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

A THESIS PROPOSAL SUBMITTED TO MY THESIS COMMITTEE

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

By
Hongbing Kou

Thesis Committee:

Philip M. Johnson, Chairperson
Daniel Port
David Pager
Kim Binsted

March 29, 2007
Version 1.0.0

Abstract

In my dissertation research, I propose to develop a systematic approach to automatically inferring software development behaviors using a technique I have developed called Software Development Stream Analysis (SDSA). Software Development Stream Analysis is a generic framework for inferring low-level software development behaviors. Zorro is an implementation of SDSA for Test-Driven Development (TDD). In addition, I designed a series of validation studies to test the SDSA framework by evaluating Zorro with respect to its capabilities to infer TDD development behaviors. An early pilot validation study found that Zorro works very well in practice, with Zorro recognizing the software development episodes of TDD with 88.4% accuracy [31]. After this pilot study, I improved Zorro system's inferencing rules and evaluation mechanism as part of my collaborative research with Software Engineering Group at the National Research Council of Canada (NRC-CNRC). I am planning to conduct two more extended validation studies of Zorro in academic and industrial settings for Fall 2006 and Spring 2007.

Table of Contents

List of Figures

List of Tables

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 [37, 38]. 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 [32, 1], in which a phase may last several minutes to a few hours only. Each phase defines how developers and development team should carry on the work on daily basis. The Personal Software Process (PSP) [20] and Extreme Programming (XP) [22, 2, 13] are two examples of a low-level software process. Although proven to be useful in improving software quality[14, 29, 42, 21], low-level software process are hard to execute correctly and repeatedly. In order to improve the quality of practice and research of low-level software process, there must be some supporting tools. In my dissertation research, I focus on one low-level software process, the called Test-Driven Development (TDD) [4], and I developed Zorro software system to study it.

Test-Driven Development (TDD) is an innovative one of the practices of Extreme Programming. In TDD, the software development process is iterative and incremental [32]. There is only one task to accomplish in an iteration. In a particular iteration, a unit test of the task is created first followed by production code implementation. TDD is built on the foundation of the XUnit framework [40], which has been ported to more than 30 languages. Unit testing has become a de facto standard in the software industry. TDD is widely adopted by software professionals. An informal survey [44] conducted by 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.

"Clean code that works"[4] is the goal of Test-Driven Development. To achieve this goal, TDD summarizes its 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 [4]:

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 all the duplication created by merely getting the test to work.

At first glimpse, TDD seems easy, but in fact, it is a very hard and 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 program they develop. Therefore, in a lot of cases, software systems are not designed for easy testing. Consequently, developers often find it is hard for them 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 effort. In TDD, software developers must continuously remain in the mindset of test-first, which is initially counter-intuitive to many of them [3, 45]. So they often apply it differently according to their own experience level and understanding [3].

TDD is gradually becoming a standard well accepted for software development in industry, and yet there are problems in testability and differences in understanding of this methodology. Not surprisingly, the immaturity of TDD causes problems. There are many important research questions regarding software development using TDD. For example, how do we know software developers will faithfully commit to the highly disciplined TDD practice? Will developers slip away from TDD? When does it pay off to use TDD, and when does it not pay off? One thing is clear: these questions cannot be answered accurately without good software process measurement. However, Janzen and Saiedian [21] 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 [21, 44]. Fortunately, as my initial case study demonstrates, measuring the use of certain software development methods is becoming feasible with the emergence of technologies such as the Hackystat system [41, 27, 28, 26], an in-process software metrics collection and analysis framework.

As part of my dissertation research, I developed a software system called Zorro (Figure 1.1) on top of Hackystat to infer TDD development behaviors using low-level software development

activity data collected by Hackystat Eclipse Sensor. Zorro recognizes and evaluates TDD patterns using rule-based system support and the software development stream analysis (SDSA) framework. SDSA is a three-stage analysis technique that brings the Hackystat framework and Zorro system together. First, it merges software development activities and in-process metric data together to create a "software development stream", a sequential stream of low-level software development activities. Second, SDSA includes a tokenization subsystem that divides a single sequential stream of low-level software development activities into collections of events called "software development episodes". Third, the JESS [15] rule-based system recognizes and classifies these episodes according to the classification schema. SDSA binds these three components together to assist the measurement of software development methods and low-level software process.

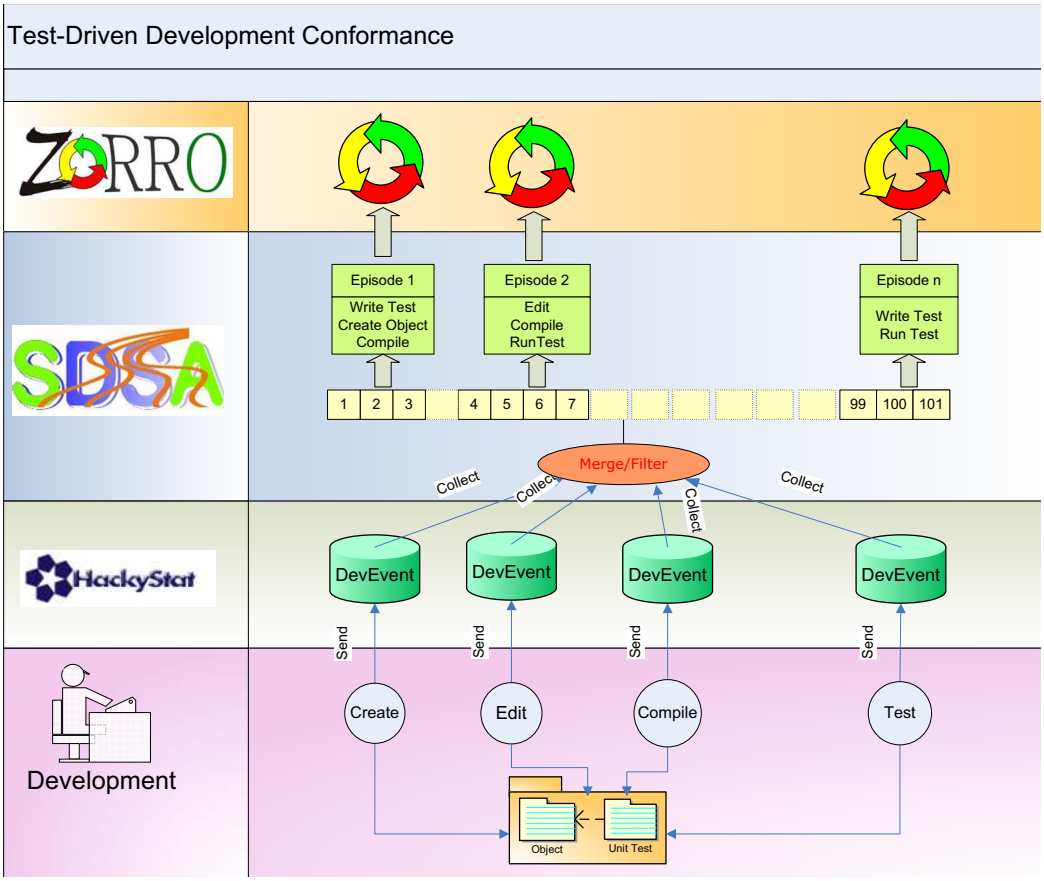


Figure 1.1. Zorro Infrastructure

With the capabilities provided by SDSA, I defined a set of specific rules for TDD in Zorro according to Beck [3, 4] and others who have described the practices of TDD. Zorro uses a

two-step procedure to measure and evaluate the compliance of the developer's behaviors with the practices of TDD. First, Zorro recognizes and classifies the episodes independently according to the classification schema. Second, Zorro evaluates the internal structure as well as the context of the episodes to deduce whether an episode is TDD conformant or not.

