
interview
Usefulness evaluation
Cor
Ready
NUM

	A	B
1	Observation	Zorro
2	(1) 12:18:13-12:19:31Create TestBowlingGame	(1) 12/07/2006 12:19:56 TestBowlingGame.java ADD METHOD void TestFram
3	(2) 12:19:37-12:20:03 Create test case TestFrame()	(2) 12/07/2006 12:20:13 TestBowlingGame.java ADD IMPORT import org.junit.
4	(3) 12:20:12-12:20:14 Add import org.junit.Test	(3) 12/07/2006 12:20:49 TestBowlingGame.java TEST EDIT 33sec MI=+1(1), S
5	(4) 12:20:43-12:20:53 Test run succeed	(4) 12/07/2006 12:20:53 TestBowlingGame.java UNIT TEST TEST OK
6		
7		
8	(1) 12:21:01-12:21:33 Edit test to instantiate BowlingGame	(1) 12/07/2006 12:21:33 TestBowlingGame.java TEST EDIT 17sec MI=0(1), SI
9	(2) 12:21:35-12:21:35 Compilation error: BowlingGame can	(2) 12/07/2006 12:21:33 TestBowlingGame.java COMPILE BowlingGame cann
10	(3) 12:21:43-12:21:58 Create BowlingGame	(3) 12/07/2006 12:21:57 BowlingGame.java ADD CLASS BowlingGame.java
11	(4) 12:22:00-12:22:00 Switch to TestBowlingGame	(4) 12/07/2006 12:21:59 BowlingGame.java BUFFTRANS FROM TestBowling
12	(5) 12:22:11-12:22:12 Test run succeed	(5) 12/07/2006 12:22:05 TestBowlingGame.java BUFFTRANS FROM Bowling
13		(6) 12/07/2006 12:22:10 TestBowlingGame.java UNIT TEST TEST OK
14		

Figure 7.3. Validation of Zorro's Development Activities

With side by side comparison (Figure 7.3), I found two categories of situations that were responsible for Zorro's excessive data collection in the following. First, Zorro sometimes reported two or more activities while I only observed one development activity as what I described in the item of "Kill two birds with one stone". Second, though technically ESR should capture everything that happens in the Eclipse IDE, it did not in situations that were described in items of "Invisible editing activities" and "Problems view of Eclipse could be hidden".

1. Kill two birds with one stone

When a developer changed statements that were associated with object components such as import, package declaration, attributes, method name, method return type, or method parameters, the Eclipse sensor collected two types of development activities: editing and refactoring. Similarly the Eclipse sensor would also collect both types of development activities when a developer used refactoring commands supplied in Eclipse. In both cases, Zorro doubled development activities.

2. The problems view of Eclipse was hidden

   In videos of participants ‘K’, ‘M’, and ‘T’, the problems view of Eclipse was overlapped by other views for a while (see Figure 7.4). I cannot observe compilation errors in this situation because Eclipse reports them in the problems view. For instance, in Figure 7.4, the JavaDoc view overlapped the problems view. Since the title of the problems view was highlighted, it probably meant that there should have had compilation errors, but I could not tell it by watching the ESR video.

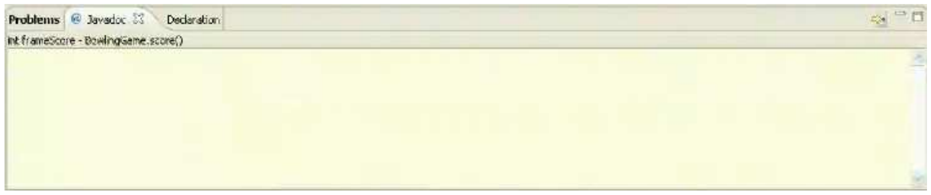


Figure 7.4. Invisible Problems View in Eclipse

3. Invisible editing activities

   A developer could input a few blank spaces while programming. Since ESR captured the screen of Eclipse, I would not be able to observe this kind of development activities because there is no visible changes.

The above three items can answer why Zorro collected more development activities than what I observed in recorded videos. However, there were also a few cases in which Zorro missed development activities (see items “Quick editing” and “Quick buffer transition”).

1. Quick editing

   The Eclipse sensor uses “state change” as the foundation to detect editing development activities. A timer thread in the sensor wakes up every 10 seconds to check the active buffer. If there are any changes made to the active buffer, the sensor will fire a "state change" event. Zorro reduces a series of consecutive "state change" events into an editing activity when processing development streams. This mechanism works well unless a developer edits a file for less than 10 seconds and then switches to another buffer. If so, the Eclipse sensor will miss an editing development activity.

2. Quick buffer transition

   The Eclipse sensor collects "buff trans" activities by checking the active buffer. A timer thread wakes up every 5 seconds to detect whether there is a buffer transition activity. Five-second is a small time period, but it is long enough for a developer to change the active buffers two or more times. If two or more consecutive buffer transition activities occur in less than 5 seconds, the Eclipse sensor might fail to capture some or all of them.

Conclusion

In this section, I summarized Zorro's data collection validation results (see Table 7.9). It turned out that Zorro collected more development activities per episode than I observed for every participant. Further investigation indicated that Zorro only missed a few development activities when participants quickly edited code or switched buffers. Also, Zorro collected development activities that were invisible in recorded videos. The analysis in this section indicates that there is partial supporting evidence for the research questions Q2a because Zorro can collect development activities more precisely than ESR, a recorder that can capture almost every development activity that occurs in the Eclipse IDE.

7.6.3 Validation of TDD Behaviors and TDD Compliance Inference

Zorro recognizes TDD development through two steps (Chapter 4): (1) inferring development behaviors by matching development activities in episodes to a set of predefined development behaviors; (2) and then deducing TDD compliance using inferred episode behaviors. Thus, in this study, the validation analysis also had two steps: episode behavioral inference validation and TDD compliance inference validation, both of which were introduced in Section 7.5.2 for an individual participant. After validating Zorro for all participants, I summarized validation results in this section.

Validation of Development Behavioral Inference

For an individual participant I created a table similar as Table 7.2 after validating Zorro's inference of his/her development behaviors. I then assigned the value 1 to an episode if the development behavior I observed in the video agrees with the development behavior Zorro inferred. In the end, I counted the number of 1's episodes and presented results in Table 7.10 in which the last column is the percentage of 1's episodes to total episodes. This percentage could be an indicator of Zorro's development behavioral inference accuracy.

Table 7.10. Video observation validation of development behaviors

ID	Episodes	1's Episodes	Percentage
A	19	15	78.9%
K	10	8	80.0%
L	8	7	87.5%
N	9	4	44.4%
O	16	15	93.8%
P	18	12	66.7%
Q	21	11	52.4%
R	14	13	92.9%
S	9	2	22.2%
T	13	9	69.2%
Total	137	96	70.1%

In total, the participant observation analysis agreed that Zorro correctly inferred development behaviors in 96 episodes resulting in 70.1% inference accuracy. But the inference accuracy is inconsistent from participant to participant. The lowest is 22.2% and the highest is 93.8% (Table 7.10).

To address this inconsistency, one thing we can do is to study the impact of G2-DevBehavior by separating Table 7.10 into two sub-tables: one for group G1 (Table 7.11), and the other one for group G2 (Table 7.12). Clearly, Zorro performed much better on inferring development behaviors for group G1 than for group G2. For G1, the average inference accuracy was 89.6%, which is much higher than the average accuracy for G2. It indicates that G2-DevBehavior, which affected 24 out of 89 episodes, had a substantial impact on the accuracy of Zorro's inference of development behaviors.

Table 7.11. Video observation validation of development behaviors for G1

ID	Episodes	1's Episodes	Percent
K	10	8	80.0%
L	8	7	87.5%
O	16	15	93.8%
R	14	13	92.9%
Total	48	43	89.6%

Table 7.12. Video observation validation of development behaviors for G2

ID	Episodes	1's Episodes	Percent	Episodes with G2-DevBehavior
A	19	15	78.9%	3
N	9	4	44.4%	3
P	18	12	66.7%	4
Q	21	11	52.4%	9
S	9	2	22.2%	2
T	13	9	69.2%	3
Total	89	53	59.6%	24

Validation of TDD Compliance Inference

Table 7.2 lists not only development behaviors I observed but also TDD compliance for an individual participant. An episode is TDD compliant if its development behavior is either a portion of a TDD iteration such as refactoring or a complete TDD iteration. After observing TDD compliance for all participants, I summarized results in Table 7.13 for validation.

In Table 7.13, the "TDD Compliant Episodes" column has numbers of TDD compliant episodes. It is divided into three sub-columns: "By Zorro", "By Video Analysis" and "Difference". With values in Table 7.13, we can compute percentages of TDD compliant episodes in the following:

$$\begin{aligned} TDD(Zorro)\% &= \frac{CompliantEpisodes(Zorro)}{TotalEpisodes} * 100 \\ &= \frac{113}{137} * 100 \\ &= 82.5, \end{aligned}$$

$$\begin{aligned} TDD(VideoAnalysis)\% &= \frac{CompliantEpisodes(VideoAnalysis)}{TotalEpisodes} * 100 \\ &= \frac{128}{137} * 100 \\ &= 93.4. \end{aligned}$$

Table 7.13. Video observation validation of TDD compliance

ID	Episodes	TDD Compliant Episodes		
		By Zorro	By Video Analysis	Difference
A	19	10	12	-2
K	10	9	10	-1
L	8	7	7	0
N	9	6	8	-2
O	16	15	16	-1
P	18	16	18	-2
Q	21	21	21	0
R	14	13	14	-1
S	9	3	9	-6
T	13	13	13	0
Total	137	113	128	-15
Mean				-1.5
Median				-1
STDEV				1.78

Using episode numbers as the measurement, Zorro is somewhat conservative on inferring TDD compliance compared to the participant observation analysis. The latter found that participants in this study complied to TDD in 93.4% of episodes, while Zorro inferred that 82.5% of episodes were TDD compliant. In Table 7.13, I computed the episode number difference for all participants. The mean value is -1.5 and the standard deviation is 1.78, both of which are very small.

However, the above evidence was not sufficient because the sample size was too small. Eleven students participated in this study and I can only analyze data collected from 10 of them. In order to generalize research conclusions, it is necessary to include more participants. But the resource was limited in this study since the two software engineering classes only had 16 students (Section 7.3).

Next, I will discuss whether and how G2-DevBehavior affected Zorro's TDD inference by separating participants into groups G1 and G2.

Discussion of G2-DevBehavior's Impact on TDD Compliance Inference

Using G2-DevBehavior we can separate Table 7.13 into two sub-tables: one for group G1 (Table 7.14) and the other one for group G2 (Table 7.15).

Table 7.14. Validation of TDD Compliance Inference for Group G1

ID	Episodes	TDD Compliant Episodes		
		By Zorro	By Video Analysis	Difference
K	10	9	10	-1
L	8	7	7	0
O	16	15	16	-1
R	14	13	14	-1
Total	49	45	48	-3
Mean				-0.75
Median				-1
STDEV				0.5

Table 7.15. Validation of TDD Compliance Inference for Group G2

ID	Episodes	TDD Compliant Episodes			
		By Zorro	By Video Analysis	Difference	G2-DevBehavior
A	19	10	12	-2	1
N	9	6	8	-2	1
P	18	16	18	-2	1
Q	21	21	21	0	0
S	9	3	9	-6	2
T	13	13	13	0	0
Total	89	69	81	-12	5
Mean				-2	
Median				-2	
STDEV				2.2	

There is noticeable difference between Table 7.14 and Table 7.15. Using participant observation analysis results as the benchmark, Zorro inferred many fewer episodes as TDD compliant for group G2 than for group G1. Zorro's inference has a wide confidence interval because of the G2-DevBehavior. The standard deviation was 0.5 for group G1 in contrast to 2.2 for group G2.

Next, I will discuss the inference errors that were unrelated to the G2-DevBehavior.

Discussion of Zorro's Inference Errors

Out of 137 total episodes, Zorro inferred that 113 were TDD compliant, whereas the participant observation analysis validated that 128 were TDD compliant (see data in Table 7.13). So the participant observation did not agree with Zorro's inference for 15 episodes. In Table 7.15, we learned that the G2-DevBehavior caused the inference error for 5 episodes. In this section, I will investigate what happened to the remaining 10 episodes.

It turned out that both Zorro's inference rules and participants' development behaviors played important roles for the inference errors. To make it simple I will term all of these errors as "Inference Error". In Table 7.16, for each participant, I included episodes that are affected by both G2-DevBehavior and Inference-Error. We can see that each participant, except for participant 'S',

Table 7.16. Zorro's TDD Compliance Inference Error

ID	Episodes	TDD Compliant Episodes				
		By Zorro	By Video Analysis	Difference	G2-DevBehavior	Inference-Error
A	19	10	12	-2	1	1
K	10	9	10	-1	0	1
L	8	7	7	0	0	0
N	9	6	8	-2	1	1
O	16	15	16	-1	0	1
P	18	16	18	-2	1	1
Q	21	21	21	0	0	0
R	14	13	14	-1	0	1
S	9	3	9	-6	2	4
T	13	13	13	0	0	0
Total	137	113	128	-15	5	10

had at most one episode affected by G2-DevBehavior and one episode affected by Inference-Error.

By carefully comparing development activities collected and behaviors inferred by Zorro (Figure 7.3), I identified several causes for Inference-Error.

1. Additional production code editing before the next TDD episode

   Data Source: 1 episode from participant 'A' and 1 episode from participant 'R'

   Description: After implementing a new feature, the TDD process typically requires a developer to refactor. Then the developer should rerun all the tests to make sure that the refactoring does not break anything. Should the developer rerun all the tests if the changes made to production code turn out to be very trivial? Strictly speaking, the answer for TDD is "Yes". Despite this, participants 'A' and 'R' refactored production code a small amount but opted not to run all tests. As a result, editing activities on production code fell into the following TDD episode. So the following episode began with production editing activity, which is inferred as a “test-last” episode. For example, the 9th episode from participant ‘A’ was inferred as “test-last” because he refactored following code

```
int first;
int second;
```

to

```
int first = 0;
int second = 0;
```

but did not rerun all tests.

2. No successful test invocations at the end

   Data Source: 1 episode from participant ‘K’
   Description: Successful test invocations are characteristic activities that mark the completion of a TDD iteration. An episode that does not end with successful test invocation is “unknown” to Zorro, and as a consequence, it is not TDD compliant. In this study, participant ‘K’ did not finish the last user story; therefore, he ended with an “unknown” episode although he was doing TDD.

3. Insufficient Inference Rules

   Data Source: 1 episode from participant ‘N’, 1 episode from participant ‘O’, and 1 episode from participant ‘P’
   Description: The design of Zorro’s TDD inference rules was rooted from the Red/Green/Refactor model in which tests should always have been created first. When test creations spread out in a TDD iteration, Zorro gets confused. For example, in the 7th episode from participant ‘N’, he added the test method “TestGameScore()” first, then implemented the functionality to compute the score of a bowling game, and then added assertion statements. So the test was created in two steps. As a result, this episode was wrongly inferred as “test-last”.

4. Unit tests were not properly structured

   Data Source: 4 episodes from participant ‘S’
   Description: Zorro recognizes the unit test class based upon the existence of test methods and assertion statements. A unit test could be wrongly recognized as production code, particularly at the beginning of a programming session when there are not any test methods or assertion

statements. In this situation, Zorro infers the development behavior as “test-last”, not “test-first”. For example, test code in the first episode from participant ‘S’ did not have any test method and assertion statement at the beginning. As a result, the first episode was inferred as “test-last” although participant ‘S’ thought that he was doing TDD. In turn, due to the chain effect, following three refactoring episodes were also inferred as TDD noncompliant.

Conclusion

In this section, using participant observation, I validated Zorro’s inference of TDD development behaviors and TDD compliance. The validation indicates that Zorro inferred development behaviors with 70.1% accuracy. Regarding TDD compliance, Zorro inferred that particpants in this study complied to TDD in 80.4% of episodes, while participant observation indicates that 93.9% of epsidoes were TDD compliant. The further discussion on G2-DevBehavior indicated that Zorro’s inference accuracy and consistency would be improved if Zorro could correctly interpret G2-DevBehavior.

In addition, to investigate causes for Zorro’s inference errors, I compared development activities collected and behaviors inferred by Zorro to what I observed in ESR videos. Interestingly, most of the inference errors were caused by participants’ nonstringent execution of TDD. Also, some of them were caused by insufficiently precise rules in Zorro.

- **The culprit of G2-DevBehavior**

  I discovered the G2-DevBehavior in this study, the development behavior to invoke tests regardless of compilation errors. When it occurs, the test invocation may succeed, which leads to episode partitioning and development behavior inference errors.

- **Research question Q2a: Does Zorro collect software development activities accurately enough for episode partition and TDD behavior inference?**

  This study provides evidence that Zorro collects a sufficient number of development activities accurately. Compared to participant observation using ESR, Zorro actually collected more development activities. A bug in the Eclipse sensor caused some data loss for one participant but I fixed it for the remaining 10 participants. The participant observation analysis validated that Zorro inferred development behaviors with 70.1% accuracy and TDD compliance with more than 80.4% accuracy. Both of these numbers would increase if Zorro could correctly interpret G2-DevBehavior. So the validation analysis in this section supports the research question Q2a.

- **Research question Q2b: Does Zorro's inference of TDD behaviors agree with analyses based upon participant observation?**

  According to participant observation, Zorro infers development behaviors correctly for 70.1% of episodes. It infers that 80.4% of episodes were TDD compliant while the particpant observation validated that 93.9% of episodes were TDD compliant. Further discussion indicates that Zorro can perform better if there were not G2-DevBehavior, and only 3 out of 128 episodes have inference errors. All these conclusions provided supporting evidence to the research question Q2b.

7.6.4 Cross-validation of Zorro using participant comments

In the previous section, we have shown that participant observation agreed with Zorro's inference on development behaviors and TDD compliance. In this section, I will analyze participants' comments to cross-validate the research conclusions we have drawn above.

