AUTOMATIC RECOGNITION OF 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

August 6, 2006
Version 1.0.0

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Test-Driven Development(TDD) [5, 3], a newly emerged evolutionary software development method, has been accepted by more and more software developers and intensive research focus has been paid on it by empirical software engineering researchers. It is said that TDD helps to have better requirement understanding, reduce debugging effort, improve productivity, yield high quality code and promote simple design[5, 14, 31, 26]. Website `testdriven.com` and user group [53] are dedicated to test-driven development community, many tutorial workshops[57, 59, 48, 63] and weblogs[54, 58, 50, 66, 40] provide hands-on practice of test-driven development. Modern IDEs Eclipse, Microsoft visual studio team system, and IDEA already have built-in support for test-driven development.

Controversially, many benefits of TDD are not backed up by empirical research [35, 60, 15]. It is a very interesting phenomenon that empirical evaluations on tdd are so divided on the fact that test-driven development is such a plain simple best practice. Moreover, Kent Beck, the pioneer of TDD, already articulated process of TDD very well and demonstrated it with Money example in book “Test-Driven Development by Example”[5]. The persuasive and sound explanations would be that developers in empirical study did not do test-driven development, or they did not do it consistently as researchers expected due to lack of discipline. This is not a hard science problem but a plain fact because test-driven development radically changes development “upside down”[61] by moving unit testing ahead of implementation, which is very counter-intuitive to software developers. A developer tries TDD on a simple problem, he/she thinks TDD is good, and then he/she forgets it in the future software development and returns to the old way to develop software. This story repeats many times already in the practice of TDD among software developers.

Disciplined test-driven development process is desired in order to conduct substantial and conclusive empirical research on it. A common problem of the finished empirical studies on TDD is that there is no better way than verbal confirmation and participation observation to ensure the necessary discipline of test-driven development. It will be very interesting and valuable to have a tool that helps empirical researcher inspect how well developers do test-driven development automatically and unbiasedly. In the course of my doctorial research I conjectured, designed and implemented Zorro software system that can recognize and evaluate test-driven development automatically.

1.2 Test-Driven Development

1.2.1 Introduction

Test-driven development (TDD) is emerging as one of the most successful developer productivity enhancing techniques to be recently discovered. The three-step: write test, write code, refactor - is a dance many of us are enjoying.
- Eric Vautier, David Vydra from testdriven.com

Test-driven development(TDD), also known as Test-first design(TFD), is a new way to develop software in which development is driven by automated unit tests. As its name suggests, TDD radically changes software development from top-down, design documentation oriented approach to bottom-up, requirement oriented approach. It has two fundamental principles[5]:

- *Write new code only if an automated test has failed.*
- *Eliminate duplication*

Test-driven development is an incremental and iterative development (IID) method[29]. Each iteration lasts several minutes and it is rarely longer than 10 minutes. In the beginning of an iteration, developer writes the automated unit test based on requirement. This newly created unit test may not be able to compile or its execution will fail because it tests a feature that is not implemented yet. Failed unit test drives implementation of production code to make it pass. With TDD, a feature is always accompanied by unit test to guard implementation correctness. The hidden philosophy behind TDD principles is that developer only writes enough code to make test pass. If new code adds redundancy developer should refactor it to remove duplication and keep all tests pass. Wake[65] uses metaphor stop-light to abstract pattern of TDD style software development since the unit testing framework xUnit uses red bar to indicate test failure or error and green bar to indicate test success and the iterative development manner of TDD works similar as traffic light.

1.2.2 Related work

"Clean code that works" is Ron Jeffries's pithy phrase to the goal of Test-Driven Development[5]. Since code is driven by automated tests, system developed in this manner will have 100% coverage and developers will have incredible courage to refactor it fearlessly. This reward is so great that "most people learn TDD find that their programming practice changed for good"[5]. This shift is coined as "test infected" by Erich Gamma. The test in TDD is unit test, a.k.a component test. Early practice of TDD helped to create xUnit framework, the de facto standard of unit testing. It was originally designed for SmallTalk and is already expanded to 35 languages support according to the list on Wikipedia[45].

Many TDD advocators appraise TDD because they think TDD helps them improve productivity and code quality; however, the research conclusions on TDD are very divided. George and Williams's study got positive results on TDD. They conducted a structured TDD experiment and compared TDD group who developed in test-driven development and controlled group who developed in waterfall-like process [14]. Their study found that TDD group yielded superior external code quality compared to controlled group, but TDD group took 16% more time on the development. It is quite interesting that most TDD developers thought that TDD is effective and improves their productivity (80% and 78% respectively) in the follow-up survey. Another study conducted by Maximilien and Williams also reported code quality increase. Developers reimplemented a legacy software system with TDD in this study. They found that the new system reduced defect density by 50% in functional verification test(FVT) compared to the legacy system that was developed in ad-hoc manner[31]. There was somewhat productivity decrease in the experiment but developers inclined to continue using TDD in their future software development.

The study conducted by Geras et. al. did not see significant increases on either productivity and code quality. They designed an experiment to ask participants work on two projects with Test-First and Test-Last in different order in two groups [15]. Development effort, tests per KLOC, customer test invocations and unplanned failures after delivery were compared between two groups and there was only slight difference between Test-First and Test-Last processes in both groups. Counter result was reported by Müller and Hagner[35]. In their study, TDD group and controlled group worked on a same graphic library and they verbally confirmed that Test-First subjects got along with Test-First process. They found that TDD does not accelerate the implementation and the resulting programs are not more reliable except that it supports better program understanding.

1.2.3 Problem statement

Researchers realized that TDD is hard to be done consistently by test subjects and they tried all they can do to guard TDD discipline in their experiments. Pair-programming, management, script guideline and verbal confirmation were ever used in the research to help test subjects do TDD in the reported empirical studies. The problem is that none of these measures is efficient enough. Navigator in pair-programming can review the code constantly and insist driver staying on the track of test-driven development, but navigator is not always available and many people still cannot do well on pair-programming. Script guideline and verbal confirmation are not reliable because developers can go other way around such as adding unit tests after production code or neglecting unit tests sometimes. Management of test-driven development is viable but it can only reach a certain level due to the fact that TDD is iterative and is constructed of many short-duration activities.

1.3 Research statement

On one hand, TDD is getting more popular among software developers and intensive efforts have been taken to improve practice of TDD by professionals and researchers[5, 55, 53]. On another hand, TDD changes software development “upside down” [61] and is very counter-intuitive to software developers’ problem solving mind-set nature. The result is that it is hard for developers to do TDD style development consistently and researchers have hard time to conduct conclusive empirical evaluation on TDD, not even to mention process improvement.

Test-driven development is a low-level software process that consists of many short duration activities. This nature determines that traditional software process management technique can not be applied directly. The discipline of low-level software process in general and test-driven development in particular will largely depend on developer’s self control.

Measuring and assessing low-level software process become possible with the emergence of Hackystat[19], a framework for collecting various kinds of software metrics and developer behaviors. My research attempts to solve the discipline problems exist in empirical research and practice of test-driven development with the help from Hackystat. Meanwhile, I generalize this approach to Software Development Stream Analysis (SDSA) framework to assist empirical research and practice of other low-level software processes.