Chapter 2

Related Work

Much of the research work on TDD suffers from the threat of "construct validity" [45] 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. Janzen warned that inability to accurately characterize process conformance is harmful to TDD research [21]: Many organizations might be using the methodology without talking about it. Others might claim to be using a methodology when in fact they are misapplying it. Worse yet, they might be advertising its use falsely. Surveys might be conducted to gauge a method's usage, but often only those who are much in favor or much opposed to the methodology will respond.

A handful of research work has been done on software process validation [6, 23] and the process compliance of Test-Driven Development [31, 45, 46]. Cook and Wolf [6] developed a client-server software system called Balboa to do process discovery and validation using a finite state machine (FSM). Balboa collects developers' invocations of Unix commands and CVS commits to learn the software process using FSM and machine learning techniques. Cook was able to reproduce the ISPW 6/7 process with Balboa in his research. However, FSM does not look like an ideal solution for process validation because of the complexity of the process FSM it generates. In his example, the three algorithms RNET, KTAIL and MARKOV generated 15, 20 and 25 states respectively, and the states are interweaved in complicated manners. It is hard to interpret the process state chart without thorough understanding of Balboa and the adopted software process. Jansen and Scacchi [23] simulated an automated approach to discovery and modeling of 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.

Janzen [21] 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. 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."[3] 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 a low-level software process is that it is 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 [31].

Low-level software processes often face similar research questions as other, longer duration software processes. For instance, what process is currently occurring, what process should occur, what are the impacts of a given process on the important outcomes of software such as quality and productivity, and how can a given process be improved and tailored in an organization? So far, software engineering researchers have focused heavily on the important outcomes that TDD brings to software products and software developers. Both pedagogical [34, 11, 18, 36, 12, 30] and industrial [16, 33, 5] evaluations of TDD have been conducted in the last few years. It is interesting to note that number of research studies on TDD in academic settings is greater than the number of research studies in industrial settings.

2.1 Research Work in Academic Settings

Most TDD research studies in academic settings seems to indicate that there is some degree of quality improvement, but that there are little programmer productivity benefits. Indeed, some studies have shown quality improvements but at the cost of decreased productivity.

Muller and Hanger [34] conducted a study in an XP class in Germany to test TDD in isolation of other XP practices 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 [11] 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.

Geras, Smith and Miller [18] also 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.

Pancur [36] 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 [12] used the well-defined test-last and TDD approaches as Pancur did in [36]. 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.

Kaufmann [30]'s pilot study on implications of TDD, in contrast, reported improved software quality and programmers' confidence.

2.2 Research Work in Industrial Settings

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

George and Williams [17] 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 [33] 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).

Another study of TDD at Microsoft conducted by Bhat and Nagappan [5] reported remarkable software quality improvement as measured in number of defects per KLOC. After introducing 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.

2.3 Process Conformance Study of TDD

As we can see from the literature, there are discrepancies in the empirical findings across both educational settings and industrial settings. Sometimes the discrepancies are dramatic, for example [34] found that the TDD group yielded less reliable programs than the control group, while [5] reported that the TDD group improved software quality by over four times.

Wang and Erdogmus [45] pointed out there are several possibilities that might explain why there are the discrepancies in TDD research findings. For example, discrepancies could occur due to differences in populations, differences in teaching methods and materials, and differences in the techniques by which TDD is compared. They argued that TDD empirical software research lacks process conformance, and therefore it suffers from the construct validity problem (as is also the case in some other empirical software engineering research). In [45], 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 use 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 [31], 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 [15].

Similarly, Wege [46] 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.

Chapter 3

Research Questions

The long-term goal of my research is to understand how to characterize and improve low-level software development behaviors. As a step in that direction, I am focusing for my Ph.D. research on a specific kind of low-level software development behavior: Test-Driven Development. The Zorro system, which attempts to infer TDD low-level development behaviors, provides a way to partially evaluate the overall approach and begin to understand its strengths and limitations.

Zorro infers developer's TDD development behaviors using SDSA. It is easy for software developers to collect in-process development activities using Hackystat sensors, and it is also easy for them to evaluate their TDD development behaviors using Zorro. If Zorro's TDD inference is correct, then we can use it to assess TDD process conformance during the daily practice of TDD as well as during empirical studies of TDD. However, does Zorro infer developers' TDD development behaviors correctly? Will it falsely categorize some non-TDD development behaviors as TDD? Or, will it misinterpret some TDD development behaviors as non-TDD? To answer these questions, we need to conduct validation studies of Zorro. Some of the most important research questions are:

- Q1: Can Zorro automate the recognition of Test-Driven Development using automatically collected low-level software development activities?
- Q2: Can Zorro help to improve the practice of TDD?

  This is a hard question, but we can divide it into three small questions with regard to user's roles.

  - For beginners, can Zorro help them improve the compliance to TDD?
  - For experienced TDD practitioners, will Zorro help them improve their TDD practice by analyzing their TDD development behaviors?

– For researchers, can Zorro help them reach legitimate research conclusions on TDD experiments by providing the TDD process conformance information.

Answering these questions requires a “mixed methods” research strategy [7]. Questions Q1 can be investigated by evaluating Zorro’s data collection and TDD inference capability using field observation research method. Investigating question Q2 requires research methods such as collecting users’ feedback or interviewing them. In my research, I designed a series of case studies using these research methods to investigate the research questions I presented above.

Chapter 4

Experiment Design and Analysis

This chapter introduces three Zorro validation studies: a pilot study, a case study with students from the software engineering class as participants, and an external collaborative case study with the TDD community of developers and researchers. Zorro uses low-level software development activity data to infer developer's TDD behaviors. In order to validate its capabilities of data collection and TDD behavior inference, a secondary data source must be used. In my dissertation research, I will introduce two ways to provide the secondary data: recording individual developer's TDD development process using the Eclipse Screen Recorder (ESR) [10]; and gathering developer's feedback to their TDD behavior inference results using the Zorro validation wizard. I have already used the ESR approach in the pilot study. In the second case study, I will plan to use both approaches.

4.1 Zorro Validation Pilot Study

In January 2006, we ran a pilot study at the University of Hawaii in order to assess how well Zorro infers TDD development process using the rule-based system. We found that Zorro accurately recognized participants' TDD behaviors in a simple environment setting.

4.1.1 Purpose of the Study

There were two purposes for this study. One was to test whether Zorro could collect enough development activity data for TDD development behavior inference. The other was to test whether Zorro could recognize the actual TDD development behaviors using rule-based approach.

4.1.2 Research Questions

In the pilot study, I wanted to test the correctness of Zorro's methodology for inferring developer's TDD behaviors. In order to test this, I developed the Eclipse Screen Recorder [10] to do field participant observation. Ad addition, I also wanted to test the capability of ESR to support Zorro validation. The specific research questions for the pilot study were:

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

4.1.3 Research Methodology and Design

Participants

The participants in this pilot study were experienced Java programmers who knew unit testing well. I recruited 7 volunteers who were interested in TDD and were willing to participate this study.