In Section 7.5.3, I demonstrated how I conducted cross-validation analysis for one participant. Following the same method described in Section 7.5.3, I analyzed all participants' comments and described validation results in the following.

Cross-validation of TDD Compliance Inference

Table 7.17 is a summary of cross-validation results for all episodes produced by participants in this study. An episode could be "compliant", "noncompliant", or "don't know". Interestingly,

Table 7.17. TDD Compliance Comparison

Episodes / Method	Compliant	Noncompliant	Don't know
Zorro	110	27	
Video Analysis	128	9	
Participant Comment	111	11	15

based upon numbers in Table 7.17, participant comments on TDD compliance were much closer to

Zorro's inference results than to participant observation analysis results. Participants commented that 111 episodes were TDD compliant, which is just 1 episode different from what Zorro inferred.

There are two possible explanations to this phenomenon. One explanation is that Zorro was really good at inferring TDD compliance. The other explanation is that participants simply went along with Zorro's perspective on their behaviors due to lack of experience with TDD. After reviewing Zorro's reasoning process and inference results, participants commented on their development behaviors. Therefore, in order to improve validity of participants' comments, I explained to them that their comments would help to improve Zorro's inference and encouraged them to read aloud while commenting. Also, I used a digital voice recorder to record their verbal comments.

I listed detailed cross-validation results for all participants in Table 7.18. Zorro inferred that 110 of 137 episodes were compliant and 27 were noncompliant. In contrast, the participant observation analysis validated that 128 were compliant and only 9 were noncompliant. Finally, participants commented that 111 were compliant, 11 were noncompliant, and 15 were neither of them. Most of the time, participants agreed that Zorro inferred their TDD compliance very well. When they were not sure to the TDD compliance of an episode, they declared it as "Don't know". Fifteen episodes are in this category.

Discussion of G2-DevBehavior

I broke Table 7.18 into Table 7.19 and Table 7.20 by separating participants according to G2-DevBehavior. Participants in group G1 commented that Zorro almost perfectly inferred their TDD compliance (Table 7.19). It also showed that the participant observation analysis was not biased because these two analyses reached the same conclusion. Because of G2-DevBehavior, participants in group G2 did not completely agree with what Zorro inferred about their compliance to TDD, but the difference was not big (Table 7.20).

Cross-validation of Episode Behavior Inference

Participants used the web page illustrated in Figure 7.2 to give their feedback after reviewing their development activities and remembering what they have done in an episode. Meanwhile, in order to get as much feedback as possible, I also recorded their verbal comments.

ID	Episode	Compliant Episodes			Noncompliant Episodes			Don't know
		Zorro	Video Analysis	Participant Comment	Zorro	Video Analysis	Participant Comment	Participant Comment
A	19	10	12	9	9	7	7	3
K	10	9	10	9	1	0	0	1
L	8	7	7	7	1	1	1	0
N	9	6	8	7	3	1	0	2
O	16	15	16	16	1	0	0	0
P	18	16	18	18	2	0	0	0
Q	21	19	21	17	2	0	0	4
R	14	13	14	12	1	0	0	2
S	9	3	9	7	6	0	2	0
T	13	12	13	9	1	0	1	3
Total	137	110	128	111	27	9	11	15

Table 7.18. Participant Comments on TDD Compliance

Table 7.19. Group G1's comments on TDD Compliance

ID	Episode	Compliant			Noncompliant		
		Zorro	Video Analysis	Participant	Zorro	Video Analysis	Participant
K	10	9	10	9	1	0	0
L	8	7	7	7	1	1	1
O	16	15	16	16	1	0	0
R	14	13	14	12	1	0	0
Total	48	44	47	44	4	1	1

Table 7.20. Group G2's comments on TDD Compliance

ID	Episode	Compliant			Noncompliant		
		Zorro	Video Analysis	Participant	Zorro	Video Analysis	Participant
A	19	10	12	9	9	7	7
N	9	6	8	7	3	1	0
P	18	16	18	18	2	0	0
Q	21	19	21	17	2	0	0
S	9	3	9	7	6	0	2
T	13	12	13	9	1	0	1
Total	89	66	81	67	23	8	10

Participants commented on their episode behaviors by selecting from a set of development behaviors including "adding new functionality", "adding test", and "just running tests" etc. Therefore, in order to compare their comments with episode behaviors inferred by Zorro and that I observed in recorded videos, it is necessary to code their comments and build a mapping schema to connect them. Table 7.21 defines a mapping schema I used in this data analysis. In the participant comments

Table 7.21. Mapping schema from participant's comment to Zorro/Video Analysis inference

Participant Comments	Relation	Zorro/Video Analysis
adding new functionality + adding test + [more] + TDD	$\equiv$	test-first
adding new functionality + adding test + [more] + $\sim$TDD	$\equiv$	test-last
adding new functionality + [more] + TDD	$\equiv$	test-first
adding new functionality + [more] + $\sim$TDD	$\equiv$	production
adding test + [just running tests]	$\equiv$	test-addition
adding test + [just running tests] + TDD	$\equiv$	test-first
refactoring + [just running tests]	$\equiv$	refactoring
just running tests	$\equiv$	regression
no comment	?	anything
other	$\neq$	anything

column, '+' means combination of development behaviors, a development behavior is optional if it is embraced by a pair of closed brackets, '$\sim$' sign stands for negation, and "[more]" matches any development behavior. In the relation column, '≡' stands for the equivalent relationship, '≠' stands for the nonequivalent relationship, and the question mark '?' stands for the unsure relationship.

Using this mapping schema, I compared development behaviors indicated by participant comments to development behaviors inferred by Zorro. I marked an episode as "agreed" if I can map the participant's comment to the development behavior inferred by Zorro, "disagreed" if the two are not equivalent using the schema in Table 7.21, or "unsure" if the participant did not comment or no mapping exists. Comparison results are available in Table 7.22. In addition, I also appended comparison results between participant observation and Zorro in the same table. This table makes

Table 7.22. Participant's Validation of Episode Behaviors

ID	Episode	Participant v.s. Zorro			Video Analysis v.s. Zorro	
		Agreed	Disagreed	Unsure	Agreed	Disagreed
A	19	12	5	2	15	4
K	10	8	2	0	8	2
L	8	6	2	0	7	1
N	9	6	2	1	4	5
O	16	14	2	0	15	1
P	18	14	4	0	11	7
Q	21	12	5	4	11	10
R	14	12	1	1	13	1
S	9	6	3	0	2	7
T	13	10	2	1	9	4
Total	137	100	28	9	95	42

it clear that the participant comment analysis yielded very close findings to the participant observation analysis. With regard to agreed episodes, one is 100 and the other is 95. Numbers of disagreed episodes are also very close. So, on the episode level, participant comments agree with participant observation results. This provides strong evidence that the video analysis method is not biased in this study.

Regarding G2-DevBehavior, participants in group G2 were more likely to disagree with Zorro's inference of their development behaviors. For example, developer 'N' noticed that "one episode goes into two because I ran the test before adding [the production code]." Developer 'P' commented that "[when there is compilation error], JUnit does tell test failures if the method is not found. It is a JUnit problem not Eclipse." Participant 'Q' also pointed out that one of his episodes was not partitioned correctly. Participant 'T' even brought up the G2-DevBehavior problem before validating Zorro.

Conclusion

In this section, I analyzed participant comments to cross-validate Zorro's inference on TDD compliance and development behaviors.

With regard to TDD compliance, participants agreed with Zorro's inference results and the participant observation analysis results (see Table 7.17). Thus, the cross-validation provides evidence that participant observation is a viable method for validating Zorro. By separating participants into groups G1 and G2, we found that participants almost completely agreed with Zorro's inference results and participant observation analysis results for participants in group G1.

Though participant comments provided strong evidence to research question Q2c, we must be cautious because the conclusion could be influenced by a caveat problem – the participant's knowledge of Zorro's inference. I advised participants use Zorro's inference results as reference only and encourage them to read aloud to give us feedback for Zorro improvement. The cross-validation analysis in this section provides some evidence that participants were able to comment independently. They commented 15 episodes as "don't know" to express their disagreements with Zorro's inference.

With regard to episode behaviors, I built a development behavior mapping schema to map participant comments to development behaviors Zorro inferred and to those that I observed in recorded videos. According to Table 7.22, participants agreed that Zorro inferred development behaviors in 73.0% (100 of 137) of episodes correctly. In comparison, the participant observation analysis validated that Zorro inferred development behaviors in 69.3% (95 of 137) of episodes correctly. Both validation analyses concluded that Zorro is capable of inferring development behaviors, but there is still room for improvements.

7.6.5 Participant interview analysis

Before this study, the instructor of the software engineering classes introduced TDD to students after they finished the semester-long course projects. Unit testing was a required practice in their course projects. The students also practiced what they had learned in the TDD lecture using the Roman numeral conversion problem (see Appendix C).

Following the interview guideline in Appendix F, I interviewed participants on unit testing and TDD at the end of this study. Then I processed the interview script using the coding research method [12, 13] to categorize the participants.

Unit Testing Survey

Among the participants, only one has used unit testing for a long time at work, one had practiced unit testing for a while, and the others had just started to learn unit testing since the beginning of the software engineering classes. Since most participants were new to unit testing, I decided not to differentiate them based on unit testing experience in the coding process.

The participants unanimously agreed that unit testing is good for software development, but they had a diversity of opinions on how helpful unit testing is. However, some of them pointed that unit testing was hard sometimes, especially when they wanted to achieve 100% test coverage.

In the open coding stage, I wrote down the values of properties including unit testing assessment, experience, testing behavior, testing effort and test coverage. I categorized participants based on these properties. Using coding and memoing methods, I categorized the participants and listed the core categories in Table 7.23 based upon the property values.

Table 7.23. Participant Categories on Unit Testing

Category	Description	Participant
Good-but-Small-Effort	Test is good for quality but I do not write test often and I have excuses.	A, L, O, P, R, S, and T
Must-and-Test-Last	All code must have unit test. The work flow is design, think, code and test.	K
VeryGood-and-Much-Effort	Test is very good and I spent quite some effort on it	N and Q

The majority of the students acknowledged that unit testing is good for improving software quality but they did not devote enough effort on it.

TDD Survey

TDD is very different from unit testing. It not only reverses the order of production and test developments, but also advocates unit tests as the driving force for software design. I coded the interview data using properties including TDD's impacts on unit testing and software quality, and how hard it is to develop software in TDD. Table 7.24 lists the core categories.

Table 7.24. Participant Categories on Perception of TDD

Category	Description	Participant
Negative	Messy design. Straight TDD is weird.	K and S
No-Change	No guarantee for quality. May take longer such that yields better quality.	A and L
Positive-With-Condition	More time on testing and better testing. Hard if there is no good to-do list. Helpful when from scratch up. Eclipse discourages TDD because of compilation error warning.	N, O, P, Q, R, and T

A common misunderstanding of the participants is that developers should create the complete To-Do list prior to implementation, which is not true for TDD. Instead, developers should dynamically maintain the To-Do list by themselves. It is reasonable that they had this impression given the experiment settings of this study. I provided the user stories to help participants develop solutions for the bowling game problem (Appendix E) without spending a lot of time to ensure that they understood the bowling game scoring methods. This is a trade-off we had to take because previous studies found that it could take participants up to 48 hours to accomplish the same programming problem [19]. Based on lessons learned from others, providing the user stories that are easy to understand is necessary for this study's experimental design, which required a reasonable number of TDD iterations to be completed within 90 minutes. Given that my research interest is on validating Zorro's automated inference, providing user stories is acceptable. In the interview, I explained to participants that they should dynamically maintain the To-Do list by themselves in real situations.

TDD Acceptance Survey

Last, I asked participants how they would respond if their project managers required everybody to use TDD. Two developers would be against the use of TDD in daily software development. Three developers would accept it if good design documentation and To-Do lists are in place. The remaining five participants would be very willing to use TDD. Table 7.25 listed the participant categories.

Table 7.25. Participant Categories on Acceptance of TDD

Category	Description	Participant
Against	Don't know why TDD benefits. Nobody can tell whether I am doing it.	S and A
Ok	Will do. Good design and To-Do list are necessary. Need time to get used to it. May get stressed out.	K, P, and T
Welcome	Like it. Don't mind.	L, N, O, Q, and R

7.6.6 Usefulness analysis

In the following, I will report participant's survey results on Zorro's TDD analysis usefulness, and then summarize the useful areas.

Survey of Usefulness

Table 7.26 is a pivot table of the TDD analysis usefulness survey. Overall, the usefulness scores

Table 7.26. Survey on TDD Analysis Usefulness

Participant / Analysis	A	K	L	N	O	P	Q	R	S	T
Episode Demography	3	4	5	4	4	5	4	4	4	4
T/P Effort Ratio	4		5	4	4	5	3	4	4	4
T/P Size Ratio	4	3	5	4	4	5	4	3	4	4
Episode Duration	4	4	5	5	4	5	4	3	4	3
Duration Distribution	4	3	5	3	4	4	4	4	3	3

to Zorro's TDD analyses ranged from 3 to 5 according to Table 7.26. The "Duration Distribution" is the least useful analysis. One reasonable explanation for this could be that the episode numbers were too small to be used for distribution analysis.

Useful Areas

The Appendix G has the participants' selections of areas that Zorro's TDD analyses could be used for. Table 7.27 is a summary of their selections. The values in Table 7.27 are the numbers of useful areas.

Table 7.27. Summary of Useful Areas of TDD analyses

Participant / Analysis	A	K	L	N	O	P	Q	R	S	T	Total
Episode Demography	1	3	3	4	3	6	4	3	2	6	35
T/P Effort Ratio	3		2	4	4	5	3	2	4	6	33
T/P Size Ratio	4	1	2	2	2	6	1	2	3	3	26
Episode Duration	3	1	3	1	3	3	2	2	3		21
Duration Distribution	3	1	4	1	4	2	2	2	2	6	27

In order to compare number of areas on both analysis wise and participant wise, I plotted the 3-D Chart (Figure 7.5) with the data in Table 7.27. With Figure 7.5, we can easily compare the differences on usefulness among these 5 analyses. The "Episode Demography" and "T/P Effort Ratio" are the two most useful analyses. Although participants gave the least usefulness score to the "Duration Distribution" analysis, it has the potential to be useful basing upon its useful areas

Conclusion

The analysis in this section provided supporting evidence to the research question Q2d. Participants generally agreed that the Zorro is useful and it can be used to understand and improve their TDD practices.

7.7 Chapter Summary

In this chapter, I validated Zorro's sensor data collection (sections 7.6.2) and TDD behavior inference (7.6.3). Based upon the participant observation analysis, these results provide supporting evidence to research questions Q2a and Q2b (See Chapter 5).

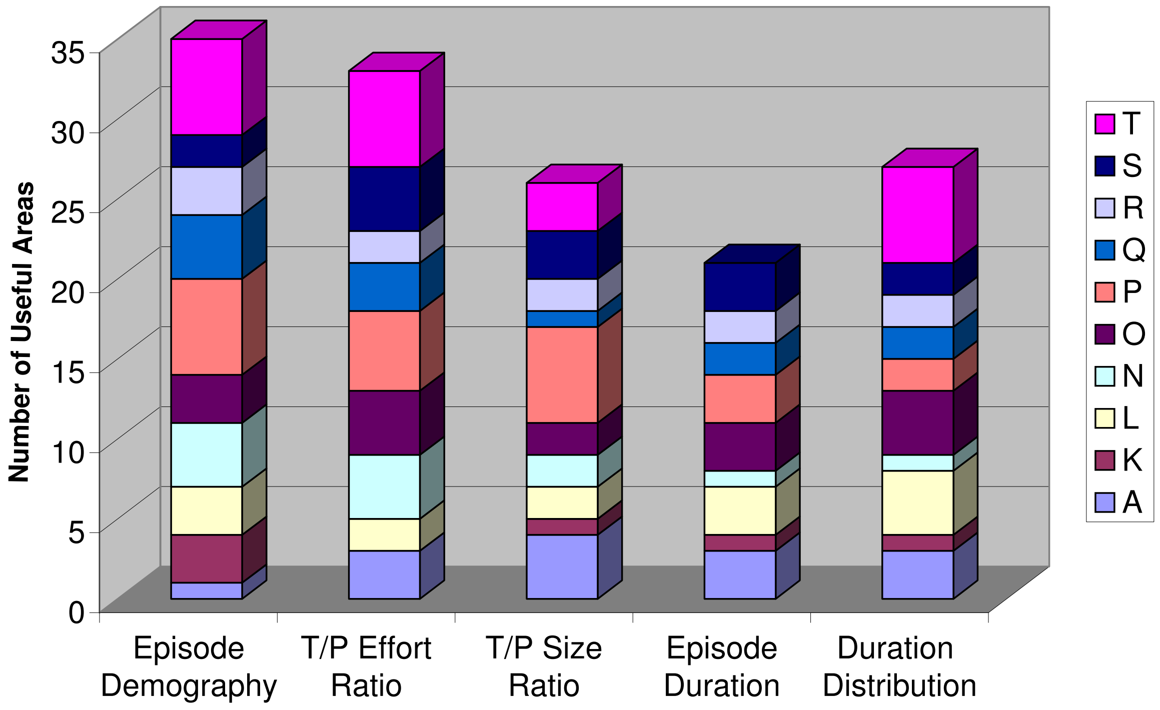


Figure 7.5. Useful Areas of TDD Analyses

In this study, I used participant comments to cross-validate Zorro's inference of development behaviors and TDD compliance. The analysis of their comments (Section 7.6.4) provided supporting evidence to the research question Q2c. Participants cross-validated that Zorro inferred TDD development behaviors accurately and that participant observation is a reasonable research method for validating Zorro.

As part of this study, I interviewed participants to survey their opinions on unit testing and TDD. The survey found that participants had a variety of opinions on how useful unit testing and TDD are (Section 7.6.5). Most of them believed that unit testing and TDD can help to improve software quality.