1.4 Proposed Solution: Zorro Software System

The system I develop to assess TDD and improve empirical use of TDD is called Zorro, which is a multi-layer software system including Hackystat, Software Development Stream Analysis(SDSA), TDD specific rules, and applications (Figure 1.1). Layer 1 is Hackystat that collects

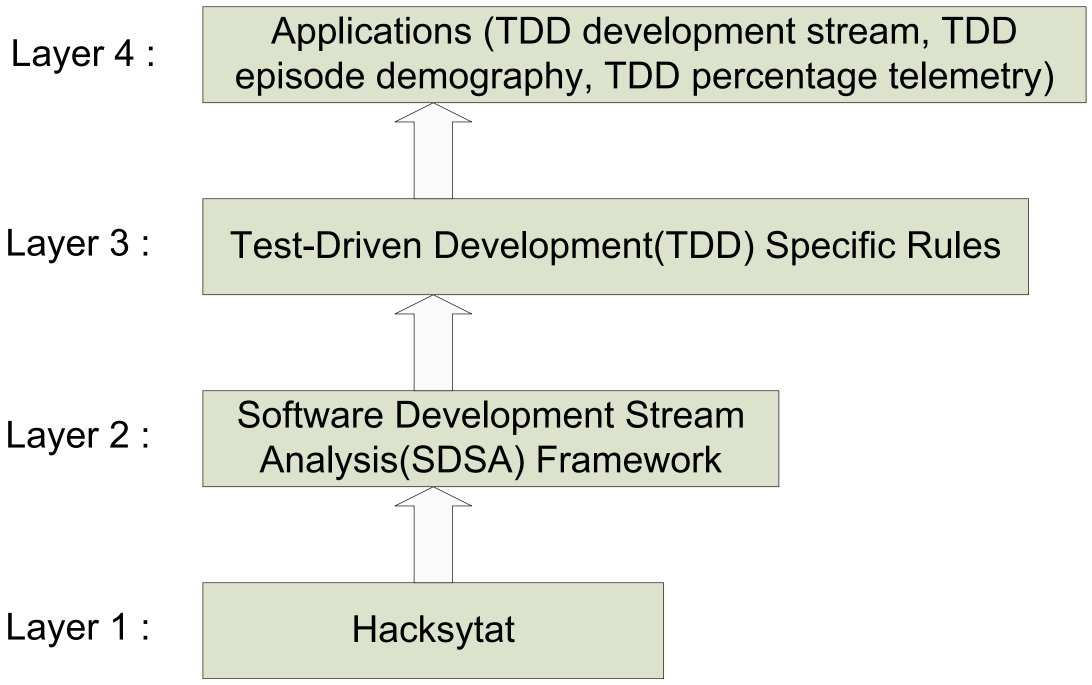

Layer 4 :
Applications (TDD development stream, TDD episode demography, TDD percentage telemetry)
Layer 3 :
Test-Driven Development(TDD) Specific Rules
Layer 2 :
Software Development Stream Analysis(SDSA) Framework
Layer 1 :
Hacksytat

Figure 1.1. Zorro architecture layers

product metric and developer behavior data to be used by upper layers. It also deals with data sending, storage and encapsulation as data collector. Layer 2 is SDSA framework, which builds software development stream out of event data and tokenizes development stream into small episodes, the smallest unit in iterative software development process in which there is a clear goal and are a series of development activities. SDSA incorporates JESS[12] rule-based system to recognize episodes derived from development stream. Specific rules of TDD are supplied in layer 3 to recognize and classify episodes of test-driven development tokenized by test-pass tokenizer. And the fouth layer is application layer which includes TDD development stream, TDD episode demography, and percentage telemetry of TDD to assist discipline study and evaluation of test-driven development.

1.4.1 Hackystat

Hackystat[19] is an open-source framework developed in Collaborative Software Development Lab(CSDL) at University of Hawaii for automated collection and analysis of software product and process metrics and empirical software engineering experimentation. Hackystat sensors are small programs plugged into the development environment to collect software product and process metrics unobtrusively. Hackystat also supplies seamless data sending, saving and retrieving. A set of analyses on product and process metrics of software are defined to support evaluations on various aspects of software engineering. Zorro analysis on test-driven development compliance extends Hackystat process metric analysis into low-level software process domain.

1.4.2 Software development stream analysis framework

Software development stream analysis (SDSA) is a generic framework to analyze low-level software process that are acted by development activities such as file editing, compilation, testing and debugging. SDSA retrieves low-level development activities collected by Hackystat sensors, constructs development stream by serializing these activities, defines a set of tokenizers to organize related activities into episodes and implements an interface to recognize episodes with rule-based system support.

SDSA is a component-based system and is highly configurable. End users of SDSA can selectively add sub stream, the unified development stream with one single type of activities, into the main development stream. It defines interface to plug in different episode tokenizing mechanisms and allows end users define episode recognition rules by themselves.

1.4.3 Recognition rules of Test-Driven Development

Test-driven development is an iterative and incremental method, each iteration consists of three portions red/green/refactor. Zorro instantiates SDSA framework and selects test-pass tokenizer to have software development episodes that end with successful unit test execution(s). A TDD iteration will be separated into two portions red/green and refactoring, both of which end with successful unit tests invocation. An episode is refactoring if there is no new test and it is red/green if there is new test added. A red/green episode is test-driven if and only if development is driven by unit test. So it can be further divided as test-driven in which unit test is created to drive product code implementation, and test-last in which unit test is created after the production code implementation. Complete classification tree of test-pass episodes is depicted in Figure 1.2.

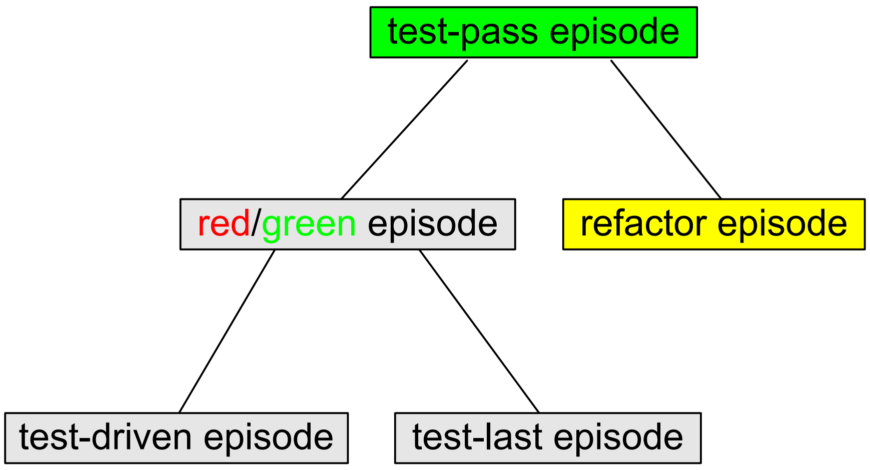


Figure 1.2. Zorro test-pass episode classification tree

1.4.4 Zorro applications

Zorro has a very simple interface that displays software development stream of TDD in episodes 3.4. Demography of TDD episodes is another application that illustrate distrubution of Zorro episode categories in blocks, pie chart and box-whisker chart.