Design and Experimental Manipulation

This study used a pre-experimental design called the one-shot case study [7]. The treatment in this study was TDD. Every participant developed a small program that simulated a stack data structure in Java using the Eclipse IDE and TDD. Before the study started, we introduced the red/green/refactor principle of TDD to the participants if they did not know TDD before. The TDD rhythm [9], TDD quick reference guide [8] and the step-wise stack TDD implementation instructions were three supplemental material to help participates program in TDD. ESR was used in this study to record the development process for participant observation.

Instruments

The IDE for this study is Eclipse. I instrumented participants' TDD development processes with the Hackystat Eclipse Sensor and ESR.

Procedure

1. Setup

   The participants worked on their own computers or on a lab computer we provided. Prior to the study we confirmed that the lab computer had the following software installed:

   - JDK
   - Eclipse IDE
   - Hackystat Eclipse Sensor [19]
   - Eclipse Screen Recorder [10]

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

2. Introduction to TDD

   When participants did not have prior knowledge of TDD, we briefly introduced TDD to them using Beck's simple TDD abstraction: the red/green/refactor order of programming.

3. Development in the Lab or at Home

   Stack is a well-known problem that works according to the Last-In-First-Out (LIFO) principle. Participants in this study developed solutions to the stack problem using TDD method. We provided them with three documents: the graphic illustration of TDD rhythm, the TDD reference guide, and the user stories of stack with TDD implementation instructions at Appendix A.

4.1.4 Data Collection

The Hackystat Eclipse sensor collected and sent development activities to the remote Hackystat server. I collected the programming videos recorded by ESR using memory sticks for study conducted in the lab and email attachments for study conducted by participants themselves at home.

4.1.5 Data Analyses and Results

Inferring Participants' TDD Behavior Inference

The Hackystat Eclipse sensor collected low-level development activities and sent them to a Hackystat server. For each participant, I defined a Hackystat project and inferred their TDD behaviors with Zorro. Figure 4.1 is a Zorro inference report example using my own data. It displays both low-level development activity data used for TDD inference and the inferred results.

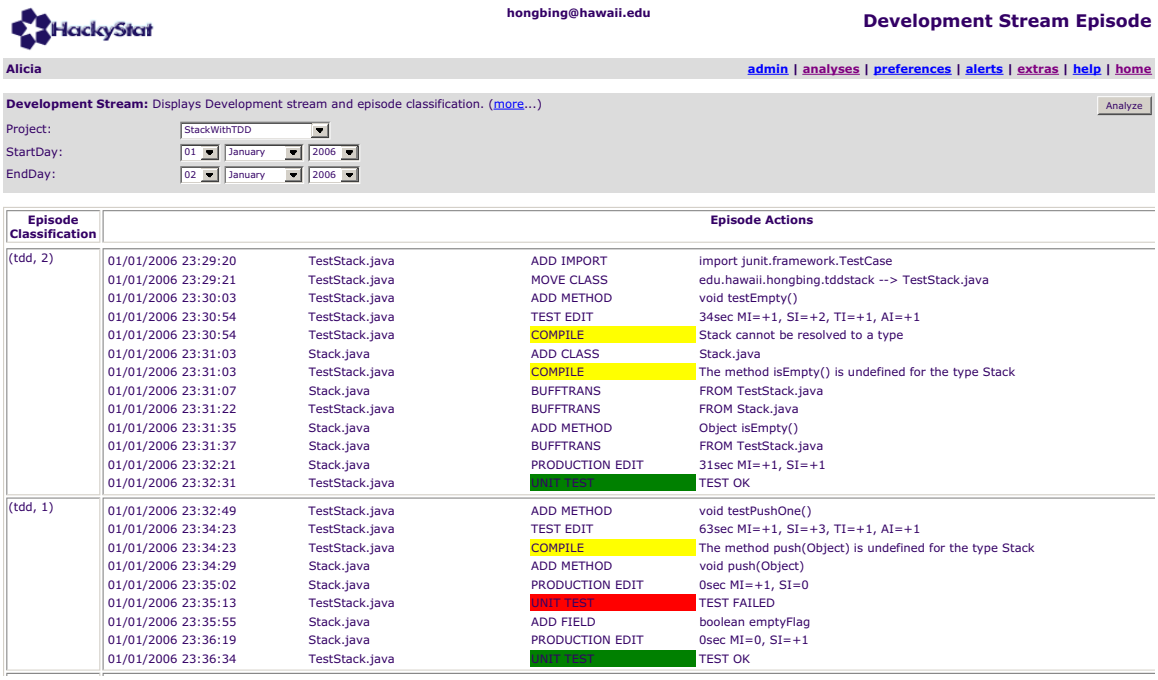


Figure 4.1. Zorro's TDD Behavior Interface Report

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 4.1. Zorro's Inference Result Summary for Pilot Study

Table 4.1 is a brief summary of participants' TDD behaviors inferred by Zorro. They spent 28-45 minutes for this study 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.

Development Process Video Analysis

While participants developed solutions to the stack data structure, they enabled ESR to record the development process as well. Here ESR is the method for field participant observation. It captures the Eclipse screen per second and compress the captured pictures into a QuickTime movie file. Figure 4.2 is a screen dump I made when I played and analyzed a ESR video using the QuickTime Pro software [39].

I used Microsoft Excel for development video annotation analysis. When there was one development activity in the recorded video, I wrote down an entry into Excel. Each entry has the start time, end time, activity abstract, and annotation observed from the video in Figure 4.3.

Validating Zorro's Data Collection

The observed activities from ESR videos in Figure 4.3 were used to validate Zorro's data collection. The comparison between the observed activities using ESR video and activities collected by Zorro allowed us to learn: which activities were missed by Zorro, which activities were not collected correctly, and whether the errors were severe or not (Figure 4.6). In this pilot study, I found 3 types of data collection problems in total:

- **Problem 1**: Edit work is not significant.

Severity:	*High*
Reason:	*Edit work does not change object metrics: number of statements and number of methods, or there is only one state change event occurred for the edit work.*
Result:	*Episodes were misclassified.*
Resolution:	*Change the implementation of file edit sub stream in SDSA to look for file size change as well.*
Affected:	*6 episodes.*

- **Problem 2**: Missing compilation error on test code.