Participants also evaluated the usefulness of five TDD analyses provided by Zorro. The usefulness evaluation analysis (Section 7.6.6) found supporting evidence to the research question Q2d (See Chapter 5).

In addition, I discovered an unexpected phenomenon (Section 7.6.1) in the data analysis. Some participants sometimes forced Eclipse, the IDE I used in this study, to continue test invocations regardless of compilation errors. This behavior impacted Zorro's development stream partitioning and development behavioral inference. In this chapter's data analysis, I termed this development behavior as G2-DevBehavior, and divided participants into groups G1 and G2 for comparison. The validation analyses showed that Zorro's inference accuracy and consistency would improve significantly if Zorro could correctly interpret G2-DevBehavior. Thus it is necessary to let Zorro to allow this development behaviors in the future.

Chapter 8

Industrial Case Study

The pilot study and the classroom case study show that Zorro can infer TDD when it occurs in the educational and controlled environments. After these two studies, I turned my focus into evaluating Zorro's usefulness for researchers. In order to achieve this goal, I collaborated with Dr. Geir Hanssen, a researcher from SINTEF ICT of Norway on an industrial case study in which Zorro was adopted. My role in this study was the technique supporter of Zorro. I assisted Dr. Hanssen in deploying Zorro to the site and helped him to analyze the collected data. In the end, I interviewed him with regard to Zorro's usefulness.

8.1 Purpose of the Study

The purpose was to test Zorro's usefulness for researchers regardless of the accuracy of Zorro's data collection and correctness of Zorro's inference of TDD development behaviors.

8.2 Research Questions

The specific research question of this study is **Q3a: How can researchers use Zorro to assist empirical evaluation of TDD?** It supports the overall research question Q2: *"How useful is Zorro?"*

8.3 Case Study Journal

In Fall 2006, Dr. Hanssen planned to conducted an evaluation study of TDD against an existing Test-Last practice in an European software company named Foosball LLC [1]. According to Dr. Hanssen, a barrier facing this study was that harvesting development data was very hard. The Foosball LLC agreed to participate in this study but did not want to engage into any research activities that do not add much direct value. So Dr. Hanssen reached us for the possibility to use Zorro in his study, which leads to this collaborative industrial case study. As a facilitator, I provided the technique support of Zorro, assisted the project manager managing sensor installation, and analyzed the development process using the data Zorro collects. To best describe this research, I will use a journal to highlight the collaborative research activities I conducted.

8.3.1 The Prelude of this study

After implementing Zorro, I revamped its data collection and inference rules according to research results of the pilot study (Chapter 6). In Summer 2006, I worked as an intern at the NRC-IIT [2] where I collaborated with Dr. Erdogmus [3] on classification of TDD development behaviors and inference of TDD compliance. In Fall 2006, under directions from Professor Philip Johnson — my thesis advisor, and Dr. Erdogmus, I implemented the “Zorro Demo” to illustrate Zorro’s capabilities for the purpose of recruiting participants.

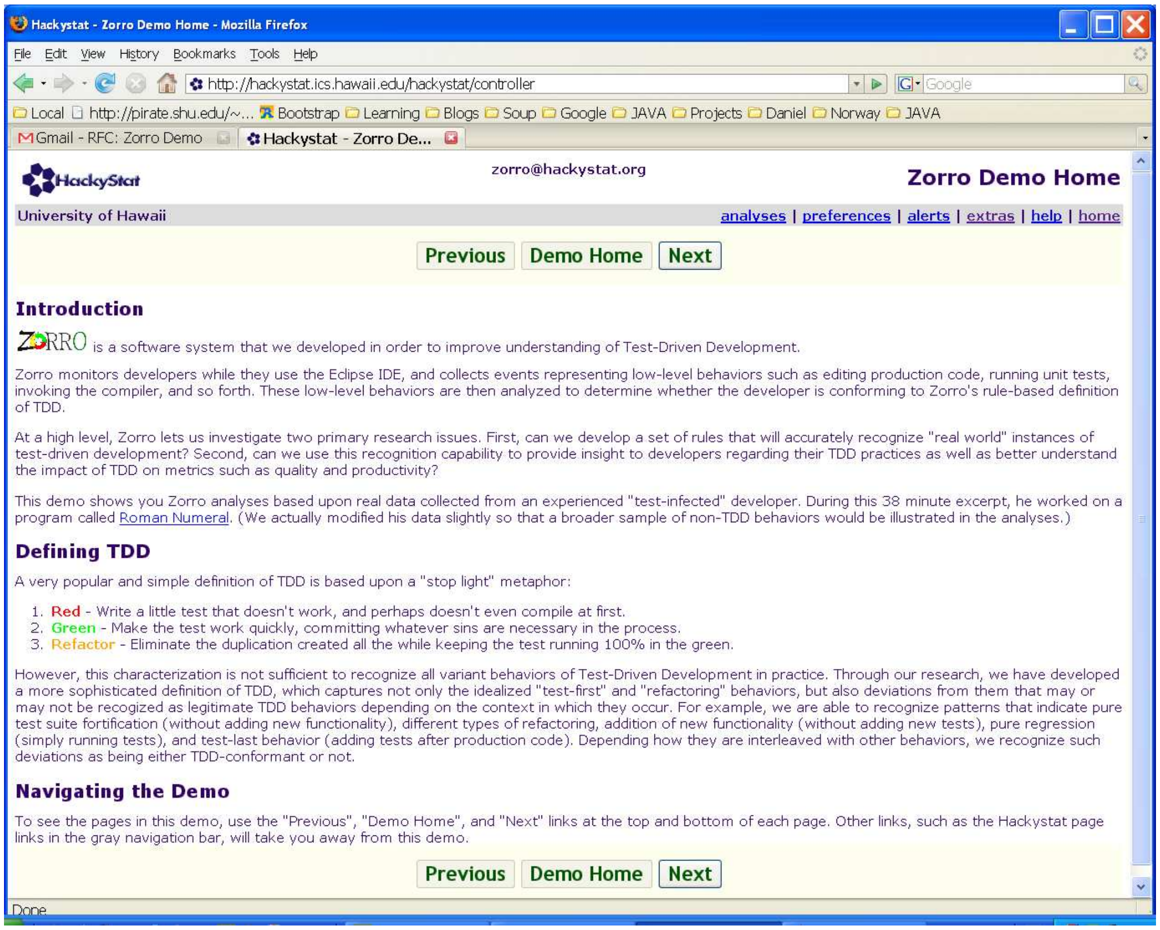
2006-10-30: Zorro Demo

I applied some TDD analyses Zorro provides to a real TDD programming session and created the “Zorro Demo”. The demo was implemented in web pages including a start page, six analysis pages and a feedback page. Figure 8.1 illustrates the start page that has a brief introduction to Zorro, how it works based upon the “stop light” metaphor of TDD, and the navigation buttons “Previous”, “Demo Home” and “Next”.

[1]The company is real but its name is faked.

[2]NRC-IIT stands for Institute for Information Technology of National Research Council Canada.

[3]Dr. Hakan Erdogmus is a Senior Research Officer in the NRC-IIT Software Engineering Group (http://iit-iti.nrc-cnrc.gc.ca/personnel/erdogmus_hakan_e.html). He has interests in Software Economics, Agile Software Development, and Software Process Measurement and Awareness.


Hackystat - Zorro Demo Home - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
Gmail - RFC: Zorro Demo
Hackystat - Zorro De...
HackyStat
zorro@hackystat.org
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
Done

Zorro Demo Home

Introduction

ZORRO is a software system that we developed in order to improve understanding of Test-Driven Development.

Zorro monitors developers while they use the Eclipse IDE, and collects events representing low-level behaviors such as editing production code, running unit tests, invoking the compiler, and so forth. These low-level behaviors are then analyzed to determine whether the developer is conforming to Zorro's rule-based definition of TDD.

At a high level, Zorro lets us investigate two primary research issues. First, can we develop a set of rules that will accurately recognize "real world" instances of test-driven development? Second, can we use this recognition capability to provide insight to developers regarding their TDD practices as well as better understand the impact of TDD on metrics such as quality and productivity?

This demo shows you Zorro analyses based upon real data collected from an experienced "test-infected" developer. During this 38 minute excerpt, he worked on a program called Roman Numeral. (We actually modified his data slightly so that a broader sample of non-TDD behaviors would be illustrated in the analyses.)

Defining TDD

A very popular and simple definition of TDD is based upon a "stop light" metaphor:

1. Red - Write a little test that doesn't work, and perhaps doesn't even compile at first.
2. Green - Make the test work quickly, committing whatever sins are necessary in the process.
3. Refactor - Eliminate the duplication created all the while keeping the test running 100% in the green.

However, this characterization is not sufficient to recognize all variant behaviors of Test-Driven Development in practice. Through our research, we have developed a more sophisticated definition of TDD, which captures not only the idealized "test-first" and "refactoring" behaviors, but also deviations from them that may or may not be recogized as legitimate TDD behaviors depending on the context in which they occur. For example, we are able to recognize patterns that indicate pure test suite fortification (without adding new functionality), different types of refactoring, addition of new functionality (without adding new tests), pure regression (simply running tests), and test-last behavior (adding tests after production code). Depending how they are interleaved with other behaviors, we recognize such deviations as being either TDD-conformant or not.

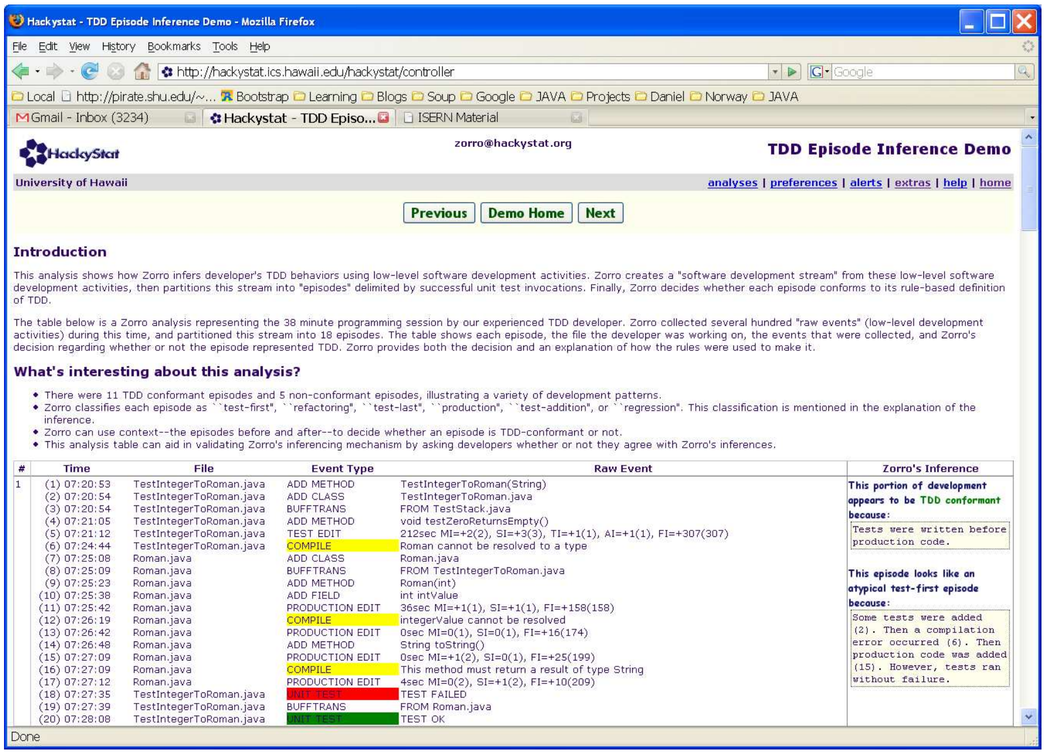
Navigating the Demo

To see the pages in this demo, use the "Previous", "Demo Home", and "Next" links at the top and bottom of each page. Other links, such as the Hackystat page links in the gray navigation bar, will take you away from this demo.

Figure 8.1. Zorro Demo Wizard

1. **TDD Episode Inference**

   Clicking the "Next" button in the start page will lead users to the first analysis, "TDD Episode Inference" (Figure 8.2). It provides a view for end users to understand how Zorro infers compliance to TDD using collected development activities. With this analysis, beginners can learn how to program in TDD; experienced practitioners can validate the inference rules and improve their compliance to TDD; and researchers can improve validity of their empirical studies by knowing participants' compliance to TDD.


Hackystat - TDD Episode Inference Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
Gmail - Inbox (3234) Hackystat - TDD Episo... ISERN Material
HackyStat
zorro@hackystat.org
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous Demo Home Next
Done

TDD Episode Inference Demo

Introduction

This analysis shows how Zorro infers developer's TDD behaviors using low-level software development activities. Zorro creates a "software development stream" from these low-level software development activities, then partitions this stream into "episodes" delimited by successful unit test invocations. Finally, Zorro decides whether each episode conforms to its rule-based definition of TDD.

The table below is a Zorro analysis representing the 38 minute programming session by our experienced TDD developer. Zorro collected several hundred "raw events" (low-level development activities) during this time, and partitioned this stream into 18 episodes. The table shows each episode, the file the developer was working on, the events that were collected, and Zorro's decision regarding whether or not the episode represented TDD. Zorro provides both the decision and an explanation of how the rules were used to make it.

What's interesting about this analysis?

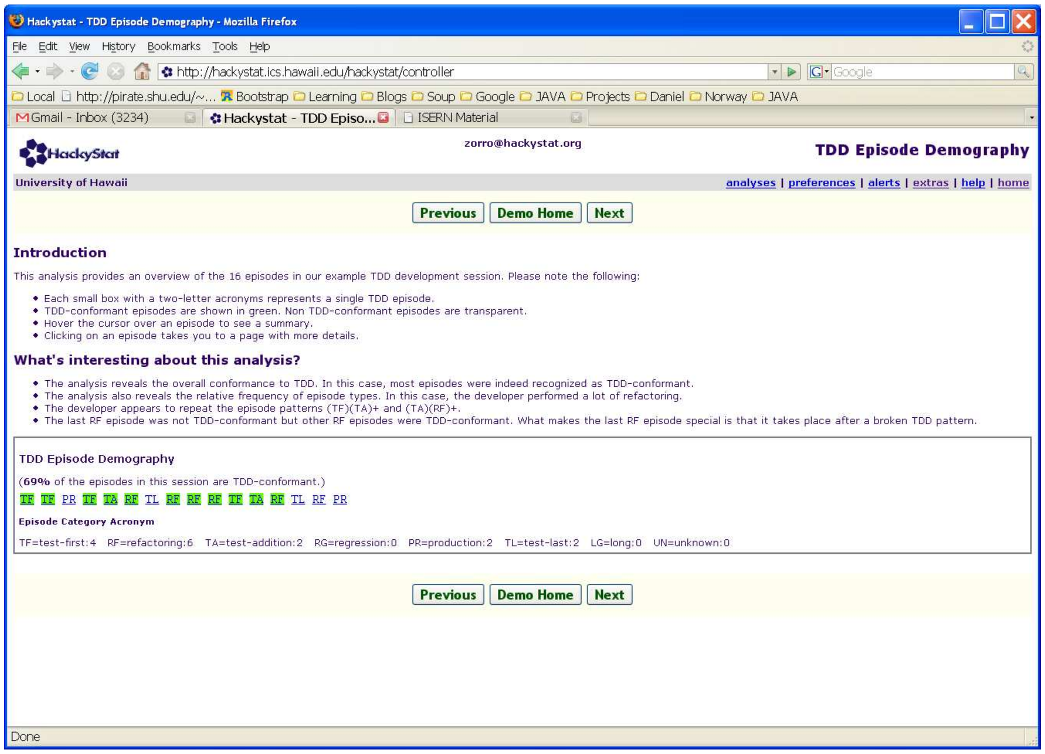
- There were 11 TDD conformant episodes and 5 non-conformant episodes, illustrating a variety of development patterns.
- Zorro classifies each episode as ``test-first", ``refactoring", ``test-last", ``production", ``test-addition", or ``regression". This classification is mentioned in the explanation of the inference.
- Zorro can use context--the episodes before and after--to decide whether an episode is TDD-conformant or not.
- This analysis table can aid in validating Zorro's inferencing mechanism by asking developers whether or not they agree with Zorro's inferences.

#	Time	File	Event Type	Raw Event	Zorro's Inference
1	(1) 07:20:53	TestIntegerToRoman.java	ADD METHOD	TestIntegerToRoman(String)	This portion of development appears to be TDD conformant because: Tests were written before production code. This episode looks like an atypical test-first episode because: Some tests were added (2). Then a compilation error occurred (6). Then production code was added (15). However, tests ran without failure.
	(2) 07:20:54	TestIntegerToRoman.java	ADD CLASS	TestIntegerToRoman.java	
	(3) 07:20:54	TestIntegerToRoman.java	BUFFTRANS	FROM TestStack.java	
	(4) 07:21:05	TestIntegerToRoman.java	ADD METHOD	void testZeroReturnsEmpty()	
	(5) 07:21:12	TestIntegerToRoman.java	TEST EDIT	212sec MI=+2(2), SI=+3(3), TI=+1(1), AI=+1(1), FI=+307(307)	
	(6) 07:24:44	TestIntegerToRoman.java	COMPILE	Roman cannot be resolved to a type	
	(7) 07:25:08	Roman.java	ADD CLASS	Roman.java	
	(8) 07:25:09	Roman.java	BUFFTRANS	FROM TestIntegerToRoman.java	
	(9) 07:25:23	Roman.java	ADD METHOD	Roman(int)	
	(10) 07:25:38	Roman.java	ADD FIELD	int intValue	
	(11) 07:25:42	Roman.java	PRODUCTION EDIT	36sec MI=+1(1), SI=+1(1), FI=+158(158)	
	(12) 07:26:19	Roman.java	COMPILE	integerValue cannot be resolved	
	(13) 07:26:42	Roman.java	PRODUCTION EDIT	0sec MI=0(1), SI=0(1), FI=+16(174)	
	(14) 07:26:48	Roman.java	ADD METHOD	String toString()	
	(15) 07:27:09	Roman.java	PRODUCTION EDIT	0sec MI=+1(2), SI=0(1), FI=+25(199)	
	(16) 07:27:09	Roman.java	COMPILE	This method must return a result of type String	
	(17) 07:27:12	Roman.java	PRODUCTION EDIT	4sec MI=0(2), SI=+1(2), FI=+10(209)	
	(18) 07:27:35	TestIntegerToRoman.java	UNIT TEST	TEST FAILED	
	(19) 07:27:39	TestIntegerToRoman.java	BUFFTRANS	FROM Roman.java	
	(20) 07:28:08	TestIntegerToRoman.java	UNIT TEST	TEST OK	

Figure 8.2. TDD Episode Inference Demo

2. **TDD Episode Demography**

   Following “TDD Episode Inference” is “TDD Episode Demography” that is an overview of a programming session (Figure 8.3). It lines up all the episodes partitioned from the development stream, each of which is a box with a two-letter acronym representing the episode development behavior. With this analysis, users can inspect TDD programming sessions and look for TDD development patterns for improvement.