Software telemetry[23] is an approach to manage software development and it provides project members and manager insights useful for local, in-process decision-making. Zorro implements telemetry reduction function to assist parallel correlation between test-driven development and other attributes such as code coverage, active time, or other software metrics. By combining low-level software process detection and software telemetry, we will have a very powerful mechanism to assit evidence-based software engineering [27].

1.4.5 Contribution of Zorro

Test-Driven Development has been claimed to naturally generate 100% coverage, improve refactoring, provide useful executable documentation, produce higher code quality and reduce defect rates[5, 14, 31]. The multi-layer structure I propose can automate collection of developer behavior data and test-driven development recognition. If it recognizes TDD correctly, then we would have a powerful mechanism for exploring how test-driven development is used in practice and its effect on software quality and productivity. The non-intrusive nature of data collection provided by Hackystat will make it very easy to deploy Zorro in both classroom and industrial settings to study

TDD compliance (management), potential discovery of alternative processes and investigation of impacts of TDD.

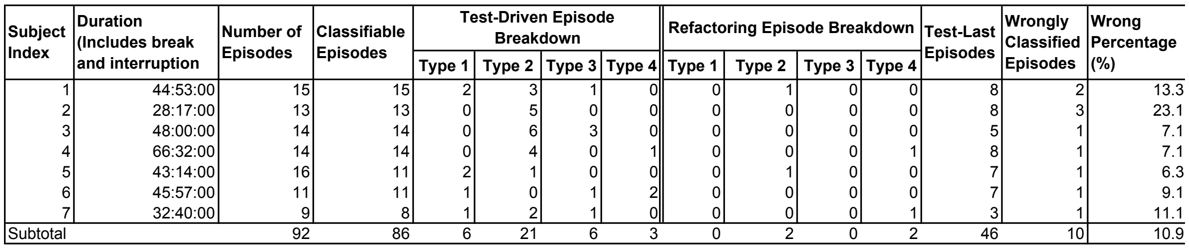
1.5 Zorro validation studies

Before Zorro can be used in the practice and empirical research of TDD, it must be validated to sustain that (1) it can collect the software metrics and developer behavior data necessary to determine TDD, and (2) it can recognize TDD correctly with the collected data. A pilot validation study on Zorro was conducted in University of Hawaii in spring 2006. An extended validation study in classroom setting and another validation study with experienced TDD developers are scheduled in fall 2006.

1.5.1 Pilot validation study

A pilot validation study of Zorro was conducted in University of Hawaii in spring 2006. We recruited 7 experienced java developers and asked them implement a stack data structure in Eclipse IDE with test-driven development method following the provided TDD tutorial and implementation guideline. Process of the study was instrumented by Hackystat Eclipse sensor for collecting development event data.

We realized that it is necessary to have another independent data source to compare data collected and analyzed by Zorro. An Eclipse plug-in, Eclipse Screen Recorder(ESR), was designed and implemented to record changes of Eclipse window caused by development activities into a QuickTime movie[28]. Because ESR can sincerely record development activities, we will have an independent high fidelity data source to validate developer behavior data collected by Zorro. Also we can use the recorded video to validate Zorro episode recognition results. Figure 1.3 presents this pilot validation results in a table.



Subject Index	Duration (Includes break and interruption	Number of Episodes	Classifiable Episodes	Test-Driven Episode Breakdown				Refactoring Episode Breakdown				Test-Last Episodes	Wrongly Classified Episodes	Wrong Percentage (%)
				Type 1	Type 2	Type 3	Type 4	Type 1	Type 2	Type 3	Type 4			
1	44:53:00	15	15	2	3	1	0	0	1	0	0	8	2	13.3
2	28:17:00	13	13	0	5	0	0	0	0	0	0	8	3	23.1
3	48:00:00	14	14	0	6	3	0	0	0	0	0	5	1	7.1
4	66:32:00	14	14	0	4	0	1	0	0	0	1	8	1	7.1
5	43:14:00	16	11	2	1	0	0	0	1	0	0	7	1	6.3
6	45:57:00	11	11	1	0	1	2	0	0	0	0	7	1	9.1
7	32:40:00	9	8	1	2	1	0	0	0	0	1	3	1	11.1
Subtotal		92	86	6	21	6	3	0	2	0	2	46	10	10.9

Figure 1.3. Summary of pilot validation study

The pilot study demonstrated that Zorro can collect complete enough developer behavior data to recognize test-driven development and the recognition accuracy is close to 90% percent (figure1.3). However, 10% episodes were wrongly classified due to incomplete data and insufficient TDD recognition rules. Causes were identified and improvements were made after the pilot study.

A provoking finding of pilot study is that developers spent quite some time doing test-last development (on the contrary to test-driven development), even though they knew it was a study on test-driven development and the guideline on TDD implementation of stack was available for reference. Our conjecture is that it could be caused by the simplicity of stack problem or nature of test-driven development. Further studies on more complicated problems with experienced TDD developers must be conducted to have more comprehensive validation on Zorro.

1.5.2 Case study in Classroom

An extended Zorro validation study is going to be conducted in a software engineering class in fall 2006 to validate Zorro's data collection and recognition of test-driven development as in the pilot study. Students will be taught unit testing and test-driven development before they take participant in this study. The development process will be instrumented by Hackystat Eclipse sensor to collect developer behavior data for Zorro assessment and ESR to record students' development process.

Materials

Programming language is Java and Eclipse is the only IDE supported in this study. Students will learn modern software engineering knowledge and development techniques including JUnit, Eclipse and test-driven development. They will be participating this case study by practicing TDD with one the the following three problems and do some test-driven development on their course projects.

1. Roman numeral converter: a program that can convert any integer number between 1 and 50 into its Roman numeral.
2. Bowling score calculator: a bowling game contains 10 frames, each has 2 maximum throws to knock down as many pins as possible. Score of a bowling game is the sum of each frame and bonus if applicable.
3. Stack: a data structure that works in Last-In-First-Out(LIFO) principle.

Data collection

Hackystat Eclipse sensor will be used through the entire study to have collect developer behavior data. Practice of TDD and the mandatory software development process will also be instrumented by ESR to collect development process video.

Experiment procedure

The classroom case study of Zorro consists of training, TDD practice TDD enactment and course project development.

Training Considering that students are new to test-driven development and they are lack of unit testing skills, we allow a skill building process at the beginning of the semester. Apart from fundamental knowledge of software engineering, students will also learn modern software development tools such as Eclipse, subversion, ANT, JUnit, and software development techniques such as coding standard, MVC pattern, test-driven development.

Practice of test-driven development I prepared three problems roman numeral converter, bowling score calculator and stack with step-wise tutorial of test-driven development implementation for students. Every student takes one of them to practice and their development process will be instrumented by Hackystat Eclipse sensor and ESR.

Enactment of test-driven development Students will be assigned a course project to implement and they must do test-driven development in the first week with at least 3 hours' active time. Again, this process is instrumented by Hackystat Eclipse sensor and ESR.