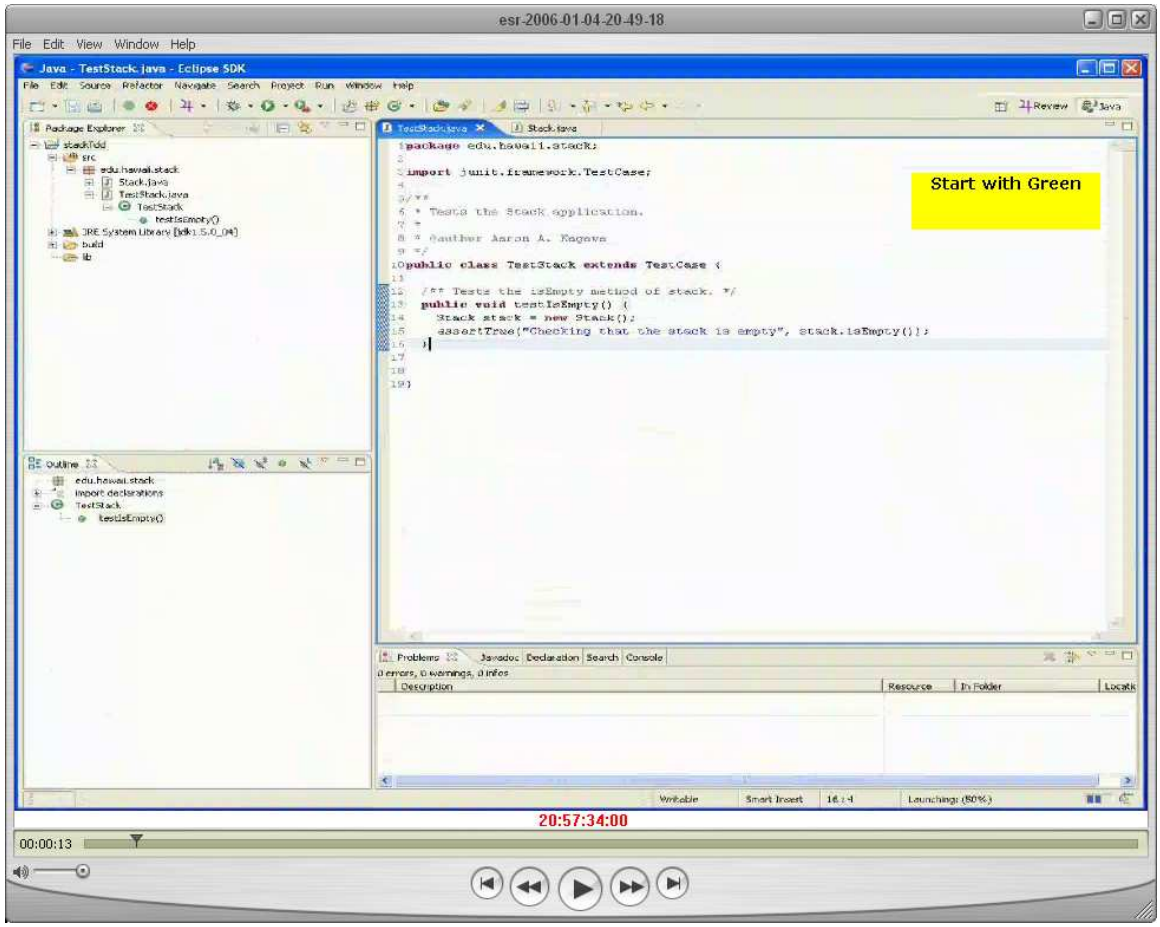
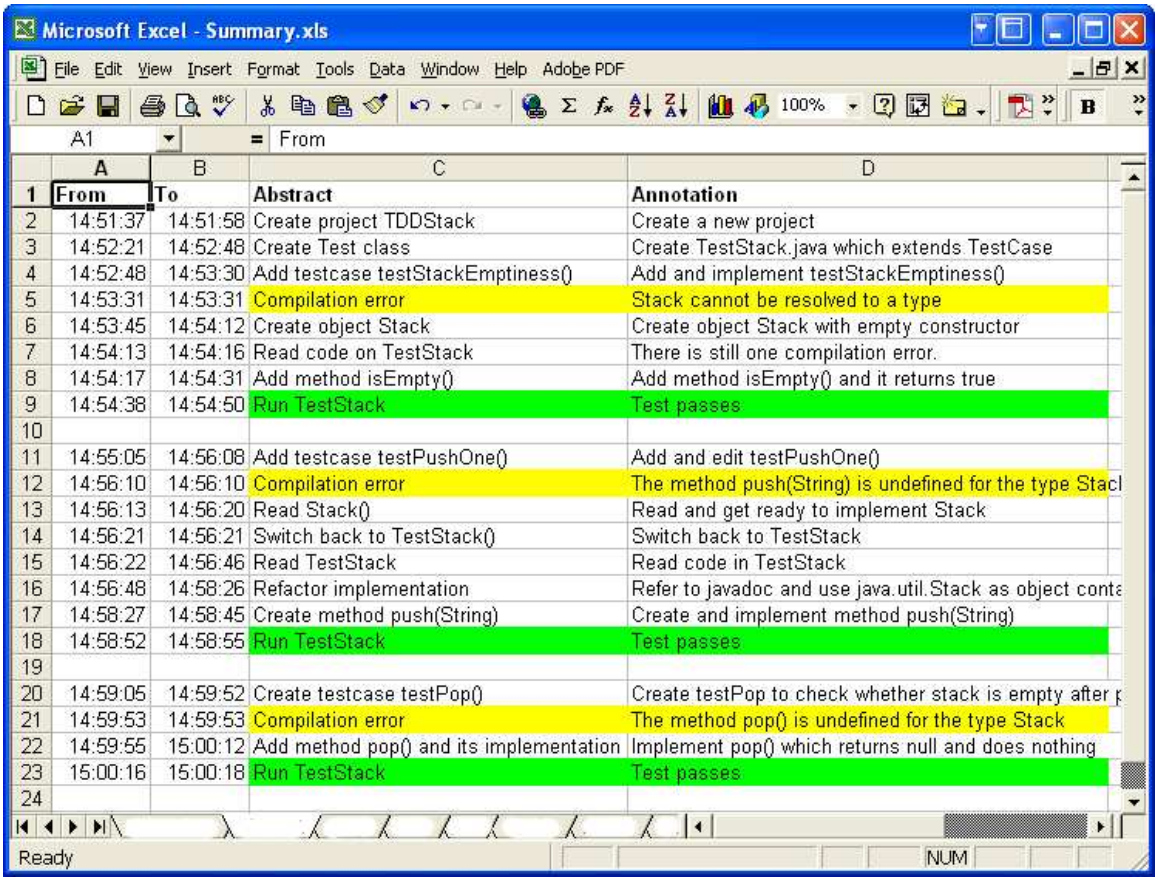


Figure 4.2. Development Process QuickTime Video Recorded by ESR



Microsoft Excel - Summary.xls

File Edit View Insert Format Tools Data Window Help Adobe PDF

A1 = From

	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				

Ready NUM

Figure 4.3. Development Process QuickTime Video Recorded by ESR

Severity: *Low*
Reason: *Changes on production code cause exception on inactive test code.*
Results: *Episode were misclassified.*
Resolution: *Fix Hackystat sensor to report all compilation on inactive file as well.*
Affected: *2 episodes*

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

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

Note that these errors affected 11 episodes in this study.

Validating Zorro's TDD Behavior Inference

ESR was the method we used to observe the participants' behaviors. By playing the recorded movie file, I compared the observed behaviors to the participants' TDD behaviors inferred by Zorro. Table 4.2 lists the comparison results. This manual comparison by human being concluded that 11.6% of the recognized episodes were wrongly inferred by Zorro in this study. It indicates that Zorro infers developer's TDD behaviors correctly 88.4% of the time.

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%

Table 4.2. Validation Result by ESR Video Analysis for Pilot Study

Data collection problems caused most of the inference errors. Infrequent invocation of unit testing by participants was another problem, which yielded episodes with too many activities. Problem 4 describes this type of error.