Hackystat - TDD Episode Demography - Mozilla Firefox

http://hackystat.ics.hawaii.edu/hackystat/controller

zorro@hackystat.org

TDD Episode Demography

University of Hawaii

Introduction

This analysis provides an overview of the 16 episodes in our example TDD development session. Please note the following:

- Each small box with a two-letter acronyms represents a single TDD episode.
- TDD-conformant episodes are shown in green. Non TDD-conformant episodes are transparent.
- Hover the cursor over an episode to see a summary.
- Clicking on an episode takes you to a page with more details.

What's interesting about this analysis?

- The analysis reveals the overall conformance to TDD. In this case, most episodes were indeed recognized as TDD-conformant.
- The analysis also reveals the relative frequency of episode types. In this case, the developer performed a lot of refactoring.
- The developer appears to repeat the episode patterns (TF)(TA)+ and (TA)(RF)+.
- The last RF episode was not TDD-conformant but other RF episodes were TDD-conformant. What makes the last RF episode special is that it takes place after a broken TDD pattern.

TDD Episode Demography

(**69%** of the episodes in this session are TDD-conformant.)

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

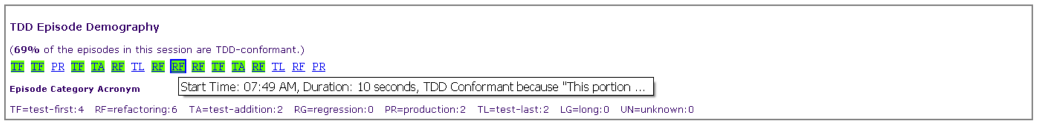
Episode Category Acronym

TF=test-first:4 RF=refactoring:6 TA=test-addition:2 RG=regression:0 PR=production:2 TL=test-last:2 LG=long:0 UN=unknown:0

File Edit View History Bookmarks Tools Help
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
Gmail - Inbox (3234)
Hackystat - TDD Episo...
ISERN Material
HackyStat
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
Previous
Demo Home
Next
Done

Figure 8.3. TDD Episode Demography

In addition to showing overall development behaviors and compliance to TDD, this analysis can interact with the “TDD Episode Inference” analysis. When users move mouse over a box in this analysis, a tooltip of episode synopsis will appear on the screen next to the mouse cursor (Figure 8.4), and border of the box will change to blue which indicates that it is clickable. Clicking a box will direct users to “TDD Episode Inference” (See Figure 8.5).



TDD Episode Demography

(**69%** of the episodes in this session are TDD-conformant.)

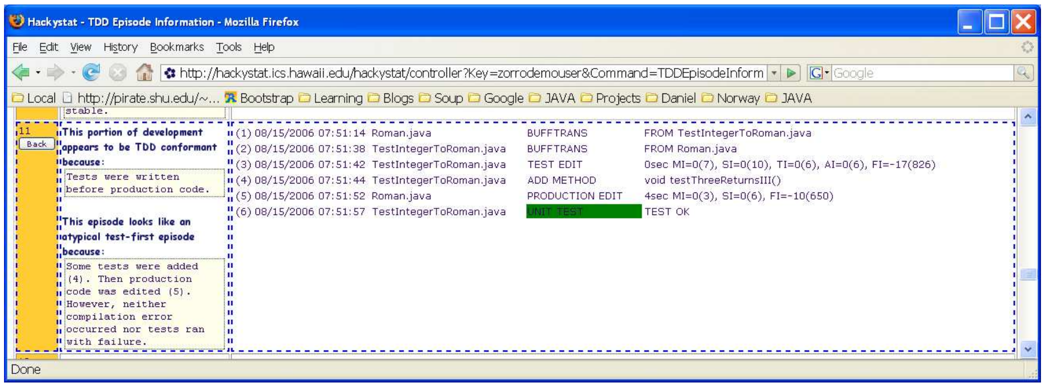
TF TF PR TF TA RF TL RF RF RF TF TA RF TL RF PR

Episode Category Acronym

Start Time: 07:49 AM, Duration: 10 seconds, TDD Conformant because "This portion ...

TF=test-first:4 RF=refactoring:6 TA=test-addition:2 RG=regression:0 PR=production:2 TL=test-last:2 LG=long:0 UN=unknown:0

Figure 8.4. Tooltip of Episode Synopsis


Hackystat - TDD Episode Information - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller?Key=zorrodemouser&Command=TDDEpisodeInform
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
Back
Done

stable.

11

This portion of development appears to be TDD conformant because:

```
Tests were written
before production code.
```

This episode looks like an atypical test-first episode because:

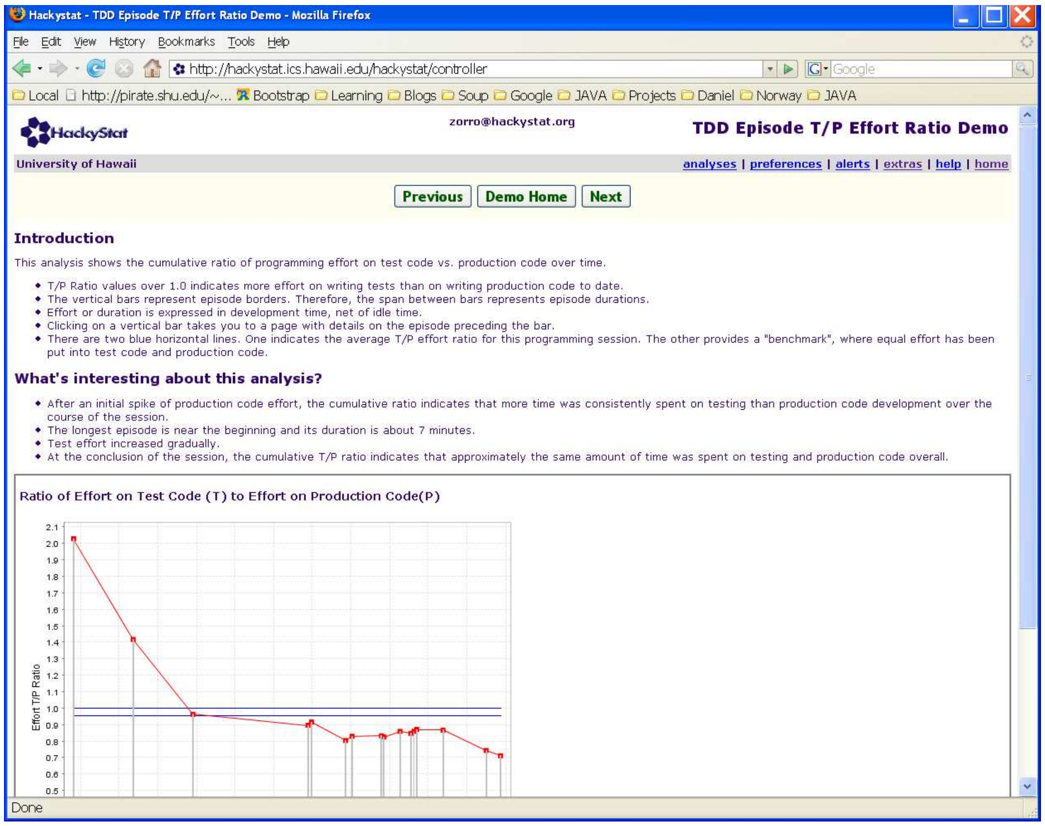
```
Some tests were added
(4). Then production
code was edited (5).
However, neither
compilation error
occurred nor tests ran
with failure.
```

```
(1) 08/15/2006 07:51:14 Roman.java                BUFFTRANS         FROM TestIntegerToRoman.java
(2) 08/15/2006 07:51:38 TestIntegerToRoman.java   BUFFTRANS         FROM Roman.java
(3) 08/15/2006 07:51:42 TestIntegerToRoman.java   TEST EDIT         0sec MI=0(7), SI=0(10), TI=0(6), AI=0(6), FI=-17(826)
(4) 08/15/2006 07:51:44 TestIntegerToRoman.java   ADD METHOD        void testThreeReturnsIII()
(5) 08/15/2006 07:51:52 Roman.java                PRODUCTION EDIT   4sec MI=0(3), SI=0(6), FI=-10(650)
(6) 08/15/2006 07:51:57 TestIntegerToRoman.java   UNIT TEST         TEST OK
```

Figure 8.5. Episode Details with Back Button

3. **TDD Episode T/P Ratio of Development Time**

   The third analysis is "TDD Episode T/P Ratio of Development Time" (Figure 8.6). This ratio is a good indicator of TDD compliance according to [74]. The drop of T/P ratio could be a warning signal indicating that developers might slip away from TDD. If users are not confident in Zorro's automated inference of TDD compliance, they can inspect their unit testing development activities using this analysis. For others who do not practice TDD but unit testing, this analysis can provide their dynamic unit testing behaviors.


Hackystat - TDD Episode T/P Effort Ratio Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
HackyStat
zorro@hackystat.org
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
Done

TDD Episode T/P Effort Ratio Demo

Introduction

This analysis shows the cumulative ratio of programming effort on test code vs. production code over time.

- T/P Ratio values over 1.0 indicates more effort on writing tests than on writing production code to date.
- The vertical bars represent episode borders. Therefore, the span between bars represents episode durations.
- Effort or duration is expressed in development time, net of idle time.
- Clicking on a vertical bar takes you to a page with details on the episode preceding the bar.
- There are two blue horizontal lines. One indicates the average T/P effort ratio for this programming session. The other provides a "benchmark", where equal effort has been put into test code and production code.

What's interesting about this analysis?

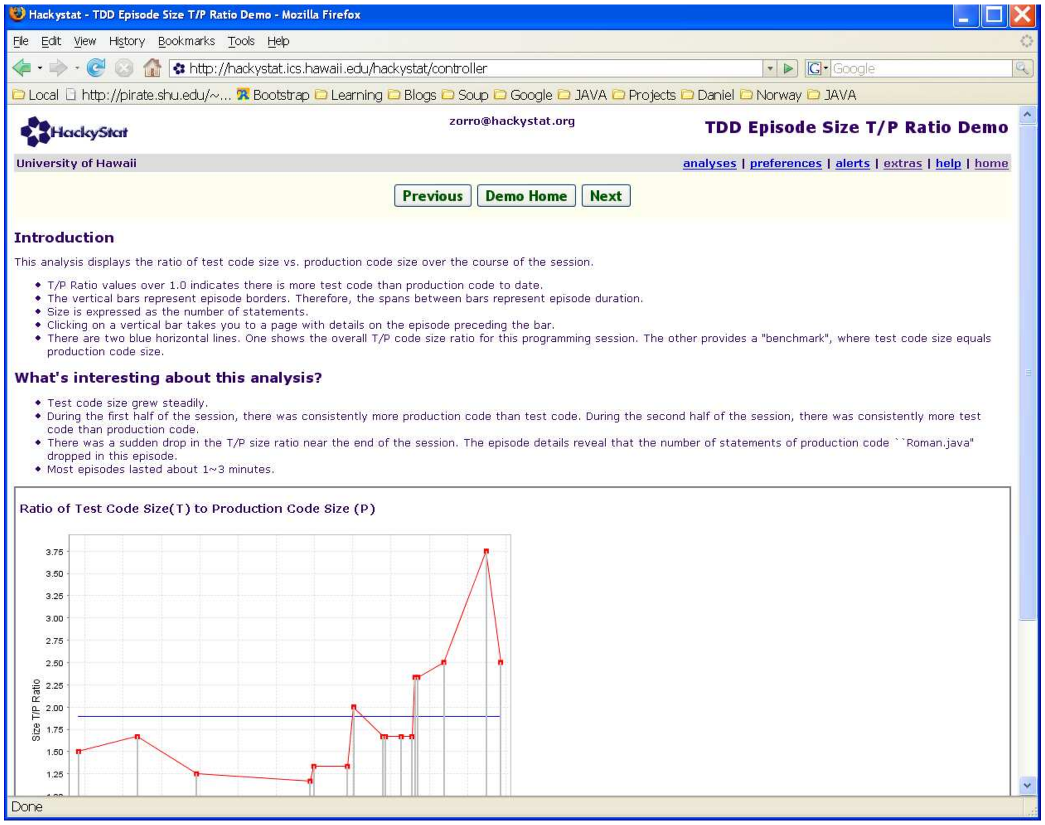
- After an initial spike of production code effort, the cumulative ratio indicates that more time was consistently spent on testing than production code development over the course of the session.
- The longest episode is near the beginning and its duration is about 7 minutes.
- Test effort increased gradually.
- At the conclusion of the session, the cumulative T/P ratio indicates that approximately the same amount of time was spent on testing and production code overall.

Ratio of Effort on Test Code (T) to Effort on Production Code(P)

Figure 8.6. TDD Episode T/P Ratio of Development Time

4. **TDD Episode T/P Ratio of Code Size**

   The fourth analysis is “TDD Episode T/P Ratio of Code Size” (Figure 8.7). Similarly. it provides another metric for measuring TDD compliance. The metric of T/P ratio of code size measures the cumulative net increase of test code and production code relatives. This analysis can easily tell whether test code is written incrementally.


Hackystat - TDD Episode Size T/P Ratio Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
HackyStat
zorro@hackystat.org
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
Size T/P Ratio
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
Done

TDD Episode Size T/P Ratio Demo

Introduction

This analysis displays the ratio of test code size vs. production code size over the course of the session.

- T/P Ratio values over 1.0 indicates there is more test code than production code to date.
- The vertical bars represent episode borders. Therefore, the spans between bars represent episode duration.
- Size is expressed as the number of statements.
- Clicking on a vertical bar takes you to a page with details on the episode preceding the bar.
- There are two blue horizontal lines. One shows the overall T/P code size ratio for this programming session. The other provides a "benchmark", where test code size equals production code size.

What's interesting about this analysis?

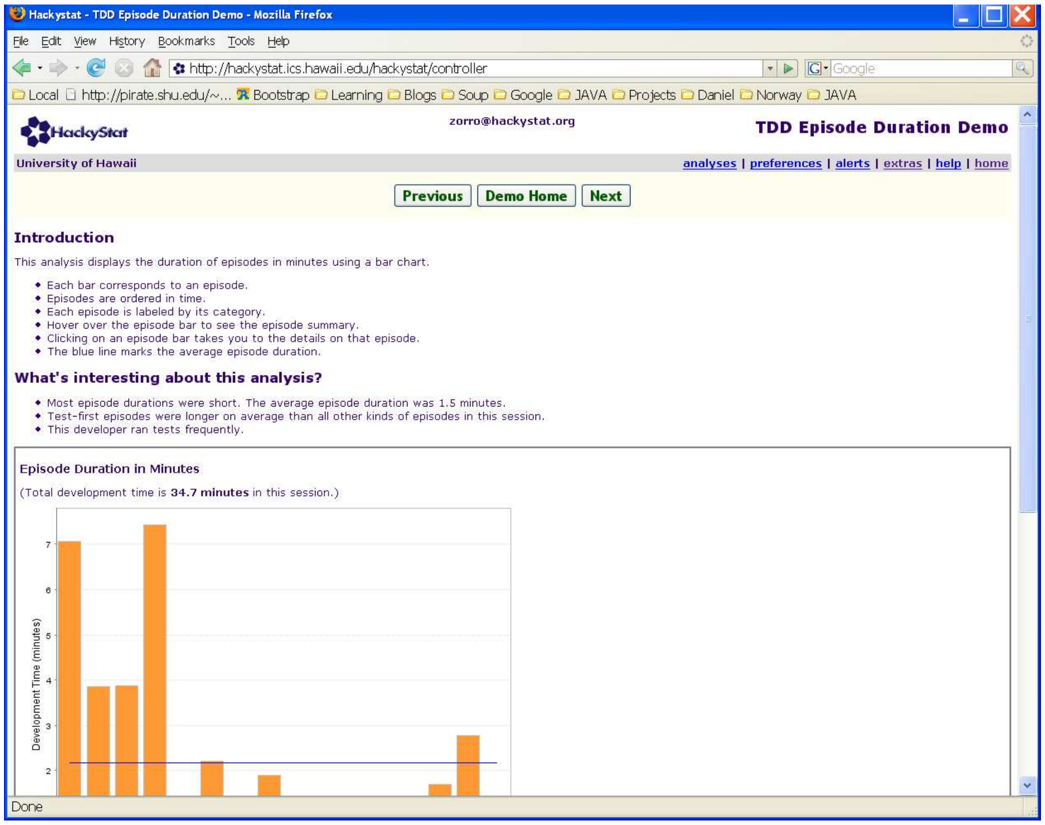
- Test code size grew steadily.
- During the first half of the session, there was consistently more production code than test code. During the second half of the session, there was consistently more test code than production code.
- There was a sudden drop in the T/P size ratio near the end of the session. The episode details reveal that the number of statements of production code ``Roman.java" dropped in this episode.
- Most episodes lasted about 1~3 minutes.