Course project development After the first week's mandatory test-driven development, it is up to students to choose whatever process works best for them in their course project development. Of course students can still do test-driven development on their own choices. Only Hackystat Eclipse sensor will be used to instrument the course project development.

1.5.3 Case study within TDD community

Eventually Zorro is supposed to help discipline study and management of test-driven development in practice. Case study in academic setting suffers external validity problem, we must run multiple studies[70] to generalize the validation results of classroom case study of Zorro. With the resource available for my Ph.D research, it could be viable to reach experienced TDD developers of test-driven development community.

Test subjects

The best way to reach experienced TDD developers is through TDD community, website *testdriven.com* and TDD user group [53]. I will solicit participation in the study of experienced TDD developers by evangelizing Zorro software system into the community with the hope that they can see the benefits of it in their professional software development.

Data collection

The unobtrusive data collection manner of Zorro and intuitive process recording tool ESR[10] should make the participants painless to configure the testing environment. Hackystat provides a DocBook chapter of Zorro case study guideline[19] to assist developers on configuring testing environment.

Experiment procedure

Participants will do test-driven development on their own with instrumentation of Hackystat Eclipse sensor and ESR. To encourage participation, test subjects can choose the problem they want to work on in test-driven development. Problem candidates are:

- One problem from problemset of stack, roman numeral converter and bowling score calculator.
- Another interesting problem such as Money example, Sudoku or Spreadsheet.
- A test-driven development session at work.

For Zorro validation, we suggest developer working on the chosen problem in test-driven development for more than half an hour. Developers can send their ESR movies to me for data analysis. A summary of the analysis result will be avaulable for them as reference.

Validation analysis

Developer behavior data and ESR video are two data sources for analysis. A Zorro project can be configured to every test subject to run Zorro analysis. Similar as pilot study we will use the recorded video to validate the collected behavior data and the recognition results.

1.6 Contribution

I proposed software development stream analysis (SDSA) framework to study low-level software process with very fine-grained developer behavior data. Unlike traditional software process research that treats software process as either a rigid process defined by formal methods or a stochastic process, SDSA adopts the mix of bottom-up and top-down method to assess the compliance of low-level software process with rule-based system support. It introduces a new and executable way to automate process comformance evaluation.

The software system out of this research work is Zorro. If the validation studies agree that Zorro can collect complete enough developer behavior data and recognize test-driven development correctly, the TDD community and researchers will have a very powerful full to improve TDD practice and find ways to improve it. The possible uses of Zorro in test-driven development process mangement and training will greatly improve the discipline of TDD in practice and education. Other than compliance assessment of test-driven development, the rich developer behavior data collected in the development process can be used for other research such as development pattern search.

Another contribution of this research will be the Eclipse Screen Recorder (ESR), the data validation tool. It can be used to conduct developer behaviors related research such as peer software review process observation.

1.7 Roadmap

This thesis work introduces Zorro software system for recognizing test-driven development to improve practice and empirical evaluation of it with software development stream analysis technique. Chapter **??** briefly goes over literature work on test-driven development and automation of software process research. Zorro architecture and the implementation details are addressed in chapter **??**. Case studies to validate Zorro are described in chapter 3.

Chapter 2

Related Work

2.1 Extreme Programming and Test-Driven Development

2.1.1 Extreme Programming

Extreme Programming (XP) is a light-weight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements [4]. It take common-sense principles and practices to extreme level.

- 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).

XP is an agile process including 12 practices of: Planning Game, Small Releases, Metaphor, Simple Design, Testing, Refactoring, Pair Programming, Collective Code Ownership, Continuous Integration, Forty-hour Week, On-site Customer and Coding Standards. [4] As Kent Beck said XP is not from ground up but many good practices derived from previous software development. In XP these 12 practices support each other and complement each other's weakness. (Figure 2.1 Excerpted from P70 [4])

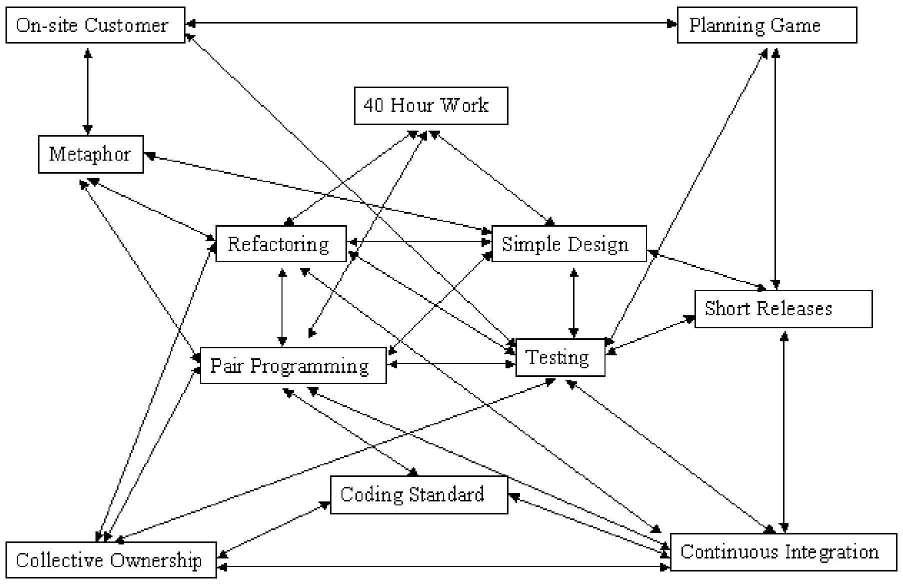


Figure 2.1. XP Support Network

A typical XP project starts from planning game. Release plans will be laid out in the planning game. Each release will vary from several weeks to several months but not over half a year. Customers or stake holders define release scope and important or critical business components go first. Because XP is aiming at building simple but works software the advanced and not-necessary features will not be considered until demanded. This will maximize the Return-on-Investment (ROI). Communication, simplicity, feedback and courage are four values XP will provide. Communications between customer and developer, coach and developer, inter-developers are encouraged by XP practices such as on-site customer, metaphor, coding standard, collective ownership and pair programming directly or indirectly. Feedback and courage are provided by test-driven development, on-site customer, pair programming and continuous integration. Simplicity is the theme of XP through the entire development. XP is another kind of iterative incremental development (IID). The release plan is iterative and development in each release is iterative too. Iteration in development is from user story that can last only several days. The task assignment is done in stand-up meeting. [4]

Management of XP is through metrics, coaching, tracking and intervention. It has roles developers, customer, tester, tracker, coach, consultant and big boss. These roles are responsible for smooth execution and balance maintenance of all 12 XP practices. Though XP acknowledges that each practice has its own weakness it is not feasible for a developing team to tranfer to XP seamlessly. Don Wells [4] commented that XP adoption can be iterative:

1. Pick your worst problem.
2. Solve it in XP way.
3. When it is no longer your worst problem, repeat.

As we already know XP consists of 12 practices. Many of them can exist independently. Planning game can fit into other IID process. On-site customer can be available to other process models such as water-fall model. Among 12 XP practices Pair Programming (PP) and Test-Driven Development (TDD) often exist independently. They are widely applied and studied by software developers, educators and researchers.