- **Problem 4**: An episode has 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*

4.1.6 Conclusion and Discussion

Participants in this study spent 28 to 66 minutes on the programming task using TDD. 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 analysis result demonstrates that Zorro has the potential to understand developer's TDD development behaviors automatically using low-level development activities. Using ESR video analysis, we found that there were 3 kinds of data collection problems in Zorro, which affected 11 out of 92 episodes. Overall, it collects enough low-level development activities correctly most of the time for TDD behavior inference. This provides the supporting evidence to research question Q1a. Following this study, I fixed these three data collection problems in the current version of Zorro.

Two out of 93 episodes were incorrectly inferred by Zorro because its inference rules do not work well for long episodes which have too many activities internally. It provides the supporting evidence to research question Q1b. In the current version of Zorro, I improved the inference rule for relatively long episodes, and introduced a new type of episodes which have too many activities or lasts too long a time.

The results from this pilot study indicates that the research method is appropriate. The ESR has the capability to record incremental small changes made by participants. Although ESR caused a small delay when it is initialized, participants did not notice much delay in the development process. With the ESR video, I was able to validate both the Zorro's data collection and inferences of TDD behavior. Thus, there is supporting evidence to research question Q1c. The ESR is an appropriate tool to observe participant's programming behaviors for Zorro validation study.

Overall, Zorro works well in collecting low-level development activities and inferring developer's TDD behaviors in the pilot study. However, one problem with our pilot study is that participants only spent 50% of their development time doing TDD. There are several possibilities

that could explain this phenomenon. One possibility could be that stack is too simple and developers did not need to fail tests first to have the correct implementation. Or it could be that Beck's concise summary of TDD is just too simple. Real TDD development is much more complicated than 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 pattern should be TDD compliant although it is neither test-driven nor refactoring. Therefore, I defined a more sophisticated two-step model to infer TDD development behaviors (Figure 4.4) in this study. First, TDD

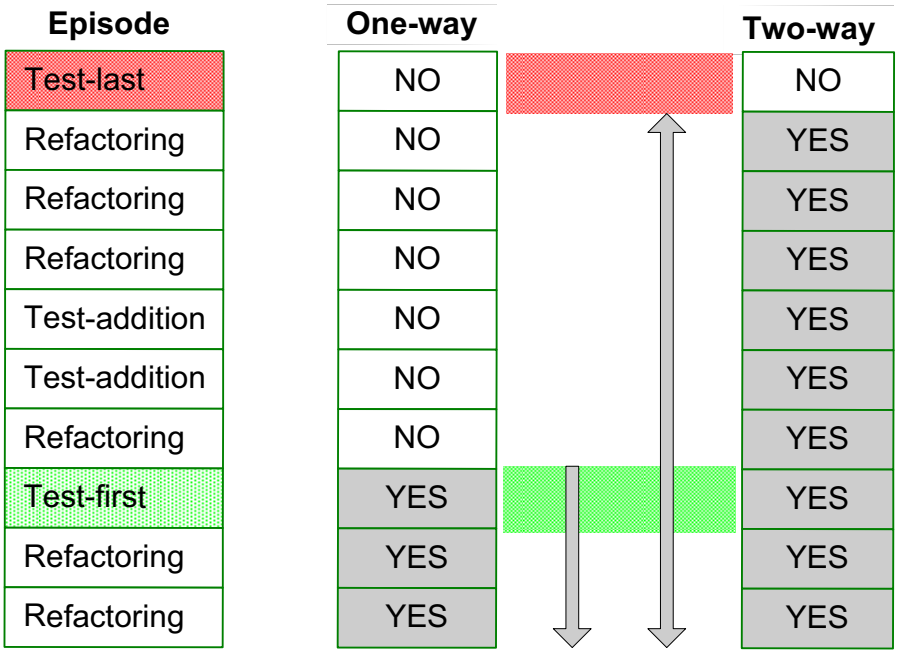


Figure 4.4. TDD Heuristic Algorithms

development episodes are classified independently using internal data. Second, a heuristic algorithm is applied to determine whether an episode is TDD conformant or not. Figure 4.4 has three lists. The left-most one is a list of episodes recognized by Zorro's TDD 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 4.4. 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 4.4 indicates this difference using a single-head arrow and a double-head arrow. Our preliminary work suggests that the two-way heuristic algorithm can understand real world situations better than the one-way algorithm.

4.1.7 Validity Analysis

There were several threats to the validity of this study. One of them is that some participants did not know TDD well prior to the study. Therefore, we provided a graphic illustration of the TDD rhythm [9] and a short list of TDD reference guides [8]. Another threat to validity is that certain applications are hard to test. To minimize the effects of untestability, we used the simple and well-known stack problem in this study. With regard to the validity of data collection, we used unobtrusive data collection utilities: the Hackystat Eclipse Sensor and ESR. Both tools required little overhead from participants [26, 25] at the beginning or end of the study.

There were two valid external validity problems in this study. The first one was the simplicity and stringency of TDD. In the pilot study, we interpreted TDD as strictly as Kent suggested in [3, 4] and Doshi recommended in [9, 8]. The second one was that we only had 7 participants in this study. We hope to address both problems in the future studies.

4.2 Zorro Validation Case Study

The pilot study of Zorro was a success. It convinces us that Zorro's rule-based approach has promise for developer's TDD behavior inference. It also demonstrates that the research methodology works. Following this study, I fixed several data collection problems found in the pilot study. We also improved Zorro's TDD inference rules based on the pilot study and collaboration with Software Engineering Group at the National Research Council of Canada.

In Fall 2006, we plan to conduct a case study of Zorro in a software engineering class at the University of Hawaii.

4.2.1 Purpose of the Study

Currently Zorro collects development activity data more accurately, has a more sophisticated episode classification schema, and infers developer TDD behaviors based not only on the episode's internal structure but also the context in which the episode occurred. The purpose of this study is to:

1. perform Zorro validation study using the Eclipse Screen Recorder;
2. perform a second type of validation in which participates provide feedback through the web-based validation wizard of Zorro;

3. obtain feedback regarding whether Zorro can help TDD beginners through a post-test interview.

4.2.2 Research Questions

In this case study I will test Zorro's abilities to: collect the necessary activity data, infer TDD behaviors correctly, and help beginning TDD learners. The specific research questions for this study are:

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

Note that these research questions support the overall research questions for this thesis as described in Chapter 3.

4.2.3 Research Methodology and Design

Participants

The participants in this study will be students in the software engineering classes at the University of Hawaii during Fall 2006. Unit testing and Test-Driven Development are two skills required by this study. There are 15-16 students in this class and we anticipate that at least a dozen students will participate in this study.

Design and Experimental Manipulation

This study uses mixed research methods[7]. While test subjects work on the bowling game problem using TDD, we will record their development process with ESR[10]. After finishing the TDD programming, participants will launch the analysis validation wizard of Zorro to validate its

TDD behavior inference. Finally, we will interview them. The study will last 2 hours for each test subject including a 90-minute TDD programming session, a 15-minute Zorro evaluation session, and a 15-minute interview.

Instruments

Eclipse is the IDE that will be used. We will instrument participants' TDD development using the Hackystat Eclipse sensor[24] and ESR[10]. Participants will evaluate Zorro's inference of their TDD development using Zorro's web validation wizard. We will also record the participant interview with notepad and tape recorder.