Ratio of Test Code Size(T) to Production Code Size (P)

Figure 8.7. TDD Episode T/P Ratio of Code Size

5. **TDD Episode Duration**

   The fifth analysis is "TDD Episode Duration". How often should developers invoke unit testing is a grey area of TDD. A general consensus is that an iteration of TDD should not last more than ten minutes; otherwise, the process is not agile. In Zorro, I provide the "TDD Episode Duration" analysis as a validation tool to observe duration of TDD iterations. It is reasonable that most episodes in TDD are short in length of duration.


Hackystat - TDD Episode Duration Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
HackyStat
zorro@hackystat.org
TDD Episode Duration Demo
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
Done

Introduction

This analysis displays the duration of episodes in minutes using a bar chart.

- Each bar corresponds to an episode.
- Episodes are ordered in time.
- Each episode is labeled by its category.
- Hover over the episode bar to see the episode summary.
- Clicking on an episode bar takes you to the details on that episode.
- The blue line marks the average episode duration.

What's interesting about this analysis?

- Most episode durations were short. The average episode duration was 1.5 minutes.
- Test-first episodes were longer on average than all other kinds of episodes in this session.
- This developer ran tests frequently.

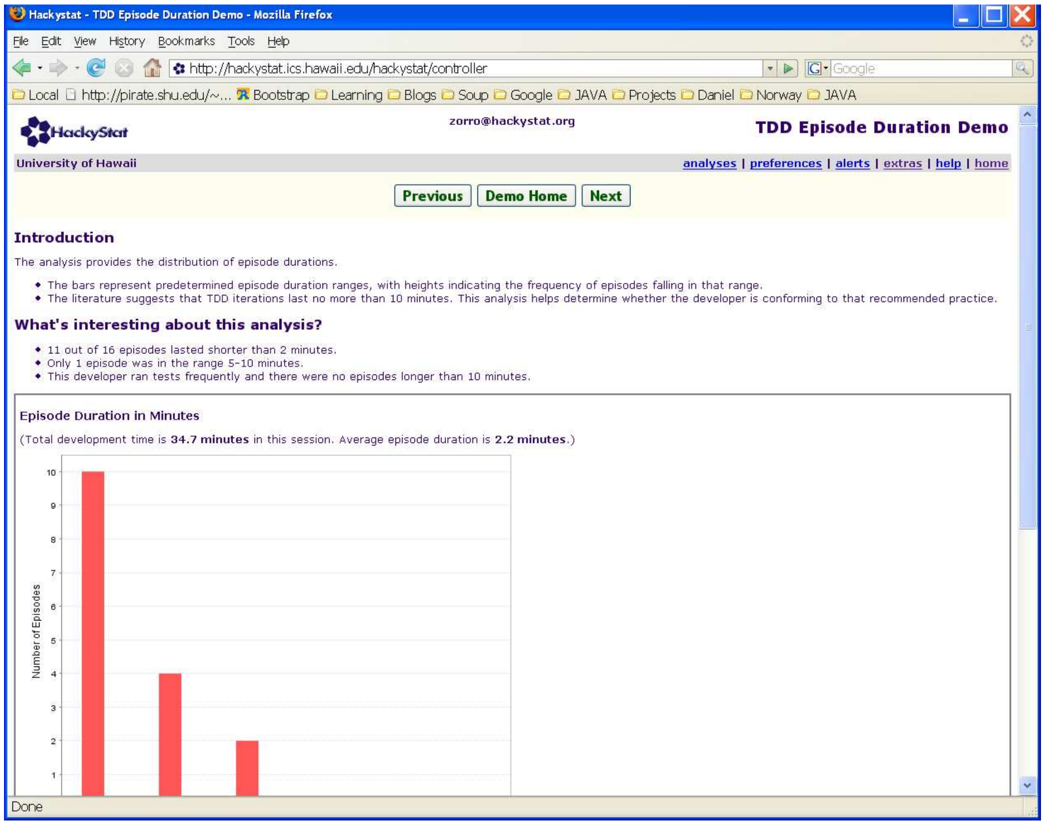
Episode Duration in Minutes

(Total development time is **34.7 minutes** in this session.)

Figure 8.8. TDD Episode Duration

6. **TDD Episode Duration Bins**

The last analysis is "TDD Episode Duration Bins". Simply as its name indicates, this analysis puts episodes into a set of bins according to their duration. Each bin defines a range of duration. In Figure 8.9, the vertical axis is for numbers of episodes that fall into each bin.


Hackystat - TDD Episode Duration Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
HackyStat
zorro@hackystat.org
TDD Episode Duration Demo
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
Number of Episodes
10
9
8
7
6
5
4
3
2
1
Done

Introduction

The analysis provides the distribution of episode durations.

- The bars represent predetermined episode duration ranges, with heights indicating the frequency of episodes falling in that range.
- The literature suggests that TDD iterations last no more than 10 minutes. This analysis helps determine whether the developer is conforming to that recommended practice.

What's interesting about this analysis?

- 11 out of 16 episodes lasted shorter than 2 minutes.
- Only 1 episode was in the range 5-10 minutes.
- This developer ran tests frequently and there were no episodes longer than 10 minutes.

Episode Duration in Minutes

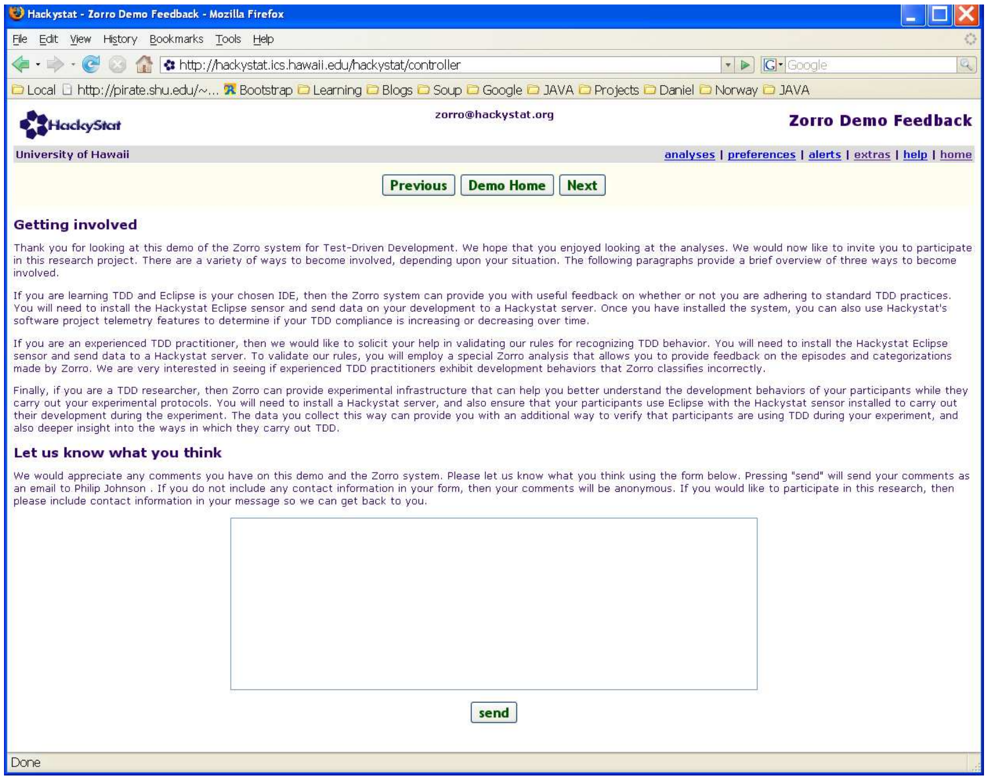
(Total development time is **34.7 minutes** in this session. Average episode duration is **2.2 minutes**.)

Figure 8.9. TDD Episode Duration Bins

To the TDD programming session analyzed in this demo, most episodes fell into bins of short durations according to Figure 8.9.

7. **Zorro Demo Feedback**

The last page of the "Zorro Demo" is "Zorro Demo Feedback" that invites readers to collaborate on studying TDD. With this demo, we targeted at three kinds of users: TDD beginners, experienced TDD practitioners, and TDD researchers. After reading information about collaborative research opportunity with us, readers can provide feedback to us using the box included in this page (Figure 8.10).


Hackystat - Zorro Demo Feedback - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Google
Local http://pirate.shu.edu/~... Bootstrap Learning Blogs Soup Google JAVA Projects Daniel Norway JAVA
HackyStat
zorro@hackystat.org
Zorro Demo Feedback
University of Hawaii
analyses | preferences | alerts | extras | help | home
Previous
Demo Home
Next
send
Done

Getting involved

Thank you for looking at this demo of the Zorro system for Test-Driven Development. We hope that you enjoyed looking at the analyses. We would now like to invite you to participate in this research project. There are a variety of ways to become involved, depending upon your situation. The following paragraphs provide a brief overview of three ways to become involved.

If you are learning TDD and Eclipse is your chosen IDE, then the Zorro system can provide you with useful feedback on whether or not you are adhering to standard TDD practices. You will need to install the Hackystat Eclipse sensor and send data on your development to a Hackystat server. Once you have installed the system, you can also use Hackystat's software project telemetry features to determine if your TDD compliance is increasing or decreasing over time.

If you are an experienced TDD practitioner, then we would like to solicit your help in validating our rules for recognizing TDD behavior. You will need to install the Hackystat Eclipse sensor and send data to a Hackystat server. To validate our rules, you will employ a special Zorro analysis that allows you to provide feedback on the episodes and categorizations made by Zorro. We are very interested in seeing if experienced TDD practitioners exhibit development behaviors that Zorro classifies incorrectly.

Finally, if you are a TDD researcher, then Zorro can provide experimental infrastructure that can help you better understand the development behaviors of your participants while they carry out your experimental protocols. You will need to install a Hackystat server, and also ensure that your participants use Eclipse with the Hackystat sensor installed to carry out their development during the experiment. The data you collect this way can provide you with an additional way to verify that participants are using TDD during your experiment, and also deeper insight into the ways in which they carry out TDD.

Let us know what you think

We would appreciate any comments you have on this demo and the Zorro system. Please let us know what you think using the form below. Pressing "send" will send your comments as an email to Philip Johnson . If you do not include any contact information in your form, then your comments will be anonymous. If you would like to participate in this research, then please include contact information in your message so we can get back to you.

Figure 8.10. Zorro Demo Feedback

2006-11-07: Feedback from ISERN

As the first step to find collaborative opportunity with researchers of TDD, we sent an RFC of Zorro Demo to the ISERN mailing list that contains researchers from worldwide who are interested in empirical software engineering research. On the same day we sent this collaboration solicitation email, Dr. Geir Hanssen and Dr. Tor Erlend Fægri from SINTEF ICT of Norway gave very positive feedback on Zorro, and expressed that they were very interested in using Zorro in a comparative study between TDD and an existing Test-Last practice.

They planned to conduct this study under the assistance from an European software company by harvesting data such as time spent for testing/coding, test code size/production code size etc. They were also interested in finding ways of validating any effect that the TDD-practice may have on product quality, speed of production, and also potentially on code quality (maintainability, code design-style and others). However, doing these was very hard in practice because the software company did not want to be disturbed when developers put maximum pressure on reaching the release deadline. As a result, collecting data became a problem in this already planned study. Thus, automated data collection became tempting and Zorro is such a tool that can bring immediate values to them.

Although Zorro was a good fit to this study, the software company used Visual Studio .NET rather than Eclipse, and the programming language was C# rather than Java. A minimum set of sensor data types (See Table 4.1) are required in order for Zorro to function. Unfortunately, the Visual Studio .NET senor was very premature and it can only collect a few of them. Thus, if we wanted to engage in this study, the most imperative task was to develop a Visual Studio .NET sensor that can collects necessary sensor data types. Moreover, the sensor had to be implemented and tested in a very short time frame since the software company planned to start the project in January 2007.

8.3.2 Preparation

With confidence in Zorro and our software development skills, we decided to participate in this collaborative industrial case study. The preparation tasks include implementation of the Visual Studio .NET sensor and installation of Hackystat.

Visual Studio .Net Sensor Implementation

We exchanged several emails with Dr. Hanssen to understand the company's development environment in December 2006. With the help from Qin Zhang, another Ph.D student who was also my colleague in CSDL, I implemented the Visual Studio .NET sensor that was Zorro-compatible. After developing the sensor, I tested that Zorro can infer TDD development behaviors occurred in the IDE of Visual Studio .NET.

Hackystat Server Installation

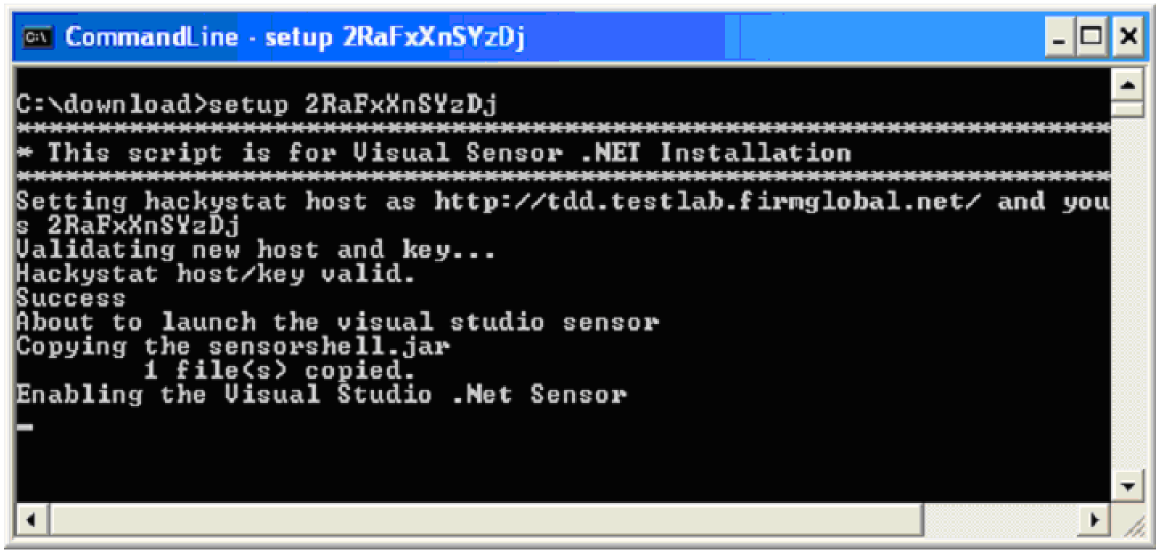
Zorro runs on the server-side of Hackystat. Foosball LLC provided a dedicated server for this study. Using Windows' Remote Desktop application, I logged into the server, installed and configured a Hackystat server for this case study in February 2007.

8.3.3 Collaborative Research Activities

In March 2007, the company launched both the TDD project and the Non-TDD project with 20 developers, twelve of which developed in TDD. I assisted developers installing the Visual Studio .NET sensor, which collected development activities in the background unobtrusively. Periodically, I analyzed the collected data and generated technique reports for Dr. Hanssen and the project manager regarding developers' development behaviors.

2007-02-22: Sensor Installation Instruction

Pertaining to the least possible interruption to the development process required by the participated company, I wrote a script to automate the sensor installation for developers. Figure 8.11 illustrates the output from this script when a developer invoked this script. For each developer, I sent an individual email to him/her with customized installation instructions. Only one developer encountered difficult in installing the sensor due to his customization of the Windows user profile. I fixed this by patching the installation script.



```
C:\download>setup 2RaFxXnSYzDj
**************************************************************************
* This script is for Visual Sensor .NET Installation
**************************************************************************
Setting hackystat host as http://tdd.testlab.firmglobal.net/ and you
s 2RaFxXnSYzDj
Validating new host and key...
Hackystat host/key valid.
Success
About to launch the visual studio sensor
Copying the sensorshell.jar
        1 file(s) copied.
Enabling the Visual Studio .Net Sensor
_
```

Figure 8.11. Output of Visual Studio .NET Sensor Installation Script

2007-02-28: #1 admin guide and sensor installation status

The admin guide is a document for Dr. Hanssen and the project manager to use Hackystat and Zorro. This document covers following contents:

- Login as the administrator,
- Login as any developer,
- Check sensor installation status (days with sensor data),
- Check a developer's collected development activities,
- Invoke the analyses described in the Zorro Demo.

Following this guide, they can login as the administrator, check received data, define Hackystat projects, and invoke Zorro's analyses.

I also summarized the sensor installation status based on data received on the server. According to this report, two developers successfully installed the sensor within a day after I sent the instructions to them.

2007-03-02: #2 admin guide

I updated the admin guide on March 2, 2007. According to this guide, some developers sent data to the Hackystat server. After observing the collected data, I registered projects for developers who installed the sensor so that the project manager can use Zorro to analyze their development behaviors.

When writing the sensor installation status report, I was surprised to find that none of developers invoked any unit tests. The project manager noticed this as well and asked us whether the sensor worked correctly with TestDriven.NET add-in. Unfortunately I never knew that developers ran tests with TestDriven.NET, a third-party add-in to the Visual Studio .NET. Later, we discovered that some emails from the project manager sent in January 2007 were discarded by the mail server of the University of Hawaii as SPAMs for unknown reasons. I configured my SPAM settings to allow emails sent from the software company, but I had to enhance the sensor to support TestDriven.NET quickly in order to continue this collaborative research.

2007-03-05: Support for TestDriven.Net and Sensor Upgrade

TestDriven.NET was an open source project initially but turned into a proprietary Visual Studio add-in. It is not as extensible as the JUnit plugin of Eclipse so that a third-party observer can not listen to invocations of unit testing. Perhaps because Microsoft started a legal issue against him, the author of TestDriven.NET did not respond to my request for help. Soon I discovered an approach to capture and parse test running results from TestDriven.NET by observing the console output in Visual Studio .NET. After some extensive testing, I released a new Visual Studio .NET sensor for this collaborative industrial case study.

It was a little pity that the project manager was very unhappy with this update but there was no alternatives if we wanted to continue this research. As a result, some developers did not install this upgrade according to my observation of the server status. I addressed this issue in the following reports I wrote.