2.1.2 Pair Programming

Pair programming is an Extreme Programming practice. In pair programming two programmers sit side-by-side at one computer, continuously collaborating on same design, algorithm, code or tests. One acts as the driver who types at computer. Another one acts as the navigator who is responsible for high-level tasks like over-viewing design strategy, inspecting code being typed for typos, syntax errors, or defects. Roles in pair programming are dynamic and they can be exchanged during the programming session or rotated in different programming sessions. Also the pairs can be dynamically formed in a developing team. Two people actively work on the same programming task with continuous collaboration.

Pair programming takes the privilege of code inspection a.k.a. code review to improve code quality. Even though design or coding errors still can exist but they will not last long because driver has to explain what he or she is doing to navigator which provides a chance for both parties to think it over. It looks like resources are wasted in pair programming because two developers are invested on tasks that can be done by a solo developer; however, Laurie Williams's case study concluded that pair programming will not double development effort. In her study paired programmers are only 15% slower than two independent individual programmers but produced 15% fewer bugs. The knowing fact is that test and debug cost will be much higher than initial programming so it will be economically paid off, especially when some team members have to leave.

In extreme programming pair programming serves multiple purpose. Programs are written by driver and navigator such that code review is done simultaneously. The duo continuously communicate for better solution and everyone inspects other's work. If driver goes the wrong way it can be adjusted quickly by navigator without committing more mistakes, for instance, when driver is not creating a test case before implementation navigator can point it out so that they stay on the right track. Since driver and navigator are exchanged during development both parties know code well and own it. In case one programmer has to leave the developing team risk is minimized because other people can take his or her work over easily. The other benefit of pair programming is that junior programmer can be coached and knowledge can spread over in the team.[39]

2.1.3 Test-Driven Development

Test-Driven Development is another Extreme Programming practice that has its own benefits alone like pair programming. It's both the developing method and design tool in extreme programming. Often it is called Test-First Design (TFD) or Test-First Programming (TFP) because of its natures. It has two fundamental rules:

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

Development of TDD is iterative. Test and implementation are added incrementally under these two rules. An iteration can be elaborated as following [49]:

1. Write the test
2. Write the code
3. Run the automated tests
4. Refactor
5. Repeat

TDD is on the ground of a universally agreed claim that testing is good to software project success. If it is done perfectly there is no need to have a coverage tool because system is always

100% tested. It supports simplicity since developers only need to write enough code to make test pass. Refactoring happens at the end of each iteration. Also unit tests can serve as executing documentation of system so that re-usability is improved.

In TDD unit tests are supported by XUnit framework. It is brought up by Kent Beck, Ward Cunningham and Ron Jeffries in 1996 [11]. It has the following structure [5]:

1. Invoke test method
2. Invoke setUp first
3. Invoke tearDown afterward
4. Invoke tearDown even if the test method fails
5. Run multiple tests
6. Report collected results

In recent years xUnit has already been ported to more than 30 language supports such as JUnit for Java, PyUnit for Python, NUnit for C#.NET, PHPUnit for PHP, CPPUnit for C++, DUnit for Delphi etc. [69] and it has becoming the de facto standard of unit testing in software development. With xUnit the unit testing is shifted from quality assurance specialists or customers to developers.

This framework modulates unit testing onto method level. All public methods in object-oriented domain are testable with this framework. It moves unit testing from customer-oriented functional testing to the combination of developer-oriented unit testing and customer-oriented functional testing. Once project is brought up for functional testing it is already in high quality insured by unit testing.

Theoretically speaking it is possible to do TDD perfectly but it is not feasible in reality. First, developers as human beings are not good at executing these two rigid rules. Second, there exists many holes such that it is either not possible to do unit testing to some code or practically infeasible in some cases. For instance, private methods are not accessible for test purpose unless developer exposes them; GUI or event-driven methods are not testable because they need humans intervention to make it happen; some operations like database accesses are very time consuming such that unit testing will take long time to run. Because of these limitations developers would not be able to follow TDD rules perfectly as Kent Beck wished.

In extreme programming process TDD can be executed better because other practices can support it as figure 2.1 shows. Pair programming supports TDD because two brains are better on creating good unit tests and navigator can inform driver if unit tests are not created before implementation. Coach can help pairs to design unit testing in case it is hard or infeasible to do unit testing. Continuous integration can help developers be aware of some unit tests fail or there is no unit test to some code.

2.2 Test-Driven Development in Practice

Test-Driven Development is an incremental software development methodology that stresses on exhausitive unit testing by creating test cases before code implementation. It's a design and implementation methodology but testing. It ends with a rich suite unit test cases and high quality code as it promises. It appears very often in software development tutorial workshops ([57, 59, 37, 48, **?**]), technical reports, journals ([56]) and blogsphere ([54]). Also many commercial and open-source supporting tools ([62]) are developed in recent years.

2.2.1 Characteristics of Test-Driven Development

Disciplined Small Steps

In TDD development is incremental and iterative. Developers only write a small portion of a unit test, or just one assertation each time, and then just write enough code to test pass. At the beginnig the test target does not exist so it cannot pass compilation. For instance, before we create class Stack for a stack implementation we create a test class called TestStack.

```
package com.stack;

import junit.framework.TestCase;

/**
 * Tests stack operations.
 */
public class TestStack extends TestCase {
}
```

To test whether stack can report its emptiness correctly developer adds one assertation only in the first iteration.

```
  /**
   * Tests empty stack.
   */
  public void testEmptyStack() {
    Stack stack = new Stack();
    assertTrue("Test empty stack", stack.isEmpty());
  }
```

It is not runnable because Stack class is not created yet. After creates dummy implementation or stub of Stack it still cannot be compiled because isEmpty() is not provided.

```
package com.stack;

/**
 * Implements a stack.
 */
public class Stack {
  /**
   * Constructs a stack.
   */
  public Stack() {
  }
}
```

To make it compile developer goes ahead to add method isEmpty() which just returns a constant.

```
  /**
   * Whether stack is empty.
   *
   * @return True if stack is empty.
   */
   public boolean isEmpty() {
     return false;
   }
```

Now it compiles but test fails because isEmpty() method simply returns false. To make test pass developer changes it to return true. Then he/she can go ahead to add another assertation and test case. As we can see from above example development pace is really small in TDD. Generally one iteration or cycle lasts from 30 minutes to 5 minutes. It rarely grows to 10 minutes [58]. It is totally okay to just add implementation stub or simply returns a constant value to fake implementation (p13 [5], p169 [3]).

Consistent Refactoring

Refactoring is an important portion in Test-Driven Development, it the second TDD rule. By definition, refactoring is a disciplined technique for restructing an existing body of code, altering its internal structure without changing its external behavior [42]. To make it simple and concise refactoring is to change a program's interal structure without affecting its external behaviors. In TDD developers need to consistently do refactoring to support growing test suite without leaving implementation code badly structured. When we do TDD the first priority is to get test pass and the next is to remove duplication [5]. These two steps fulfil a complete TDD cycle. ”Make it run, make it right.” p24 [5] Using the above example we can refactor isEmpty() method to return an instance variable instead of a constant.

```
private int size = 0;
/**
 * Whether stack is empty.
 *
 * @return True if stack is empty.
 */
 public boolean isEmpty() {
   return this.size == 0;
 }
```

Now we run TestStack it passes. The duplication can be either on test code or production code. As long as you get all tests pass you are confident to do refactoring because unit tests can ensure you will not commit unrecoverable big mistake.