Procedure

Students will learn TDD in the software engineering class and have hands-on practice on TDD programming after the class. After this training, we will request volunteers to participate this case study, and schedule a 2 hour time slot to participate the study in the lab. There, they will do TDD development on the "bowling score keeper" problem (Appendix E) for 90 minutes. Afterwards we will ask them to validate Zorro's inferences of their TDD development. Finally I will interview them for 15 minutes. Below is a more detailed description of this case study procedure.

1. Teaching of TDD

   Instructor of the software engineering class will give a TDD lecture to students. Students will have the first 20 pages of [4] as the reading assignment and a hands-on practice on "Roman Numeral" as the programming assignment.

   The lecture will include the following contents:

   - Introduction to TDD
     - The two principles of TDD from [4]
     - The red/green/refactor pattern of TDD
     - TDD rhythm [9]
     - 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 [5]
  - Test Driven Development proves useful at Google[43]
- 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" [4]
  - TDD Quick Reference [8]
  - Practice TDD on Roman Numeral Problem (Appendix C)

2. TDD Development in the Lab (90 minutes)

   "Bowling score keeper" is a widely used problem for TDD research. I designed user stories for this problem to fit the purpose of this case study research. Participants will develop solutions following the provided user stories (Appendix E). A 90-minute time limit will be enforced. This time frame should be sufficient regardless whether they finish the programming task or not.

3. Zorro's TDD Behavior Inference Validation (15 minutes)

   After participants finish the TDD programming work on the bowling game, they will use the Zorro evaluation wizard to analyze their TDD development and validate Zorro's TDD behavior inference (Figure 4.8).

4. Interview (15 minutes)

   In the end I will interview participants. The purpose of this interview is to learn participant's opinions on unit testing and TDD, discover questions and problems they may have, and investigate whether and how Zorro can help TDD beginners. The interview protocol and outline are available at Appendix F.

Data Collection

Hackystat sensor data and the participants' Zorro evaluations will be stored at the remote Hackystat server. ESR will record the TDD development process into QuickTime movie files in the lab computers. In the interview I will use notepad and tape recorder to record the conversations with participants.

4.2.4 Proposed Data Analyses

Zorro Data Collection Validation

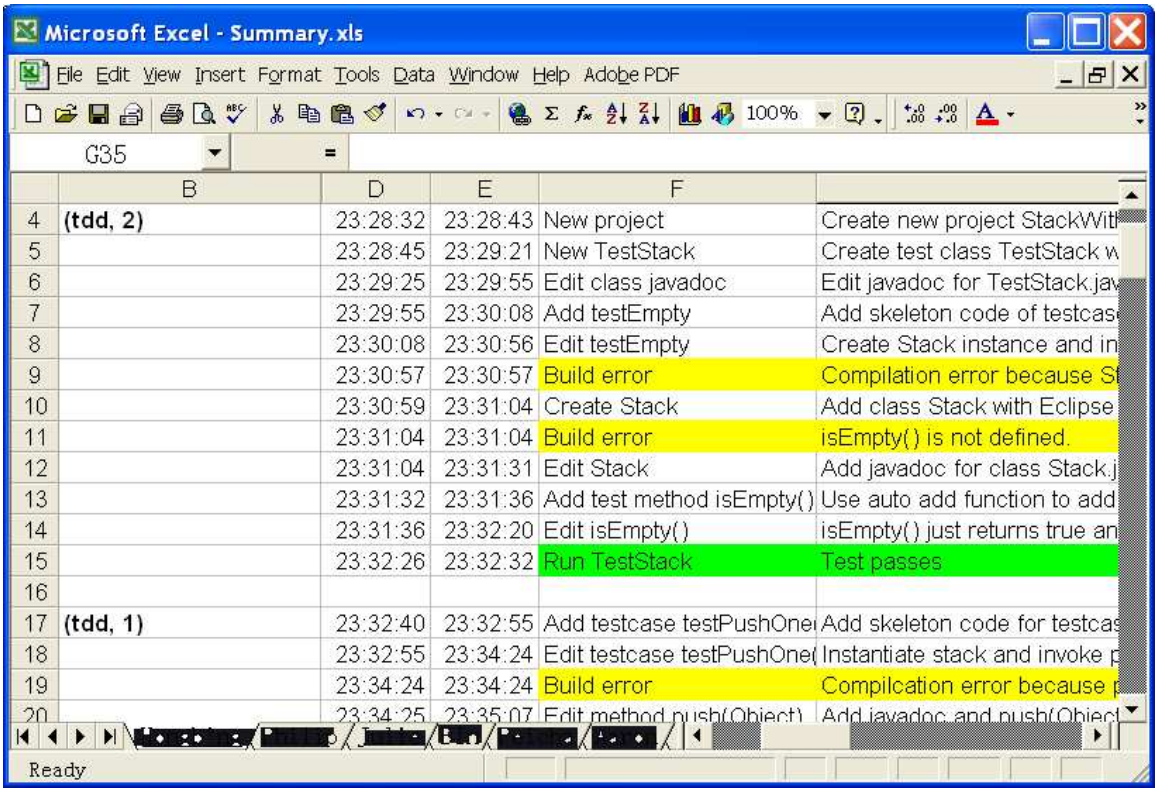
The Hackystat Eclipse sensor collects low-level development activities. These raw sensor data are sent to a Hackystat server. Zorro processes these raw sensor data to perform TDD behavior inference. One purpose of this analysis is to verify that the Hackystat Eclipse sensor can collect enough development activity data, and collect it correctly for TDD developer behavior inference. There are two aspects of this problem. One aspect is whether collected data are accurate, which is research question Q2a. The other aspect is whether the data collection errors will cause episode misclassification, which is related to research questions Q2a and Q2b.

I will use the same analysis method as in the pilot study. First, I will play the development process video recorded by ESR to observe the development activities. Then I will write down the observed development activities into Excel as shown in Figure 4.5.

The observed development activities are used for comparison against the development activities reduced by Zorro (Figure 4.1) from raw sensor data. Figure 4.6 is an excerpt of the comparison of the development activities from these two data sources. Comparing the two sources of data will allow us to verify Zorro's data collection completeness and correctness.

I will use the descriptive analysis to summarize analysis results after comparing the two data sources. For example, assuming there is a problem in collecting unit test invocations, I will present it as follows:

- **Problem**: Two unit test invocations are grouped together.
- **Result**: Two or more episodes can be grouped together so that they cannot be classified correctly.
- **Affected Episodes**: 2



	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 St
10		23:30:59	23:31:04	Create Stack	Add class Stack with Eclipse
11		23:31:04	23:31:04	Build error	isEmpty( ) is not defined.
12		23:31:04	23:31:31	Edit Stack	Add javadoc for class Stack.j
13		23:31:32	23:31:36	Add test method isEmpty( )	Use auto add function to add
14		23:31:36	23:32:20	Edit isEmpty( )	isEmpty( ) just returns true an
15		23:32:26	23:32:32	Run TestStack	Test passes
16					
17	(tdd, 1)	23:32:40	23:32:55	Add testcase testPushOne	Add skeleton code for testcas
18		23:32:55	23:34:24	Edit testcase testPushOne(	Instantiate stack and invoke p
19		23:34:24	23:34:24	Build error	Compilcation error because p
20		23:34:25	23:35:07	Edit method push(Object)	Add javadoc and push(Object