2007-03-20: #3 sensor installation status

I updated the sensor installation status report on March 20, 2007. According to this report, 20 developers fell into five groups:

- **Group 1: Those who are successfully using the system**

  Four developers are sending coding and testing data regularly, indicating the sensor is installed and updated.

- **Group 2: Those who need to install the sensor**

  Four developers have not sent any data. They appear to have not installed the sensor at all or are not programming.

- **Group 3: Those who might have uninstalled the sensor**

  Four developers sent data at the beginning but have not sent any data recently. They have either uninstalled the sensor or are not programming.

- **Group 4: Those who might need to update the sensor**

  Four developers are sending coding data but not testing data. Either they are not testing, or have not updated the sensor.

- **Group 5: Those who apparently need to update the sensor**

  Four developers are sending coding data and testing data, but the test data lacks path information indicating the need to upgrade the sensor.

Because I conducted this study off-site, the above categorization of developers was only based upon my observation of data received on the Hackystat server. The project manager generously agreed to have developers update the sensor following my suggestions made in this report.

2007-04-14: #4 introduction to the TDD telemetry analyses

In the report I wrote on April 14, 2007, I introduced the TDD telemetry analyses (See Section 4.3.2) to Dr. Hanssen and the project manager. In addition, I defined two Hackystat projects for them. One project was "The-TDD-Project" which includes all programmers who developed in TDD as project members, and the other project was “The-NON-TDD-Project”. With these two projects, they can invoke TDD telemetry analyses to compare the TDD project and the Test-Last project. A list of TDD telemetry analyses is in the following.

- **DevTime-Chart:** computes development time developers spent in the Visual Studio .NET environment.
- **DevTime-Members-Chart:** computes and plots all project members’ development time spent in the Visual Studio .NET environment.
- **DevTime-TotalProductionTest-Chart:** computes and illustrates total development time, the portion on coding, and the portion on testing.
- **EpisodeDurationAverage-Chart:** illustrates the average episode durations.
- **TDD-Percent-Chart:** reports the percentage of a project’s TDD compliance.
- **TDD-All-Members-Chart:** reports and compares project members’ TDD compliance.
- **TDDPercent-And-DevTime-Chart:** combines the TDD compliance percentage and development time to a given project.

Basically this report opened a door to Zorro and Telemetry for Dr. Hanssen and the project manager. With the sensor data collected from developers, Zorro can be used to not only infer developers’ TDD development behaviors but also conduct comparative study between TDD and Non-TDD.

According to feedback received from Dr. Hanssen regarding this report, the number of programmers producing data was disappointing, and he would contact the project manager directly for possibility to improve it. However, he viewed this study as a pilot and planned to replicate it in another software company, which might have a better start and more complete data. Also, other researchers are interested in Zorro’s automation of data collection and development behavior inference.

2007-05-04: #5 Comparison Between TDD and Test-Last Projects

I used Zorro's TDD telemetry analyses to compare development behaviors between the TDD project, and the Test-Last project and wrote a technique report on May 4, 2007. The main points included in this reports are:

- **TDD-Percent-Chart (by development time)**

  The "TDD-Percent-Chart" telemetry analysis computes and plots percentages of development time that developers complied to TDD based upon Zorro's inference. Partially due to the imperfect sensor installation, developers did not comply to TDD faithfully in this study. The compliance percentage was 5% at the best on the weekly basis.

  Living far from developers who participated in this study, I did not have the capabilities to investigate causes for the low TDD compliance rate. Perhaps developers did not have the necessary skills to use TDD on their development tasks, or they modified TDD according to their needs.

- **TPRatio-DevTime-Chart**

  The "TPRatio-DevTime-Chart" telemetry analysis is a metric for measuring effort developers devoted to testing. It was interesting to note that developers who were in TDD project evenly divided their effort on testing and production code. On the contrary, developers in the Test-Last project did not test much at the beginning but gradually caught up unit testing.

- **DevTime-TotalProductionTest-Chart**

  The "DevTime-TotalProductionTest-Chart" telemetry chart analysis plots cumulative effort spent on test and production code. According to this analysis, the TDD project had larger growth rate of test code while the Test-Last project had larger growth rate of production code.

- **DurationAverage-Chart**

  The "DurationAverage-Chart" telemetry analysis illustrates how frequently developers invoked unit tests. Interestingly, the episode durations average of the two projects are far longer than 10 minutes.

In addition to the telemetry report, I updated the sensor installation status with development time and T/P ratio for each developer. After reading this report, Dr. Hanssen were more concerned about the legitimacy of the data rather than the hypotheses I claimed regarding the development process. Unfortunately, the project manager concentrated on the software development and did not respond to this report.

2007-05-25: #6 Comparison Between TDD and Test-Last Projects

Based on the report I wrote on May 25, 2007, developers in the Test-Last project started putting more effort on testing than effort on production code, in contrast developers in the TDD project evenly divided their effort on testing.

8.3.4 Phone Interview with the Researcher

By the time I write this chapter, developers in Foosball LLC are still working on the software to make the next release. In this research, I installed and configured a Hackystat server for them, developed the Visual Studio .NET sensor, managed the Hackystat server, and wrote six technique reports for the researcher and the project manager. After four months collaboration, Professor Philip Johnson and I interviewed the researcher on July 3, 2007. Table 8.1 lists the structure of this interview.

Table 8.1. Structure of the Interview with the Researcher

Interviewer	Hongbing Kou, Philip Johnson
Interviewee	Geir Hanssen
Instrumentation	Voice Recorder
Method	Phone
Time	9PM July 3, 2007
Length	45 minutes
Location	Collaborative Software Development Laboratory

In the interview, we mainly focused on the experiment details behind the scene as well as the researcher's experience regarding this collaborative research.

- Research Background

  It is interesting to note that the Foosball LLC was not specially recruited to participate in this comparative study of TDD. The company worked with Dr. Hanssen before on studying Evo

Agile, an agile process that the company has adopted. For whatsoever reason, the company decided to include TDD in its software process and so this case study was initiated. After seeing the Zorro Demo, Dr. Hanssen suggested that they use Zorro for data collection in this study. Because of the unobtrusiveness of Zorro's data collection, the Foosball LLC agreed to have its process instrumented.

- TDD Conformance

  The TDD compliance percentage of this study was very low according to Zorro's inference. However, probing the development process to understand the situation was impossible for me and Dr. Hanssen as well at this moment. Typically, the Foosball LLC has good reputation in complying to what they agreed to, but meeting the deadline has the top priority. It is worthwhile to note that Evo Agile, the software process that the company is using, has tight and well scheduled development activities. TDD is an extra addition to the process, which might help to explain why the project manager did not respond to our requests for understanding their software process.

- Concerns on Sensor Installation

  In the #3 report of "Sensor Installation Status Update", I described the sensor installation status after a month's deployment. We were confident that 4 developers successfully installed and updated the sensor as requested, but not sure what happened to the rest developers. A real problem occurred in this study was from the TestDriven.NET plugin. I had to enhance the Visual Studio .NET sensor rapidly, and in turn, developers had to update the sensor shortly after installation. Likely some developers gave up the sensor although upgrading the sensor was a trivial task from our perspective.

  As concerned as we were on this problem, Dr. Hanssen said that he will call each developer after the project is released. A questionnaire will be designed regarding the development process, sensor installation and Zorro's inference results.

- Experience on Collaborative Research with Industry

  Collaborative research with industry in generally is hard. In this research, the project manager did not respond to Zorro's inference regarding their development process. To the industrial participants, meeting deadlines of project releases have much higher priority than engaging in research activities. In Dr. Hanssen's opinion, this is just the way it is, and as researchers, we have to adopt this. A few key points he described in the interview were:

- This study is a training to use tools such as Zorro that automate the data collection.
- What we can do better in the future is to let developers install Zorro, make sure that it works.
- He is interested in conducting more of this kind of study. Other companies are interested in TDD too.

8.4 Conclusions

This study shows that Zorro can support empirical research of TDD. The automation of data collection makes it possible for researchers to collect data without interrupting the development process. It provides supporting evidence to the research question Q3a, but it also shows that installing sensors to enable automated data collection can be a challenging task to the participants.

8.5 Chapter Summary

This chapter introduces the industrial case study I collaboratively conducted with Dr. Hanssen, a TDD researcher, and an European software company. In this collaborative study, I provided technique support of Hackystat and Zorro.

The lessons I learned from this study are:

- **Pilot**

  Pilot is a must for studies to be conducted in the industrial settings. Participants are very precious resources, and if possible, all the research activities should be pilot tested beforehand. Otherwise, it will be very easy to make mistakes that may blow up the research opportunity.

- **On-site**

  Another lesson I learned from this study is that case studies should be conducted on-site instead of off-site. Responding to requests from researchers is more or less a burden to the participated companies. In case when the schedule is tight, participants may ignore requests from researchers. To effectively conduct case studies, the researchers can get maximum input from participated companies if they can go to the field to conduct research.

- **Periodical Sensor Installation Status Check**

  With automated data collection, researchers may tempt to focus on other research areas instead of data collection. The lesson we learned from this research is that periodically checking data collection is necessary because many things can go wrong. For example, some unit testing activities were not collected at the beginning of this industrial case study. Also, developers might silently decline the request to instrument their development processes with sensors.

- **The Author's Involvement**

  Finally, so far my involvement is still necessary in order to use Zorro in the evaluation case studies of TDD. In this industrial case study, I deployed Zorro to the site, assisted developers installing the sensor, analyzed the data, and wrote analysis reports for the researcher. For those who have no prior experience to Hackystat and Zorro, adopting Zorro in the empirical evaluation studies might be hard.

Chapter 9

Research Summary, Contributions and Future Directions

This chapter finishes up this thesis. It begins with the research summary in Section 9.1, followed by contributions in Section 9.2. Then it discusses future directions in Section 9.3.

9.1 Research Summary

In the software engineering field, researchers and practitioners have put increasing effort on low-level software processes [1, 46] such as PSP, TSP and Agile Processes. Though proven to be useful in improving software quality[21, 43, 70, 31], low-level software processes are hard to execute correctly and repeatedly. Also, low-level software processes have the potential to require new skills from software organizations, project managers, and software developers. For example, in Test-Driven Development (TDD), each developer is a requirements analyst, designer, tester, and coder. As a result, a low-level software process could be used differently in different software organizations. Worse yet, an organization might think they are using a particular low-level process, such as TDD, but in reality, they are doing something quite different. Thus, I put my research effort on automated recognition of low-level software process development behaviors. As a step in this research area, I focused on one low-level software process called TDD and implemented the Zorro software system to automate the recognition of development behaviors of TDD. In addition, I proposed the Software Development Stream Analysis (SDSA) framework to assist the research on low-level software processes.

Test-Driven Development (TDD), a core practice of Extreme Programming, has been widely adopted by software industry and studied by software engineering researchers. So far, software engineering researchers have focused most of their energy on the outcomes that applying TDD brings to software products and software developers. However, compared to the claims made by practitioners, research findings of TDD on software quality and developer productivity are mixed. In fact, much of the research work on TDD suffers from the threat of “construct validity” [74] because of the “process conformance” problem. Janzen and Saiedian [31] warn that the inability to accurately characterize process conformance is harmful to TDD research, and that it is so hard to measure the usage of a development method such as TDD that current reports on adoption of TDD are not valid. Fortunately, with the development of sophisticated software metrics collection system such as Hackystat [35], it is possible to study the process conformance of TDD.

In order to study the process conformance of TDD, I implemented the Zorro software system with the aids of the Hackystat and SDSA frameworks. Hackystat sensors instrument the development environments to collect software process and product metrics. SDSA is a framework that I have created for studying low-level software processes. Using SDSA, Zorro abstracts a variety of software metrics into development activities, merges these activities together to form a time-series software development stream, and finally partitions the stream into a group of episodes ended with successful unit test invocations. In order to infer development behaviors of TDD, I defined a set of specific rules in Zorro according to Beck [5, 6] and others who have described the practices of TDD. The “test-pass” episodes are categorized as “test-first”, “refactoring”, “test-addition”, “regression”, “code-production”, “test-last”, “long”, or “unknown”. After inferring development behaviors in episodes and categorizing them, Zorro uses the classification results as well as the context of episodes to reason the conformance of TDD. Moreover, with the inferred results, Zorro implements a handful of analyses that are grouped into two categories. The first category of analyses study a single programming session and report different aspects of TDD. One analysis in this category is the “TDD Episode Demography” analysis that can be used to look for the development patterns of TDD. The second category of analyses leverage software project telemetry [40, 79] that can support interpersonal in-process project management and decision makings in the granularities of daily, weekly and monthly. For instance, I compared differences of developers’ testing effort between

“The-TDD-Project” and “The-NON-TDD-Project” on the weekly basis using the telemetry stream of “TPRatio-DevTime-Chart” in the industrial case study (Chapter 8).

In order to empirically evaluate Zorro, I have conducted three case studies — a pilot study (Chapter 6), a classroom case study (Chapter 7), and an industrial case study (Chapter 8) to investigate whether Zorro can collect sufficient software metrics and how well it can infer TDD compliance. I summarize the research findings of these studies in the following.

9.1.1 Data Collection

One of my primary focuses was on evaluating Zorro’s data collection because collecting necessary development activities is a must for Zorro to infer development behaviors. I implemented the Eclipse Screen Recorder (ESR [17]), an Eclipse plug-in that can record the software development activities occurred in the Eclipse IDE. Most importantly, it can be configured to record the Eclipse screen in the frequency of one frame per second and the recorded video file size is just 5-7MB per hour. With the help of ESR, I validated Zorro’s data collection in the pilot study and the classroom study.

According to my research in these two studies, Zorro is capable of collecting development activities. In the classroom case study, I found that Zorro on average collected more development activities (16.8 per episode) than what I observed (14.9 per episode) in the recorded ESR videos (See Table 7.9). Given that ESR can capture almost every activity occurs in the Eclipse IDE, Zorro does a good job in collecting development activities.

In both the pilot and the classroom studies, I compared two sources of data side by side to discover any hidden problems in Zorro’s data collection. It turned out that a few problems (Sections 6.5.3 and 7.6.2) existed but none of them were significant. Instead, the G2-DevBehavior, a development behavior that I discovered in the classroom case study, significantly impacted Zorro’s inference accuracy of development behaviors and compliance of TDD. If there were not G2-DevBehavior, Zorro could infer TDD compliance with 90+% accuracy (Section 7.6.3). Thus, correctly recognizing the G2-DevBehavior has the potential to greatly improve Zorro’s reliability.

9.1.2 Development Behavioral Inference

The other primary focus was on evaluating Zorro's inference of TDD development behaviors. I evaluated the development behavioral inference with the help of ESR in both the pilot study and the classroom study. In the data analyses, I watched the recorded ESR videos to observe participants' development behaviors and used the observed results to validate Zorro's inference.

Compared to participant observation, Zorro's inference accuracy of development behaviors is 88.4% in the pilot study (See Table 6.2). This value is 70.1% in the classroom study (See Table 7.10). However, these two values can not be directly compared since I revamped the classification of development behaviors in Zorro after the pilot study. A notable phenomenon occurred in the classroom case study was the so called G2-DevBehavior (Section 7.6.1), which diverted Zorro's partitioning of episodes and inference of development behaviors. Further investigation in Chapter 7 concludes that the inference accuracy is 89.6% for group G1 who did not conduct G2-DevBehavior. All in all, it indicates that allowing Zorro to interpret G2-DevBehavior is necessary.

9.1.3 Usefulness

Last, I also focused on evaluating Zorro's usefulness in the case studies. The "Episode Demography" and "T/P Effort Ratio" are two most useful analyses for beginners to understand and improve TDD practice based on evaluation results of the classroom study. In the industrial case study, I collaborated with Dr. Hanssen, a researcher who conducted a comparison study between TDD and an existing Test-Last practice in an European software company. This research supports the conclusion that Zorro's automated unobtrusive data collection and inference of development behaviors are useful for researchers to collect data without interrupting the development process.

9.2 Research Contributions

This research has three main contributions:

- Software Development Stream Analysis Framework,
- Automated recognition of TDD with Zorro,

- Empirical evaluation of Zorro.

9.2.1 Software Development Stream Analysis (SDSA) Framework

The SDSA framework is the most significant contribution of this research. A problem with low-level software processes is that many organizations might use them differently based upon their understanding, which makes it hard to study their impacts on software development. The SDSA framework can automatically evaluate how well an organization executes software processes with only minimum interruption to the development process.

In order to study low-level software development processes, SDSA abstracts development activities of a programming session into a software development stream, a linear and time-series data structure. Corresponding to the incremental and iterative property of many low-level software processes, SDSA uses tokenizers to partition a long development stream into many short episodes, another abstract data structure representing a micro-iteration of a software process. Finally, SDSA uses JESS, a rule-based system in Java to recognize and classify development behaviors of partitioned episodes.

In my thesis research, I instantiated the SDSA framework on Test-Driven Development (TDD), and the system resulting from this work is the Zorro software system. Zorro can automatically infer the development behaviors and the compliance of TDD. This research work demonstrates that the SDSA framework has the potential to be useful for researching other low-level software processes.

9.2.2 Automated recognition of TDD with Zorro

Another significant contribution is the Zorro software system that was built on the top of the Hackystat and SDSA frameworks. Zorro recognizes TDD development behaviors conducted in an IDE as long as its sensor supplies all required metrics listed in Table 4.1. In my research, I enhanced the Eclipse sensor and evaluated its data collection capability in the pilot and classroom case studies. As part of the industrial case study, I enhanced the Visual Studio .NET sensor to be Zorro compatible.

Many analyses such as “Episode Demography” were developed to leverage Zorro’s reasoning of development behaviors of TDD. Furthermore, with the help of software project telemetry, I developed a set of telemetry reducers of TDD to support management of TDD projects.

Also, I implemented a rule set for TDD based upon the descriptions of many well-known TDD practitioners including Beck [6], Doshi [15] and Erdogmus [74], and additionally my grounded observation of TDD in practice.