Courage, Rapid Feeback and One Ball in the Air at Once

Kent Beck thinks TDD is a way of managing fear during development [5]. Programmers are in fear in software development because the requirements are not clearly stated, techniques are new, problems are hard to solve, or there is no enough time to do a thorough design and review. Fears happen more often to novice programmers but experienced programmers have fears too because a smallest mistake may break the whole system. Experience programmers are good at debugging and have better intuition on finding and solving bugs than novice programmers. In TDD developers “test a little, code a little and repeat.”[5] Test cases guard code development and provide instant feedback. Since the incremental step is small it is impossible to commit big mistakes. If test fails it is easy to fix it too. In software development developers carry the baggage with requirement, system structure design, algorithm, code efficiency, readibility and communication with other code etc. According to Martin Fowler it is like keeping several balls in the air at once (Page 215 [5]). In TDD you only keep one ball in the air at once and concentrate on that ball properly. Developer only needs to make the test pass without worrying it is a good or bad design in TDD. In refactoring step developer only worries about what makes a good design. Keeping one ball only in the air helps doing good job on every aspect.

Always Working Code

TDD developers maintain a comprehend unit test suite and all test pass at the end of each TDD cycle. The developing system is always working to the designated tasks represented by unit tests. Since all test cases are created in the domain of xUnit framework they can be integrated together to do continuous integration. A test bot or agent can be setup to run all tests everyday or every few hours [7]. It works well especially in team software development or in case it is not feasible for a developer to run all tests in development in time manner.

2.2.2 Benefits of Test-Driven Development

100% Code Coverage

If it is done perfectly TDD should always yield 100% code coverage [5, 18] as figure 2.2 shows (Excerpted from page 139 [18]).

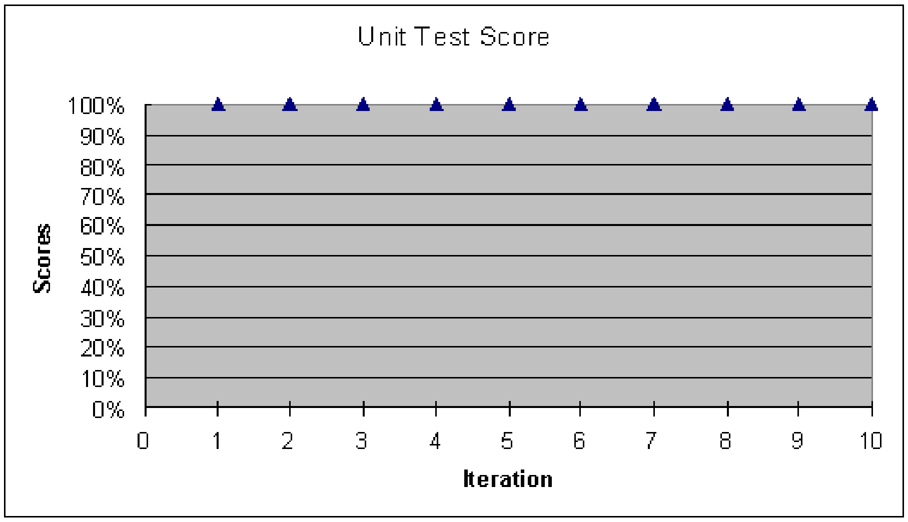


Figure 2.2. Unit Test Scores

In practical it is not feasible to maintain 100% but TDD developers do yield high code coverage. Boby George and Laurie Williams's study finds that TDD developers' test cases achieved 98% method, 92% statement and 97% branch coverage [13]. The experiment was conducted on a XP developing team with Pair Programming practice too. Another study conducted by Matthias Müller and Oliver Hagner in University of Karlsruhe found that TDD developers yielded 74% median branch coverage [35]. It's quite surprising because it is even lower than control group without doing TDD whose median coverage is 80%. On the fact that it is impossible or not necessary to achieve 100% this test coverage is still low.

Regression Test

One principle of TDD is to create a regression unit test if system breaks. The test fails first just as a typical TDD iteration. Code isolation is another TDD principal that reduces coupling between objects. High CBO is bad in object-oriented programming from modular design and reuse point of views [38]. It's also straight forward the coupling can not be zero because differant parts have to interact to make the system work. In TDD developers run all tests regressively to make sure the new change will not break other parts. Speaking in Java all package should have class `TestAll` which assemblies all unit tests in the same package together. When developer is working she or she just focus on tests in this small area. XP has another practice called continuous integration to execute all unit tests in timely manner to check system's consistency. If the new change breaks system developer will know it in a few hours or a day. A byproduct of TDD is a very comprehensive and exhaustive test set that can serve as regression test suite.

Executable Documentation

In TDD there is no stacked written design documentations instead of executing design documentation in unit tests. They are system-level documentation and show how developers intended for the class to be used [51, 1].

Clear Design

XP community argues that TDD is not about test but design [52]. There is no big up-front design in XP and TDD but small incremental change. Design decision is made incrementally by refactoring. Developers do incremental change to the current code base instead of following design documentation. Jim Little concluded that this evolutionary design yields better results than big up-front design [30]. The design architecture with upfront design generated a five-layer structure for data persistence which is too complex in practice and the developing team was completely lost. Intead the evolutionary is easy and effective in architecture design. Also, TDD developer will not do anything extra except for passing unit tests. It makes clean code that works [5] and is testable.

High Quality Code

Probably the most significant benefit of TDD to software development is high code quality. Generally speaking TDD developers write code with high coverage and the software created with TDD is more reliable than non-TDD approach. Matthias Müller and Oliver Hagner's experiment in University of Karlsruhe found that Test First Group (TFG)'s program has higher reliabilty than controll group's code. In their experiment five TDD developers achieve reliability over 96% compared to only one program from the control group that achieved this[35]. Using black box test as external validation TDD pairs passed approximately 18% more black box tests in case study conducted by Boby George and Laurie Williams [14].

Increased Productivity

TDD is faster test-last and code-N-fix. In TDD, testing is part of the design process, it doesn't take long time to write a small test [50]. It is faster than test-last because developers need to spend same amount or more time on creating tests after implementation. Boby George and Laurie William's study showed that TDD developer spent 16% more than controller group but controller group did not primarily write any worthwhile automated test cases though there were instructed to do so. TDD code is much more reliable and with high code coverage. Since code is less buggy it will save a lot time on debugging and maintenace.

> "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" – Thomas Eyde [2]

2.2.3 Barriers of Test-Driven Development

More and more developers turn to TDD and many comercial and free workshops are provided to evangelize TDD [54, **?**]. Also, many tools are created to support TDD [62]. But still, many developers are reluctant to TDD or even to give it a serious try despite on the claim that TDD is infective [5, 24, 66, 40].

Testability

Darach Ennis argued that there are a lot of fallacies blowing around various engineering organization and among various engineers [5].