Figure 4.5. Example of ESR Video Script

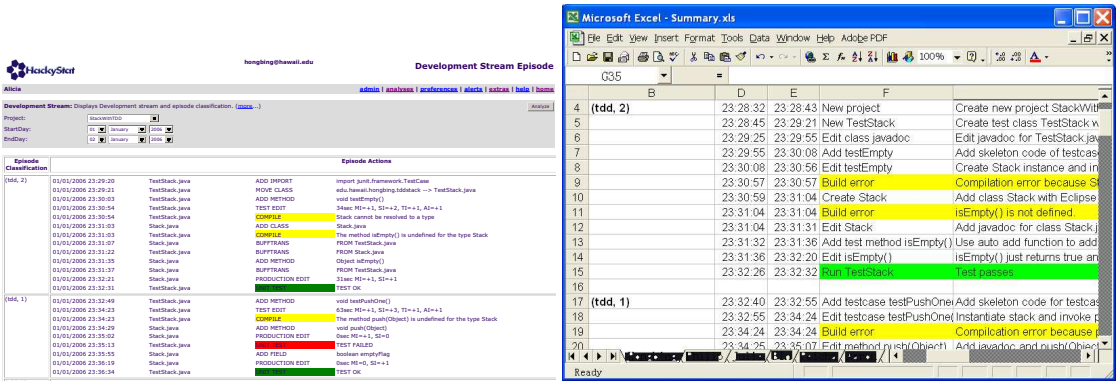


Figure 4.6. Example of Development Activity Comparison between Zorro and ESR

Validating Zorro's TDD Behavior Inference

The purpose of this analysis is to answer research question Q2b, that is, whether Zorro's TDD behavior inference agrees with the observed behaviors of the participants using ESR. ESR video is the method used for participant observation in this study. As in the pilot study, we will use the ESR video to validate Zorro's TDD behavior inference. By playing the movie files produced by ESR, we can observe the participants' development behaviors (Figure 4.7). For example, in the

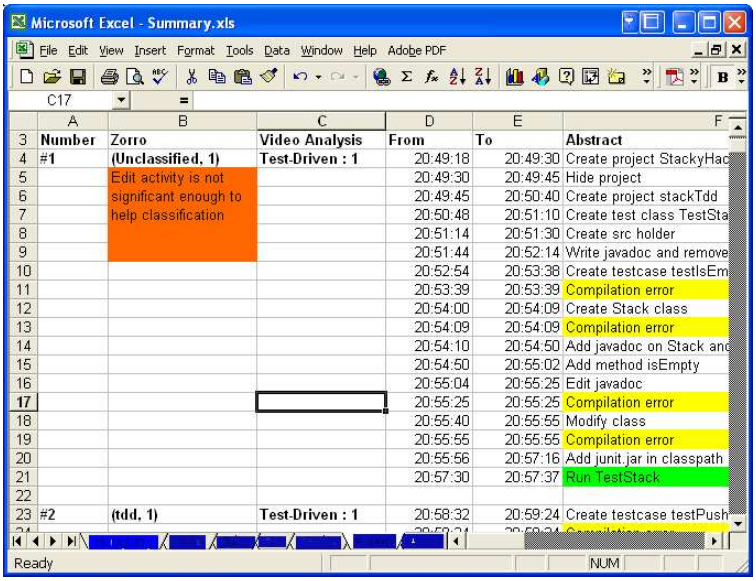


Figure 4.7. Example of Development Behavior Observed via ESR

programming session Figure 4.7, Zorro failed to recognize a legitimate TDD development behavior because the inference rules were insufficient. I will use a table as shown in Table 4.3 to summarize the episode validation results.

Subject ID	Duration	Finished User Stories	Total Episodes	Correctly Recognized Episode	Inference Accuracy
1	44:53	10	15	15	100%
2	28:17	13	20	19	95%
3	48:00	8	14	13	93%
4	66:32	12	20	18	90%
5	43:14	11	22	22	100%
6	45:57	9	15	13	87%

Table 4.3. Example of TDD Episode Validation Results

Using Developer's Feedback as a Second Method for Zorro Validation

TDD is a new practice aiming at "clean code that works". Red/green/refactor is Beck's simple model of TDD; however, it may be too simple for real world situations. For example, experienced TDD developers often write a series of tests that do not require additional production code implementation. In Zorro, I developed a set of rules to infer developer's TDD behavior based on Beck's TDD principle and additional knowledge from TDD practitioners. Therefore, Zorro's TDD inference is somewhat subjective. The purpose of this analysis is to provide additional data from participants to cross-validate Zorro's TDD behavior inference. This effort supplies research question Q2-4, that is, whether participants agree with Zorro's TDD developer behavior inference.

Zorro provides an episode validation analysis for users. This analysis presents Zorro's TDD behavior inference and the underlying reasoning process. It provides three choices for participants to indicate whether they agree or not with Zorro's inference on their TDD development behaviors. In the same analysis, they can also use a set of check-boxes and a text-box to provide additional information about their actual development behaviors (Figure 4.8).

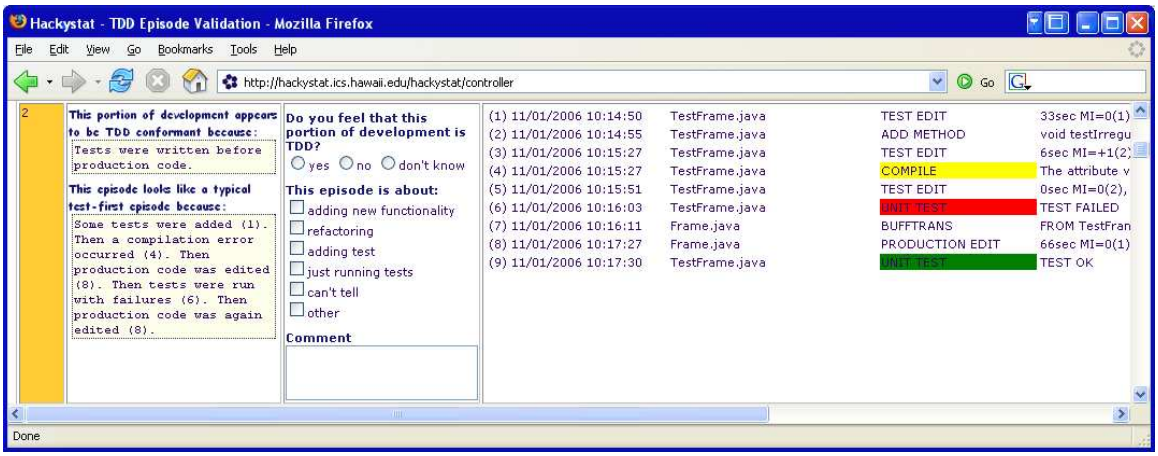


Figure 4.8. Episode Feedback

This analysis cross-validate the TDD behavior validation analysis using ESR video. I will use tables 4.4 and 4.5 to report the analysis results. I will employ categorization and description to interpret the research findings. Tables 4.4 and 4.5 illustrate the summary of this analysis.