9.2.3 Empirical evaluation of Zorro

The third contribution is the empirical evaluation studies I conducted. The pilot and classroom studies exemplify a paradigm of empirical validation of Zorro. All the research materials were made public for others to validate Zorro in different environments. The industrial case study demonstrates how to conduct TDD research that does not suffer from the process conformance problem with the help of Zorro. In addition, the actual results of these studies are also my contributions.

1. The Eclipse sensors collect sufficient in-process metrics for inference of TDD;
2. Zorro can identify TDD when it occurs in the IDEs of Eclipse and Visual Studio .NET;
3. Zorro is helpful for beginners to understand and conform to TDD;
4. Zorro is useful for researchers to conduct TDD evaluation studies.

9.3 Future Directions

9.3.1 TDD Evaluation Studies

First, Zorro can be used in evaluation studies of TDD to improve validity of research conclusions. The industrial case study I conducted in my thesis research is an attempt in this direction.

9.3.2 Unified operational definition of TDD

Second, reaching an unified operational definition of TDD is necessary. Beck uses the red/green/refactor to describe the order of TDD programming in [6]. Following this abstraction, I defined three types of development behaviors “test-first”, “refactoring” and “test-last” in the prototype implementation of Zorro. Then I used this prototype in the pilot study in which I surprisingly found that half of the episodes were “test-last”, which is very contradict to my intuition. With the pilot study, I realized that TDD in reality is quite different from in theory, and revamped Zorro’s classification of TDD development behaviors (Section 4.2.3). Moreover, I introduced a heuristic algorithm for inference of TDD compliance (Section 4.2.4) to Zorro. This is an operational definition of TDD. It works well according to the classroom case study conducted in my thesis research, but the conclusion is limited to the environment I tested. More replication studies need to be conducted in order to reach a unified operational definition of TDD that can be agreed upon by the community of TDD practitioners and researchers.

9.3.3 More practical uses of Zorro’s inference results

Third, finding more practical use of Zorro is one more future direction. In my research, I designed and implemented some typical analysis such as “Episode Demography” and “Episode Duration Distribution” and some TDD telemetry streams. My initial evaluation concludes that they are useful for beginners and researchers of TDD. However, in order to fully use Zorro’s potentials in software project management and software process improvement, finding more practical uses of Zorro is important.

One practical use of Zorro is to study how to interpret Zorro’s inference results. For example, will 100% be necessary in actual software projects? Or how much compliance of TDD is acceptable? Another interesting use of Zorro is to compare the inferred results to test coverage. Beck claims that TDD should yield 100% test coverage automatically, but more research needs to be done to study this claim.

Zorro can be used as a CASE (Computer Aided Software Engineering) tool for software project management. I applied Zorro to a TDD and a Non-TDD projects in the industrial case study. With the help of Zorro, I generated “sensor installation status” and “TDD telemetry” for the project manager. These reports helped the project manager to realize that problems existed in sensor installation and data collection. However, this research work only uses a few analyses Zorro provides. Additional research needs to be conducted to study how to use Zorro for software project managements.

9.3.4 Other low-level software processes

Zorro was built on the top of the more generic SDSA and Hackystat frameworks, this architecture makes it easily possible to study other low-level software processes or best practices of software development as well. For example, low-coupling is a desired property to objects in the object-oriented programming. With Hackystat and SDSA, we can easily define rules to find objects that are either excessively edited by developers, or overly referred by other objects in the development process.

9.3.5 Data mining

At last, applying some data mining algorithms on software development streams may create interesting applications. The SDSA framework sorts a variety of software metrics collected by Hackystat sensors and organizes them as time-serious software development streams. I applied a rule-based system on them to infer development behaviors and compliance of TDD in my research. The rule-based system is powerful for recognizing well-defined development behaviors such as TDD. However, in cases that expected development behaviors are unknown, it will be better to use some data mining algorithms. For example, Heierman et. al. introduce the Episode Discovery [28] algorithm to discover and classify naturally recurring patterns from temporal sequences of human-generated activities. This algorithm can be used to mine the software development streams to find recurring patterns of development activities.

Appendix A

Pilot Study Material

A.1 Introduction to TDD

Test-driven development is a new way to develop software. With TDD developers *(1) write new code only if an automated test has failed; (2) eliminate duplication iteratively in software development.* We will be implementing a stack data structure in TDD. Please keep this in mind while you are participating this study. I provided you with a quick reference [15] and the rhythm of TDD [16] to help you do TDD programming.

A.1.1 TDD Quick Reference

(Picture of Gunjan Doshi's TDD quick reference guide [15].)

A.1.2 Rhythm of TDD

(Picture of Gunjan Doshi's TDD rhythm guide [16].)

A.2 Stack Implementation in TDD

I provide additional instructions for this pilot study. This section includes description and instructive procedure to implement the stack data structure in TDD. Stack works in Last-In-Last-Out (LILO) principle. Its operations include *Push*, *Pop*, *Top*, and *isEmpty*.

- The *Push* function inserts an element onto the top of the *Stack*.
- The *Pop* function removes the topmost element and returns it.
- The *Top* function returns the topmost element but does not remove it from the *Stack*.
- The *isEmpty* function returns true when there are no elements on the *Stack*.

Note: some of this documentation are excerpted from [51].

1. **Test List (or TO-DO list)**

   The first step is to brainstorm a list of tasks. The goal of this activity is to create a task list from the requirements. Note that this list does NOT have to be completed at beginning and you may dynamically maintain it on the fly. Here is a task list example maintained by Kent Beck in his book "Test-Driven Development by Example" [6]:

   $5 + 10 CHF = $10 if rate is 2:1
   ~~$5 * 2 = $10~~
   Make "amount" private
   ~~Dollar side-effects?~~
   Money rounding?
   equals()
   hashCode()

   Same as Beck did, you may work out a list of tasks for stack.

   - Create a *Stack* and verify that *isEmpty* is true.
   - *Push* a single object on the *Stack* and verify that *isEmpty* returns false.
   - *Push* a single object, *Pop* the object, and verify that *isEmpty* returns true.
   - *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.

- *Push* three objects, remembering what they are; *Pop* each one, and verify that they are removed in the correct order.
- *Pop* a *Stack* that has no elements.
- *Push* a single object and then call *Top*. Verify that *isEmpty* is false.
- *Push* a single object, remembering what it is; and then call *Top*. Verify that the object returned is the same as the one that was pushed.
- Call *Top* on a *Stack* with no elements.

2. **Choose the First Test**

   There is a list of tasks to start with. The philosophy of TDD is to choose the simplest test that gets you started and solves a small piece of the problem. The simplest one in the list is: "Create a Stack and verify that isEmpty is true." It is also an option to choose a test that describes the essence of what you are trying to accomplish. Using stack as an example, functions **Push** and **Pop** are essential.

3. **Test 1: Create a Stack and verify that isEmpty is true.**

   You start with a class called TestStack and add one assertion to check whether isEmpty returns truth.

```
public void  testStackEmptiness() {
  Stack stack = new Stack();
  assertTrue("Test emptiness of Stack", stack.isEmpty());
}
```

   This code will not compile because there is no Stack object created yet. You should go ahead to implement Stack and provide *isEmpty()*. To make it simple you can just return constant boolean value true in body of *isEmpty()*.

```
public boolean isEmpty() {
  return true;
}
```

4. **Test 2: Push a single object on the stack and verify that isEmpty is false.**

   Remember to start with test first NOT to create push before you see compilation error or test failure.

```
public void testPushOne() {
  Stack stack = new Stack();
  stack.push("first element");
  assertFalse("Stack has one element, it is not empty",
              stack.isEmpty());
}
```

5. **Test 3: <u>Push</u> a single object, <u>Pop</u> the object, and verify that <u>isEmpty</u> is true.**

   This test introduces a new method called Pop, which returns the topmost element and removes it from the Stack.

```
public void testPop() {
  Stack stack = new Stack();
  stack.push("first element");
  stack.pop();
  assertTrue("Stack has no element after pop",  stack.isEmpty());
}
```

6. **Test 4: <u>Push</u> a single object, remembering what it is; <u>Pop</u> the object, and verify that the two objects are equal.**

```
public void testPushPopContent() {
  Stack stack = new Stack();
  String value = "9001";
  stack.push(value);
  String result = (String) stack.pop();
  assertEquals("The popped up value equals to the pushed one",
               value, result);
}
```

   Please keep in mind that you don't have to have the correct implementation to make test pass. You can always add a little, run the test to see it fail, and rework until it passes the test.

7. **Test 5: <u>Push</u> three objects, remembering what they are; <u>Pop</u> each one, and verify that they are correct.**

   In previous implementation you can simply have one element to make all those tests pass. With this test you will very likely implement an array, ArrayList, or vector to hold objects that are pushed onto the stack.

8. **Test 6: Pop a Stack that has no elements.**

   As you may work on Java for a while, exception should be thrown when there is illegal operation like this one.

```
public void testPopEmptyStack() {
  try {
    stack.pop();
    fail("Exception is expected when pop value from empty stack");
  }
  catch (Exception e) {
    //Do nothing. Exception is expected.
  }
}
```

9. **Test 7: Push a single object and then call Top. Verify that isEmpty returns false.**

```
public void testPushTop() {
  Stack stack = new Stack();
  stack.push("42");
  stack.top();
  assertFalse("Stack is not empty after top() is called.",
              stack.isEmpty());
}
```

10. **Test 8: Push a single object, remembering what it is; and then call Top.**

    Verify that the object returned is equal to the one that was pushed.

11. **Test 9: Push multiple objects, remembering what they are; call Top, and verify that the last item pushed is equal to the one returned by Top.**

12. **Test 10: Push one object and call Top repeatedly, comparing what is returned to what was pushed.**

13. **Test 11: Call Top on a Stack that has no elements.**

14. **Test 12: Push null onto the Stack and verify that isEmpty is false.**

15. **Test 13: Push null onto the Stack, Pop the Stack, and verify that the value returned is null.**

16. **Test 14: Push null onto the Stack, call Top, and verify that the value returned is null.**

We don’t have either instructional code in last 7 test cases. Stack is a simple data structure and TDD does not have high technique requirements you should be able to implement it and make all these tests pass with small amount of effort.

Appendix B

User Stories for Stack Data Structure

A Hands-on Practice of TDD: User Stories of Stack

The objective of this assignment is to practice TDD development with stack problem. User stories are provided to help you develop stack in TDD iteratively. Stack is a data structure that works in Last-In-First-Out principle. It includes four basic operations: Push, Pop, Top, and isEmpty.

- The Push function inserts an integer element onto the top of the Stack.
- The Pop function removes the topmost integer element and returns it.
- The Top operation returns the topmost integer element but does not remove it from the Stack.
- The isEmpty function returns truth when there are no elements on the Stack and false otherwise.

Please note that this assignment is not just about programming a stack data structure. Instead, it is a hands-on practice on Test-Driven Development. You should implement stack iteratively using the following user stories.

1. Create a stack and verify that it is empty
Requirement: Be able to construct a stack which is empty initially. Verify that it is empty.

2. Push an integer value and verify that stack is not empty.
Requirement: Push value 1001 onto the stack, check whether stack is not empty afterward.

3. Push an integer value, pop it, and verify that stack is empty.
Requirement: Push value 1001 onto the stack, call pop, check to make sure that stack is empty.

4. Push an integer value, remember what it is; pop a value from stack, verify that it is equal to the one pushed.
Requirement: Push value 1001 onto the stack, call pop, examine whether the popped value is 1001.

5. Push three integer values, remember what they are; pop each one, and verify that they are correct.
Requirement: Push integer values 1001, 2001, 3001 onto the stack, call pop three times. It should return 3001, 2001 and 1001 respectively.

6. Pop an integer value from stack that is empty.
Requirement: Exception StackEmptyException should be thrown when trying to pop a value from an empty stack.

7. Push an integer value, call top, and verify that the returned value equal to the pushed value.
Requirement: Push value 1001 onto the stack, call top, the returned value should be 1001.

8. Push three integer values, call top three times, and verify the returned values always equal to the last value.
Requirement: Push 1001, 2001, 3001 onto the stack, call top three times, and the returned values should be 3001.

9. Push one integer value, call top repeatedly, comparing what is returned to what was pushed.
Requirement: Push 1001 onto the stack, call top three times, and the returned values should be 1001.

10. Call top on a stack with no element.
Requirement: Exception StackEmptyException should be thrown when trying to top a value from an empty stack.

Appendix C

User Stories for Roman Numeral

A Hands-on Practice of TDD: User Stories of Roman Numeral Conversion

Roman numerals are written as combinations of the seven letters in the Table C.1 (excerpted from URL http://www.yourdictionary.com/crossword/romanums.html). If smaller numbers follow larger numbers, the numbers are added. If a smaller number precedes a larger number, the smaller number is subtracted from the larger. For example:

Table C.1. Roman Numerals

I=1	C=100
V=5	D=500
X=10	M=1000
L=50	

- VIII = 5 + 3 = 8
- IX = 10 - 1 = 9
- XL = 50 - 10 = 40

Table C.2. Roman Numerals Conversion Table

1	**I**	11	**XI**	21	**XXI**	31	**XXXI**	41	**XLI**
2	**II**	12	**XII**	22	**XXII**	32	**XXXII**	42	**XLII**
3	**III**	13	**XIII**	23	**XXIII**	33	**XXXIII**	43	**XLIII**
4	**IV**	14	**XIV**	24	**XXIV**	34	**XXXIV**	44	**XLIV**
5	**V**	15	**XV**	25	**XXV**	35	**XXXV**	45	**XLV**
6	**VI**	16	**XVI**	26	**XXVI**	36	**XXXVI**	46	**XLVI**
7	**VII**	17	**XVII**	27	**XXVII**	37	**XXXVII**	47	**XLVII**
8	**VIII**	18	**XVIII**	28	**XXVIII**	38	**XXXVIII**	48	**XLVIII**
9	**IX**	19	**XIX**	29	**XXIX**	39	**XXXIX**	49	**XLIX**
10	**X**	20	**XX**	30	**XXX**	40	**XL**	50	**L**

Please note that this assignment is not just about programming a roman numerals conversion. Instead, it is a hands-on practice on Test-Driven Development. You should use the provided user stories to write test case first, and let the tests to drive the code implementation.

Roman Numeral Conversion User Stories:

1. The conversion program returns empty string “ ” to value 0.
2. Roman numeral is “I” to value 1.
3. Roman numeral is “II” to value 2
4. Roman numeral is “III” to value 3
5. Roman numeral is “IV” to value 4, not ”IIII”
6. Roman numeral is “V” to value 5
7. Roman numeral is “VI” to value 6
8. Roman numeral is “VIII” to value 8
9. Roman numeral is “IX” to value 9, not VIIII
10. Roman numeral is “X” to value 10
11. Roman numeral is “XI” to value 11
12. Roman numeral is “XV” to value 15
13. Roman numeral is “XIX” to value 19
14. Roman numeral is “XX” to value 20
15. Roman numeral is “XXX” to value 30

Appendix D

Case Study Consent Form

University of Hawai'i at Manoa
Department of Information and Computer Sciences
Collaborative Software Development Laboratory
Professor Philip Johnson, Director
POST Room 307• 1680 East-West Road • Honolulu, HI 96822
Voice: +1 808 956-3489 • Fax: 956-3548
Email: johnson@hawaii.edu

Thank you for agreeing to participate in our research on understanding test-driven development practices using the Zorro tool. This research is being conducted by Hongbing Kou as part of his Ph.D research in Computer Science at the University of Hawaii under the supervision of Professor Philip Johnson.

As part of this research, you will be asked to develop or modify a program using test-driven design practices and the Eclipse IDE using the Hackystat Eclipse sensor. While you are working on your programming task, you will be sending data about how you program, including the statements that you write, the test cases that you develop, the times that you invoke the tests and their outcomes to a remote Hackystat server. You own the development activity data you send to the server, and it shall not be used by anyone for any purpose other than the one stated in this form without your consent.

At the beginning of the study, we are going to survey your opinions on doing test-driven development. Then, you will do test-driven development using the instrumentation of the Hackystat Eclipse sensor, and use the Zorro analysis package to understand your compliance of test-driven development process. Another survey will be conducted after you use Zorro. Your participation is voluntary, and you may decide to stop participation at any time, including after your data has been collected.

The survey data that we collect will be treated strictly confidential, and there will be no identifying information about you in any analysis of this data for all purposes, your data will remain anonymous.

If you have questions regarding this research, you may contact Professor Philip Johnson, Department of Information and Computer Sciences, University of Hawaii, 1680 East-West Road, Honolulu, HI 96822, 808-956-3489. If you have questions or concerns related to your treatment as a research subject, you can contact the University of Hawaii Committee on Human Studies, 2540 Maile Way, Spalding Hall 253, University of Hawaii, Honolulu, HI 96822, 808-539-3955.

Please sign below to indicate that you have read and agreed to these conditions.

Thank you very much!

__________________________________ ______________________________

Your name/signature Date

Cc: A copy of this consent form will be provided to you to keep.

Appendix E

User Stories for Bowling Score Keeper

Test-Driven Development Exercise: Bowling Score Keeper

The objective is to develop an application that can calculate the score of a SINGLE bowling game using TDD. There is no graphic user interface. You work on objects and JUnit test cases only in this assignment. We divide the bowling game requirements into a set of user stories, which can serve as your to-do list. You should be able to come up with a solution without much comprehension of the bowling game rules. We encourage you to solve this programming task using TDD as much as possible.

1. Frame

10 pins are arranged in an equilateral triangle in bowling game. It is called "frame". The goal of a frame is to knock all 10 pins down. The player has two chances, called "throws", to do so.

Requirement: Define frame so that it has two integer attribute values. Each value represents a throw.

Example: [2, 4] is a frame with two throws. Note that you don't have to check parameters.

2. Frame Score

The frame score is the sum of the first throw and second throw. For example, score of frame [3,5] is 8; score of frame[0,0] is 0, which is called "gutter" in bowling game.

Requirement: Compute score of a frame.