- You can’t test GUIs automatically (e.g. Swing, CGI, JSP/Servlets/Struts)
- You can’t unit test distributed objects utomatically (e.g., RPC and Messaging style, or CORBA/EJB and JMS)
- You can’t test-first develop your database schema (e.g. JDBC)
- There is no need to test third party code or code generated by external tools
- You can’t test first develop a language compiler/interpreter from BNF to production quality implementation.

Most of these arguments are still valid but some operations are testable nowadays. JSP and servlets can be tested by HttpUnit[68] or Cactus[25]. Mock object and Easy Mock can be used to test some complex operations by providing fake implemention to the real system [33].

Too Much Tests

TDD is about design. To implement a task or user story developer makes a list of test cases and maintains it when code grows. When it is empty and there is no more test case developers can think of, task is done. All code comes with test and sometimes test code may be larger than production code. Apparently developers need to spend time on tests which are not thought as production from customer point of view. XP says that it saves money for customers on debugging and maintance, which is still not proved yet. A model about Return on Investment (ROI) of TDD was brought up by Matthias Müller and Frank Padberg shows how TDD can pay off the investment on test[60]. So far there is still no case study on cost benefit analysis on TDD yet.

Small Steps and Time-Consuming

In TDD developers make small step each time. There will have many context swith between test code and production, also many IDE activities. Using previous isEmpty() method as example.

```
/**
 * Whether stack is empty.
 *
 * @return True if stack is empty.
 */
```

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

It is such a trivial thing such that most developers can make it right without having a failed test. Kent Beck's opinion is that you can write test that encourages hundreds of lines of code and hours of refactoring but the tendency of TDD is to have smaller steps over time. Some developers switched to TDD when old method cannot work, on example is defect removing in debug.

2.2.4 Testing Techniques and Tools

XUnit and its Related Tools

XUnit is the corner stone of TDD. It makes test very simple and test automation possible. The variation of xUnit includes JUnit for Java[16], SUnit for Small Talk[46], CPPUnit for C++[8], NUnit for C#.NET[36], VBUnit for VB[67], PYUnit for Python[41] etc. There are also some extention for JUnit to test some complex operation, for instance, HttpUnit for web[68], Cactus for Servlet[25], and DBUnit for database[9]. A new test tool called TestNG is developed to simplify unit test using annotation and configuration file [17].

Mock Object

2.3 Case Studies on Test-Driven Development

E. Michael Maximilien and Laurie Williams assessed Test-Drive Development at IBM in the development of a new IBM Retail Store Solutions version.[31] The initiation of TDD came from the fact that defect rate of each revisions did not drop in Functional Verification Test (FVT) though developers have rich domain knowledge. TDD was introduced to the developing team to alleviate the recurrent quality and testing problems. In their study they found defect rate dropped by 50% in FVT whereas productivity was not affected. They believe that the drop of productivity by TDD was complemented by the adoption of Microsoft Project Central project management tool. The byproduct of TDD is a substantial suite of unit tests. They also made test automation be possible and tests were exercised once a day with the integration support. They also verified the moral of TDD – ”test-infected” phrased by Erich Gamma[5]. Developers were positive on TDD and intended to continue using it in their future development.

Boby George and Laurie Williams summarized their findings regarding to Test-Driven Development in three trial experiments [14]. TDD approaches yielded superior external code quality measured by a set of black box tests. TDD developers' code passed 18% more functional black box tests than controlled groups' code. Their results also showed that TDD developers took more time than controlled groups. Additionally controlled group did not write worthwhile automated test cases, which makes the comparison on productivity uneven. The projects created by TDD developers have very high test coverage. The mean method coverage is 98%, statement coverage is 92% and branch coverage is 97%.

Another case study conducted by Matthias M. Müller and Oliver Hagner in University of Karlstruhe to test development efficiency, resultant code reliability and program understanding. They found that switching to TDD does not improve productivity and the code will not be more reliable than with TDD approach. The only improvement is on code reusability because tests help developers to use method or interface correctly [35].

2.4 Hackystat

Hackystat is an in-process automated metric collection system designed and built in Collaborative Software Development Laboratory in University of Hawaii. The attached IDE and ANT build sensors can collect the development activities including file editing, class creation, method addition and deletion as well as software metrics like unit test invocations, build invocations, dynamic object-oriented metrics and other metrics as well. Using Hackystat rich software metrics can be collected automatically to study the development process without much work effort involved.

2.4.1 Software Metrics

A software metric is a measure of some property of a piece of software or its specification [47]. Common software metrics include source lines of code, object-oriented metrics such as number of methods in a class, coupling between objects, number of children etc, and other metrics like function points, bugs per thousand lines of code, code coverage and so on. Software metrics are widely used in software process to predict and manage software development. A famous word by Tom Demarco is that you cannot control what you cannot measure in controlling software development [47]. Software metrics make software development process be measurable and process improvement be feasible.

Koch summarized that there are two sets of objectives of software metrics [32] :

- Measures are needed to develop project estimates, to monitor progress and performance, and to determine if the software products are of acceptable quality.
- To the software organizations measurement can be used to determine overall productivity and quality level, to identify performance trends, to better manage software portfolios, to justify investments in new technologies, and to help planning and managing the software function.

2.4.2 Personal Software Process

Personal Software Process (PSP) is a manual approach to record personal software development activities to help project planning and estimation. It's created and evangelized by Watt Humphrey at Software Engineering Institute (SEI) in Carneige Mellon University. SEI provides a series of training courses for developers and educator to grasp it [**?**]. PSP has four levels range from 0 to 3 and the training course provides 10 programming exercises and five reports. On level 0 developers learn to record their current practice using time recording log to understand how time is spent to improve time usage. Other metrics like project size and defects are measured too on level 0. On level 1 developers will do project planning and estimation with the metrics data collected to improve estimation accuracy. Code review and design review are introduced on level 2 to improve personal software quality. Level 3 is called cyclic personal process. It suggests developers should divide large tasks into small pieces to develop with PSP as Iterative Incremental Development (IID) advocates.

Many researches and experiments were conducted on PSP. Most of them show that developers can improve productivity and reduce defect density with PSP and the estimation accuracy is improved. Some researches reported poor support for team software development and issue with data collection. Ann Disney found that data entry is error prone in PSP.

2.4.3 Automated Personal Software Process

PSP helps individual developer to improve personal software process maturity. To lower data entry overhead and improve PSP data quality. Carleton A. Moore developed LEAP Toolkit in Collaborative Software Engineering Laboratory in University of Hawaii. With LEAP Toolkit developers can record their time and enter data easily. It improves data accuracy and provides regression analyses that cannot be done manually in PSP [34].