Subject ID	Episodes	TDD Episodes	Episodes agreed	Episodes disagreed	Unsure
1	20	18	16	1	1
2	14	14	14	0	0
3	19	15	15	0	0
4	24	20	19	0	1
5	22	22	22	0	0

Table 4.4. Example of TDD Episode Feedback Summary

Subject ID	Episodes	Non-TDD Episodes	Episodes agreed	Episodes disagreed	Unsure
1	20	2	1	0	1
2	14	0	0	0	0
3	19	4	3	1	0
4	24	4	4	0	0
5	22	0	0	0	0

Table 4.5. Example of Non-TDD Episode Feedback Summary

Analysis of Participant Interviews

The purpose of this analysis is to answer research question Q2-4, that is, whether Zorro provides useful information for TDD beginners. I will use the interview research method to collect data about: participant's opinions on unit testing and TDD, Zorro's usefulness, and whether Zorro is helpful for TDD beginners. I will put participants in two categories according to their opinions on unit testing: developers who are strongly in favor of unit testing for high quality software, and those who are not. Since TDD depends on unit testing, this categorization will help us understand TDD beginners' needs better.

In the interview, I will ask participants to evaluate the usefulness of Zorro's 5 TDD analyses. If an analysis is useful, then I will ask what it can be used for. I will use the pattern matching analytic technique to summarize the interview data. For example, participants who are enthusiastic about TDD improvement may find the "Zorro Demography Analysis" to be very helpful for them. The participants who do not buy into TDD may only want to know whether their manager will be okay with their TDD performance if it is required.

4.3 External Case Study

The pilot study and case study are the foundations of this research for evaluating the automation of TDD behavior inference. This last study complements the previous studies by gathering feedback from the community of TDD practitioners and researchers.

4.3.1 Purpose of the study

The first two studies tested the capabilities of Zorro's TDD behavior inference in laboratory environments. The purpose of this study is to:

- validate Zorro's rule-based inference of developer's TDD behaviors;
- investigate Zorro's uses for TDD learning, improvement, and research.

4.3.2 Research Questions

The specific questions for this research are:

- Question Q3a: Does Zorro infer the TDD behaviors correctly as participants' perception?
- Question Q3b: Are Zorro's TDD analyses useful for participants?
- Question Q3c: How can Zorro be used to assist TDD learning, improvement, or research?

4.3.3 Research Methodology and Design

Participants

The participants of this study will be TDD learners, practitioners, and researchers from the TDD community. We will solicit participation from the TDD community using email and news group.

Design and Experimental Manipulation

This external case study will use the one-shot case study research method. Zorro will be the treatment of this study. We will collaborate with participants evaluating Zorro in their environments. We will interview the participants to collect data.

Procedure

1. Zorro Demo Implementation

   As a first step, I implemented a demonstration wizard of Zorro showing the capabilities of Zorro [47]. This application demonstrates 5 analyses Zorro provides, each comes with the introduction and interesting findings. The demo also provides a feedback page for viewers to reach us.

2. Participation Invitation

   We will disseminate an email with the description and purpose of this study to the community of empirical software researchers and XP practitioners. The future actions will depend on what feedback we will get.

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 [8] and the rhythm of TDD [9] to help you do TDD programming.

A.1.1 TDD Quick Reference

(Picture of Gunjan Doshi's TDD quick reference guide [8].)

A.1.2 Rhythm of TDD

(Picture of Gunjan Doshi's TDD rhythm guide [9].)

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 [35].

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" [4]:

   $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: *Push* a single object, *Pop* the object, and verify that *isEmpty* 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: *Push* a single object, remembering what it is; *Pop* 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: *Push* three objects, remembering what they are; *Pop* 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

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

Table C.1. Roman Numerals

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:

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

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**

Table C.2. Roman Numerals Conversion Table

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

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				

Table G.1. TDD Analysis Useful Areas

Bibliography

[1] Manifesto for agile software development. <http://www.agilemanifesto.org/>.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, Massachusetts, 2000.

[3] Kent Beck. Aim, fire. *IEEE Softw.*, 18(5):87–89, 2001.

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

[5] 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.

[6] 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.

[7] John W. Creswell. *Research design: qualitative, quantitative, and mixed methods approaches*. Sage Publications, Thousand Oaks, California, 2003.

[8] Gunjan Doshi. Test-driven development quick reference guide. http://www.gunjandoshi.com/mtarchives/TestDrivenDevelopmentReferenceGui%de.pdf.

[9] Gunjan Doshi. Test-driven development rhythm. http://www.gunjandoshi.com/mtarchives/TDDRhythmReference.pdf.

[10] Eclipse screen recorder. http://csdl.ics.hawaii.edu/Tools/Esr/.

[11] 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.

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

[13] Extreme programming: A gentle introduction. <http://www.xprogramming.org/>.

[14] 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.

[15] Ernest Friedman-Hill. *JESS in Action*. Mannig Publications Co., Greenwich, CT, 2003.

[16] Boby George and Laurie Williams. An Initial Investigation of Test-Driven Development in Industry. *ACM Sympoium on Applied Computing*, 3(1):23, 2003.

[17] Boby George and Laurie Williams. A Structured Experiment of Test-Driven Development. *Information & Software Technology*, 46(5):337–342, 2004.

[18] 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.

[19] Client-side configuration: Tool sensor installation. http://hackystat.ics.hawaii.edu/hackystat/docbook/ch02.html.

[20] 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>.

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

[22] Ron Jeffries. *Extreme Programming Installed*. Addison Wesley, Upper Saddle River, NJ, 2000.

[23] 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.

[24] Philip M. Johnson. Client-side configuration: Tool sensor installation. <http://hackydev.ics.hawaii.edu/hackyDevSite/external/docbook/ch02.html%>.

[25] Philip M. Johnson. Hackystat Framework Home Page. http://www.hackystat.org/.

[26] 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.

[27] 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.

[28] 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.

[29] Jagadish Kamatar and Will Hayes. An experience report on the personal software process. *IEEE Softw.*, 17(6):85–89, 2000.

[30] 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.

[31] 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.

[32] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003.

[33] 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.

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

[35] James Newkirk and Alexei A. Vorontsov. *Test-Driven Development in Microsoft .NET*. Microsoft Press, Seattle, 2004.

[36] Matjaz Pancur and Mojca Ciglaric. Towards empirical evaluation of test-driven development in a university environment. In *Proceedings of EUROCON 2003*. IEEE, 2003.

[37] Shari Lawrence Pfleeger. *Software Engineering Theory and Practice*. Prentice Hall, Upper Saddle River, NJ, 2001.

[38] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, Boston, 2005.

[39] Quicktime 7 for windows. `http://www.apple.com/quicktime/win.html`.

[40] Beck testing framework. `<http://www.xprogramming.com/testfram.htm>`.

[41] Hackystat. `http://hackystat.ics.hawaii.edu`.

[42] 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>`.

[43] Iserializable - roy osherove's blog. `http://weblogs.asp.net/rosherove/archive/2004/12/02/273833.aspx`.

[44] Unit testing: Can you repeat please? `http://www.methodsandtools.com/dynpoll/oldpoll.php?UnitTest`.

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

[46] Christian Wege. *Automated Support for Process Assessment in Test-Driven Development*. Ph.d thesis, Eberhard-Karls-Universit at Tubingen, 2004.

[47] Zorro demo. `http://hackystat.ics.hawaii.edu/hackystat/controller?Key=zorrodemouser&%Command=ZorroDemoHome`.