Example: The score of frame [2, 6] is 8. Frame [0, 9]'s score is 9.

3. Game

A single bowling game consists of 10 frames.

Requirement: Define bowling game which consists of 10 frames.

Example: A sequence of frames [1,5] [3,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 6] is a game. Note that we will use this game many times from now on. We will modify only a few frames each time to represent different bowling game scenarios.

4. Game Score

The score of a bowling game is the sum of its 10 frames.

Requirement: Compute the score of a bowling game.

Example: The score of above game is 81.

5. Strike

A frame is called "strike" if 10 pins are knocked down by the first throw. In this case, there is no second throw. A strike frame can be written as [10,0]. The score of a strike is 10 plus the following two throws. Suppose there are consecutive frames such as [10, 0] and [3, 6], then the strike frame score will be 10 + 3 + 6 = 19.

Requirement: Compute the score of a bowling game with a strike frame.

Example: Let's suppose the first throw in above game is a strike. The bowling game will have frames [10,0] [3,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 6]. Its score will be 94.

6. Spare

A frame is called "Spare" when 10 pin are knocked down by two throws. For example, [1,9], [4,6], [7,3] are all spares. The score of a spare frame is 10 plus the next throw following it. If you have two frames [1,9] and [3,6] in a row, the spare frame score will be 10 + 3 = 13.

Requirement: Compute the score of a bowling game with a spare frame.

Example: Similarly let's assume the first frame in above game is a spare [1,9], then it will have frames [1,9] [3,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 6]. Its score will be 88.

7. Strike and Spare

A strike frames is followed by a spare frame. For example, [10,0], [4,6], [7, 2] are three consecutive frames with a strike followed by a spare. Score for the strike is 10 + 4 + 6 = 20, and score for the spare is 10 + 7 = 17.

Requirement: Compute the score of a bowling game with a spare frame follows a strike.

Example: Similarly let's assume the first two frames are [10, 0] and [4, 6] in above game. The game will have frames [10,0] [4,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 6]. Its score will be 103.

8. Multiple Strikes

Two strikes in a row is possible in a real bowling game. To three frames [10, 0], [10, 0] and [7,2], score for the first strike will be 10 + 10 + 7 = 27. The second strike score will be 10 + 7 + 2 = 19.

Requirement: Compute the score of a bowling game with two strikes in a row.

Example: Let's assume the first two frames are both strikes, then the bowling game will look like [10,0] [10,0] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 6]. Its score will be 112.

9. Multiple Spares

Two spares in a row is another case.

Requirement: Compute the score of a bowling game with two spares in a row.

Example: Assuming the first two frames are spares, then the bowling game will look like [8,2] [5,5] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 6]. The game score will be 98.

10. Spare as the Last Frame

When the last frame is a SPARE, the player will be given a bonus throw. However, this throw does not belong to a regular frame. It is only used to calculate the score of the last spare.

Requirement: Compute the score of a bowling game when the last frame is a spare.

Example: Assuming the last frame is a spare in above game, then game will be [1,5] [3,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [2, 8] with bonus throw [7]. Its score will be 90.

11. Strike as the Last Frame

When the last frame is a STRIKE, the player will be given two bonus throws. However, these two throws do not belong to a regular frame. They are used to calculate score of the last strike frame only.

Requirement: Compute the score of a bowling game when the last frame is a strike.

Example: Assuming the last frame is a strike in above game, it will be [1,5] [3,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4, 5] [8, 1] [10, 0] with bonus throws [7, 2]. The game score will be 92.

12. Bonus is a strike

Bonus strike will not be counted as strike in a bowling game.

Requirement: Assuming the last frame is a spare and the bonus is a strike, compute the score of this game.

Example: Assuming the last frame is a spare and the bonus is a strike in above game, the game will be [1,5] [3,6] [7,2] [3,6] [4,4] [5,3] [3,3] [4,5] [8,1] [2,8] with bonus throw [10, 0]. The game score will be 93.

13. Best Score

Requirement: Compute the score of the bowling game when all frames are strikes.
Example: Assuming all frames are strikes including bonus. The game looks like [10,0] [10,0] [10,0] [10,0] [10,0] [10,0] [10,0] [10,0] [10,0] [10,0] with bonus throws [10,10]. It is a perfect game and the game score is 300.

14. A Real Game

Requirement: To a game with frames [6,3] [7,1] [8,2] [7,2] [10,0] [6,2] [7,3] [10,0] [8,0] [7,3] [10], its score is 135.

Appendix F

Participant Interview Guideline in Case Study

Purpose

The purpose of this interview is to gather participants' experience of TDD including how they think about TDD, whether and how TDD affects their software development, whether can Zorro help them, and how Zorro can be used? The protocol of the interview is described here.

Interviewer

Hongbing Kou

Interviewees

Participants of the Zorro case study

Time and place

Participants will be interviewed by me in the lab after they finish validating Zorro's inference on their behaviors. The interview will last from 15 to 20 minutes.

Facility

Notepad, pen, and tape recorder. I will ask interviewee's permission for the use of tape recorder.

Outline

- Questions from the participant
- Experiences and opinions on unit testing and Test-Driven Development
- Opinions on TDD measurement with Zorro. In what way does the measurement tool help?
- Zorro usefulness evaluation
- Possible improvements of Zorro

List of interview questions

1. Questions from the participants

   I will give interviewees some time at the beginning to ask me questions. They may ask questions about TDD, Zorro or this study. Purpose of this is to let participants feel comfortable before the interview starts. This may lead them to get involved and start talking.

2. Unit testing and Test-Driven Development

   - When and where did you learn unit testing?
   - How do you apply unit testing in your software development?

     Do you write testing code when you are not confident about a program?
     Do you write testing code after you finish a program?
     Do you write testing code when you want to improve your testing coverage?
     Did you ever write testing code first before you learned TDD?
   - How much testing code do you write?

     How much is the code coverage of the programs you wrote in the software engineering class?
     Can you comment on the use of unit testing in software development?

- Can you compare TDD to the testing strategy you did before?

  How do you think of TDD? Is it helpful to improve software quality?
  How comfortable it is for you to do TDD programming? What problems you have when you programmed in TDD?

3. Please use scale 1 to 5 to assess the usefulness of Zorro's TDD analyses (1 stands for least useful and 5 stands for most useful). I would like you to justify your answers.

   - Episode Inference
   - TDD Episode Demography
   - TDD Episode Duration Distribution's
   - Test Effort vs. Production Effort
   - Test Size vs. Production Size

4. What other information you wish to have about TDD development?

   How about an Eclipse plug-in indicating whether you are doing TDD?
   How about an analysis showing your TDD performance over the time?

Appendix G

Participant Selections of TDD Analysis Usefulness Areas

Table G.1. TDD Analysis Useful Areas

TDD Analysis	Useful Areas	A	K	L	N	O	P	Q	R	S	T
Episode Demography	UA-1	X		X	X	X	X	X		X	X
	UA-2		X		X		X	X	X	X	X
	UA-3			X					X		X
	UA-4				X	X	X	X			
	UA-5		X						X		X
	UA-6						X				X
	UA-7		X				X				
	UA-8			X	X	X	X	X			X
T/P Effort Ratio	UA-1	X		X	X	X	X	X	X	X	X
	UA-2						X			X	X
	UA-3										X
	UA-4	X		X	X	X		X		X	X
	UA-5					X					X
	UA-6						X				X
	UA-7				X		X				
	UA-8	X			X	X	X	X	X	X	
T/P Size Ratio	UA-1	X		X		X	X		X	X	
	UA-2	X					X			X	X
	UA-3				X						X
	UA-4		X	X			X				X
	UA-5								X		
	UA-6	X					X				
	UA-7						X				
	UA-8	X			X	X	X	X		X	
Episode Duration	UA-1	X		X		X	X	X	X	X	
	UA-2	X					X			X	
	UA-3			X							
	UA-4	X	X	X	X		X	X		X	
	UA-5										
	UA-6					X					
	UA-7					X					
	UA-8								X		
Duration Distribution	UA-1	X		X				X	X	X	X
	UA-2	X			X		X		X	X	X
	UA-3					X					X
	UA-4		X	X		X		X			X
	UA-5					X					X
	UA-6	X		X							X
	UA-7										
	UA-8			X		X	X				

Bibliography

[1] Manifesto for agile software development. `<http://www.agilemanifesto.org/>`.

[2] Extreme js – js greenwood's web log on architecture, .net, process, and life ... `http://weblogs.asp.net/jsgreenwood/archive/2004/11/26/270503.aspx`.

[3] David Astels. Test-Driven Development: A Practical Guide. Prentice Hall, Upper Saddle River, NJ, 2003.

[4] Kent Beck. Extreme Programming Explained: Embrace Change. Addison Wesley, Massachusetts, 2000.

[5] Kent Beck. Aim, fire. IEEE Softw., 18(5):87–89, 2001.

[6] Kent Beck. Test-Driven Development by Example. Addison Wesley, Massachusetts, 2003.

[7] Thirumalesh Bhat and Nachiappan 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.

[8] Marc I. Kellner Bill Curtis and Jim Over. Process modeling. Communications of the ACM, 35(9):75–90, 1992.

[9] Susan S. Brilliant and John C. Knight. Empirical research in software engineering: A workshop. Software Engineering Notes vol, 24(3):45–52, 1999.

[10] Jonathan E. Cook. Process Discovery and Validation through Event-Data Analysis. Ph.d thesis, University of Colorado, 1996.

[11] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In ICSE '95: Proceedings of the 17th international conference on Software engineering, pages 73–82, New York, NY, USA, 1995. ACM Press.

[12] John W. Creswell. Research design: qualitative, quantitative, and mixed methods approaches. Sage Publications, Thousand Oaks, California, 2003.

[13] Bob Dick. Grounded theory: a thumbnail sketch. `http://www.scu.edu.au/schools/gcm/ar/arp/grounded.html`.

[14] Anne M. Disney and Philip M. Johnson. Investigating data quality problems in the PSP. In Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT'98), Orlando, FL., November 1998.

[15] Gunjan Doshi. Test-driven development quick reference guide. `http://www.gunjandoshi.com/mtarchives/TestDrivenDevelopmentReferenceGui%de.pdf`.

[16] Gunjan Doshi. Test-driven development rhythm. `http://www.gunjandoshi.com/mtarchives/TDDRhythmReference.pdf`.

[17] Eclipse screen recorder. `http://csdl.ics.hawaii.edu/Tools/Esr/`.

[18] Stephen H. Edwards. 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, pages 26–30. ACM Press, 2004.

[19] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. IEEE Trans. Softw. Eng., 31(3):226–237, 2005.

[20] Extreme programming: A gentle introduction. `<http://www.xprogramming.org/>`.

[21] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya. Results of applying the personal software process. Computer, 30(5):24–31, 1997.

[22] Ernest Friedman-Hill. JESS in Action. Mannig Publications Co., Greenwich, CT, 2003.

[23] Boby George and Laurie Williams. An Initial Investigation of Test-Driven Development in Industry. ACM Sympoium on Applied Computing, 3(1):23, 2003.

[24] Boby George and Laurie Williams. A Structured Experiment of Test-Driven Development. Information & Software Technology, 46(5):337–342, 2004.

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

[26] Client-side configuration: Tool sensor installation. `http://hackystat.ics.hawaii.edu/hackystat/docbook/ch02.html`.

[27] Geir Kjetil Hanssen and Tor Erlend Fægri. Agile customer engagement: a longitudinal qualitative case study. 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.

[28] E. Heierman, G. Youngblood, and D. Cook. Mining temporal sequences to discover interesting patterns. In Proceedings of the 2004 International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, 2004.

[29] Watts S. Humphrey. Pathways to process maturity: The personal software process and team software process. `<http://www.sei.cmu.edu/news-at-sei/features/1999/jun/Background.jun99.%pdf>`.

[30] Andy Hunt and Dave Thomas. Pragmatic Unit Testing in Java with JUnit. The Pragmatic Programmers, 2003.

[31] David Janzen and Hossein Saiedian. Test-driven development:concepts, taxonomy, and future direction. Computer, 38(9):43–50, 2005.

[32] Ron Jeffries. Extreme Programming Installed. Addison Wesley, Upper Saddle River, NJ, 2000.

[33] Chris Jensen and Walt Scacchi. Process modeling across the web information infrastructure. In Special Issue on ProSim 2004, Edinburgh, Scotland, 2004. The Fifth International Workshop on Software Process Simulation and Modeling.

[34] Chris Jensen and Walt Scacchi. Experience in discovering, modeling, and reenacting open source software development processes. In Proceedings of the International Software Process Workshop, 2005.

[35] Philip M. Johnson. Hackystat Framework Home Page. http://www.hackystat.org/.

[36] Philip M. Johnson. Project hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis. Technical report, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, November 2001.

[37] Philip M. Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications, Nashville, TN, December 2001.

[38] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Christopher Chan, Carleton A. Moore, Jitender Miglani, Shenyan Zhen, and William E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In Proceedings of the 2003 International Conference on Software Engineering, Portland, Oregon, May 2003.

[39] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa, and Takuya 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 Symposium on Empirical Software Engineering, Los Angeles, California, August 2004.

[40] Philip M. Johnson, Hongbing Kou, Michael G. Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Improving software development management through software project telemetry. IEEE Software, August 2005.

[41] Philip M. Johnson and Michael 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.

[42] Stanley M. Sutton Jr., Dennis Heimbigner, and Leon J. Osterwell. Appl/a: A language for softwareprocess programming. ACM Transaction on Software Engineering and Methodology, 4(3):221–286, 1995.

[43] Jagadish Kamatar and Will Hayes. An experience report on the personal software process. IEEE Softw., 17(6):85–89, 2000.

[44] Reid Kaufmann and David Janzen. 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, pages 298–299, New York, NY, USA, 2003. ACM Press.

[45] Hongbing Kou and Philip M. Johnson. 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, May 2006.

[46] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. Computer, 36(6):47–56, 2003.

[47] Johannes Link. Unit Testing in Java: How Tests Drive the Code. Morgan Kaufmann Publishers, San Francisco, 2003.

[48] Tdd research findings. `http://weblogs.asp.net/mhawley/archive/2004/04/15/114005.aspx`.

[49] E. Michael Maximilien and Laurie 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.

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

[51] James Newkirk and Alexei A. Vorontsov. Test-Driven Development in Microsoft .NET. Microsoft Press, Seattle, 2004.

[52] Benug workshop on test-driven development. `http://dotnetjunkies.com/WebLog/davidb/archive/2004/09/05/24474.aspx`.

[53] Mataž Pančur, Mojca Ciglarič, Matej Trampuš, and Tone Vidmar. Towards empirical evaluation of test-driven development in a university environment. In Proceedings of EUROCON 2003. IEEE, 2003.

[54] Shari Lawrence Pfleeger. Software Engineering Theory and Practice. Prentice Hall, Upper Saddle River, NJ, 2001.

[55] Lutz Prechelt, Sebastian Jekutsch, and Philip M. Johnson. Actual process: A research program. In Submitted to the 2006 Workshop on Software Process, May 2006.

[56] Roger S. Pressman. Software Engineering: A Practitioner's Approach. McGraw Hill, Boston, 2005.

[57] Fastest developer in the west. `http://mikemason.ca/2003/12/03/#029FastestWayToDevelop`.

[58] Quicktime 7 for windows. `http://www.apple.com/quicktime/win.html`.

[59] Beck testing framework. `<http://www.xprogramming.com/testfram.htm>`.

[60] Hackystat. `http://hackystat.ics.hawaii.edu`.

[61] Test-driven development: Way fewer bugs. `http://www.adaptionsoft.com/tdd.html`.

[62] Tdd explained. `http://homepage.mac.com/keithray/blog/2005/01/16/`.

[63] Test-driven development user group. `http://groups.yahoo.com/group/testdrivendevelopment`.

[64] Test-driven development weblogs. `http://www.testdriven.com/modules/mylinks/viewcat.php?cid=20`.

[65] Your test-driven development community. `http://www.testdriven.com/`.

[66] Test driven development workshop is a go! `http://weblogs.asp.net/rosherove/archive/2004/04/25/119764.aspx`.

[67] Test-driven development with junit workshop. `http://clarkware.com/courses/TDDWithJUnit.html`.

[68] Test-driven development workshop. `http://www.industriallogic.com/catalogs/activities/000002.html`.

[69] Test infected developers anonymous. `http://dotnetjunkies.com/WebLog/seichert/archive/2003/12/03/4214.aspx`.

[70] Microsoft's pilot of tsp yields dramatic results. `<http://www.sei.cmu.edu/publications/news-at-sei/features/2004/2/featur%e-1-2004-2.htm>`.

[71] Ivana Turnu, Marco Melis, and Alessandra Cau. Introducing tdd on a free libre open source software project: a simulation experiment. In QUTE-SWAP Workshop, New York, NY, USA, 2004. ACM Press.

[72] Iserializable - roy osherove's blog. `http://weblogs.asp.net/rosherove/archive/2004/12/02/273833.aspx`.

[73] Unit testing: Can you repeat please? `http://www.methodsandtools.com/dynpoll/oldpoll.php?UnitTest`.

[74] Yihong Wang and Hakan Erdogmus. The role of process measurement in test-driven development. In XP/Agile Universe, pages 32–42, 2004.

[75] Christian Wege. Automated Support for Process Assessment in Test-Driven Development. Ph.d thesis, Eberhard-Karls-Universit at Tubingen, 2004.

[76] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE03), pages 298–299, New York, NY, USA, 2003. ACM Press.

[77] xunit framework. `http://en.wikipedia.org/wiki/XUnit`.

[78] Robert K. Yin. Case Study Research: Design and Methods. Sage Publications, Thousand Oaks, California, 2003.

[79] Qin Zhang. Improving Software Development Process and Product Management with Software Project Telemetry. Ph.D. thesis, University of Hawaii, Department of Information and Computer Sciences, December 2006.

[80] Zorro demo. `http://hackystat.ics.hawaii.edu/hackystat/controller?Key=zorrodemouser&%Command=ZorroDemoHome`.