2.4.4 Hackystat

Tools like LEAP Toolkits make it fairly easy to record PSP data and conduct project planning and estimation. In LEAP developers enter PSP data using clock control and other data entry forms but developers will have to stop their on-hand work often to input data, which reduces its effectness on process improvement, project planning and estimation. It is also hard to do collaboration to support team project development. In 2001 Philip Johnson etc started Hackystat project to collect development automatically. Hackystat sensors can be installed in development environments such as XEmacs, JBuilder, Eclipse and Visual Studio etc to collect software development data unobtrusively. Hackystat sensors will send out metric data collected to a centerized data server with SOAP protocol [21] in a period-based fashion. Hackystat is extensible so that many kinds of metrics can be collected except for time, size and defect measurement. Most development activities such as opening file, editing file, closing file and refactoring data can be collected by Hackystat sensors. Unit test case exercises and test coverage can be recorded too by Hackystat. It supports development collaboration by defining project with common workspaces[21]. Initial case studies ([20], [22]) found that Hacksytat has very low overhead for students and Hackystat analyses are very helpful on student projects. Usage of Hackystat on extreme programming, high performance computing, project management and software reviews are being explored in Univeristy of Hawaii and affiliate organizations.

Chapter 3

Zorro Case Studies

Test-driven development is a low-level software process that consists of many short-duration activites—file edit, compilation, unit test, debug, so on and so forth. Zorro software system collects and analyzes on these low-level and short-duration activites to detect the existance of test-driven development with software development stream technology and rule-based system supports. The test run and pilot study have demonstrated that Zorro recognized test-driven development successfully. Yet, following three issues are remained to be resolved before we deploy Zorro in actual software development.

- *Data collection problem:* Does Zorro collect fine enough developer behavioral data to recognize test-driven development?
- *Result correctness:* Does Zorro infer test-driven development correctly from the collected developer behavioral data?
- *Detection of alternatives:* Test-driven development is a new best practice and it may not be so perfect that it is applicable to all applications in all the time. Can Zorro tell the difference when developers do the alternative processes, such as test-last development — developer writes unit tests afterward intead of test-driven?

Experiments and case studies should be carefully designed and carried on in order to resolve these issues systematically. In January 2006, a pilot validation study was conducted to test Zorro and the validating method, which turned out to be a success[28]. Furthermore, an expanded replication study is scheduled in a software engineering class in fall 2006 for statistic correctness among junior TDD developers. According to Yin [70], single case study in one organization suffers external validity problem and research conclusions of it can not be generally applied; therefore, I am going to conduct another case study with experienced TDD developers. These three Zorro validation studies are carefully designed and described in this chapter after an introduction of Eclipse screen recorder utility[10], a development process recording tool.

3.1 ESR: a tool for cross-validation

To validate Zorro's data collection and TDD recognizing result, we should have an independent and fine-grained data source on low-level development activities for cross-validation. Considering that human being observation of development process can not record fine enough activity data and video recording with camcorder is too hassle to setup for test, I designed and implemented ESR[10], a lightweight development process recording tool.

3.1.1 Requirement analysis of ESR

Zorro collects low-level development activities and infers test-driven development process with the collected data. Compared to high-level process activities such as requirement analysis and system design that may last months or years, low-level development activities only last seconds or minutes. Thus, one requirement is that ESR can record very fine-grained data to observe activities happened in seconds.

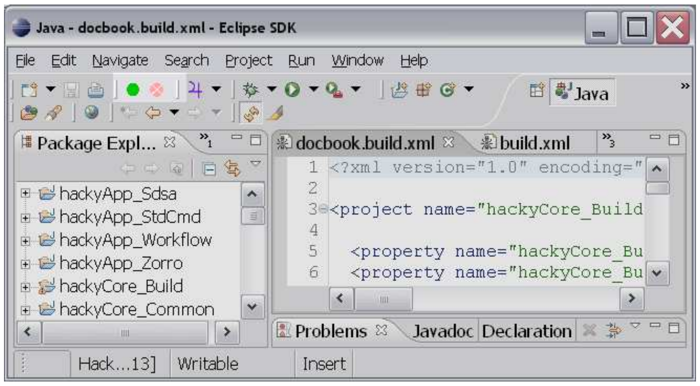
Participants will send the software development process video recorded by ESR to researchers for analysis via email. The video should be readable for researchers to tell development activities from analysis wise, and size of the videos must be small for transfer.

Also, ESR can not use too much CPU resource while recording because developers will develop software at the same time. In worst scenario, it should not use more than 50% CPU resource in order to avoid delay of response for development activities.

3.1.2 Design and implementation of ESR

Participation observation and video recording are two most often used approaches in human behavior related research. In the case of Zorro validation study, participation observation is not plausible because human being can not keep up the rapid pace of low-level software development activities in short-duration. This leaves the alternative method, video recording, as the only choice. Because laboratory test is expensive to set up and it can deflect what developers normally do in their working environment, we should allow them work in their familar environment. As the trade-off, we ended up with a screen recording tool that can record changes of computer screen caused by development activities.

The wide adoption of Eclipse IDE incites us to design and implement the screen recording tool as an Eclipse plug-in named ESR[10], acronym of Eclipse Screen Recorder. It captures Eclipse screens in a fixed sampling rate, computes delta changes of consecutive screens, and writes screen changes into a movie file. Figure 3.1 is the ESR user interface, which consists of a green



```
<?xml version="1.0" encoding="

<project name="hackyCore_Build

  <property name="hackyCore_Bu
  <property name="hackyCore_Bu
```

Figure 3.1. ESR user interface

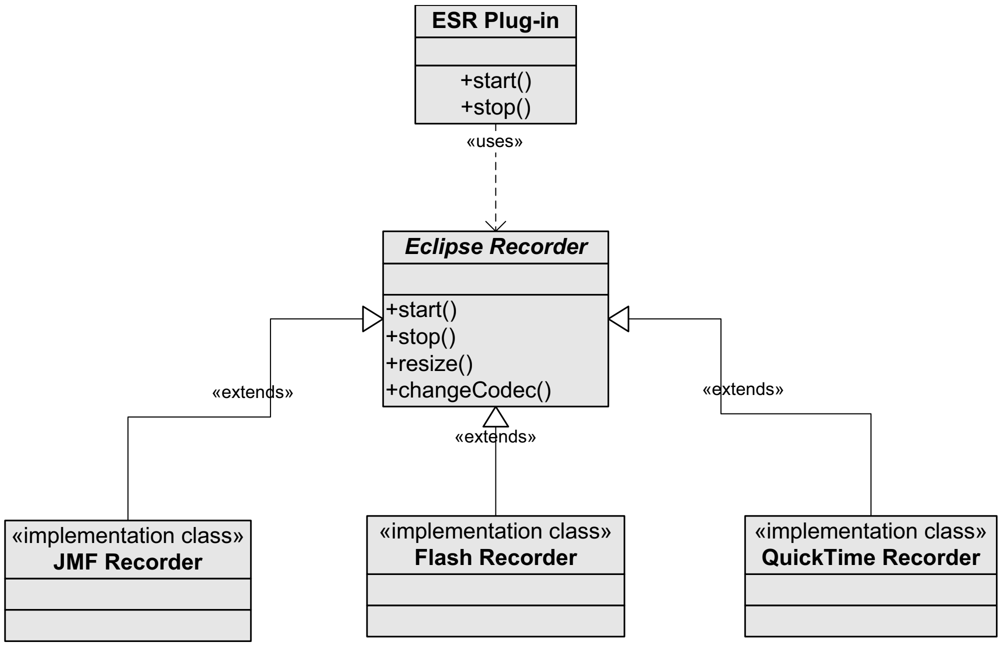


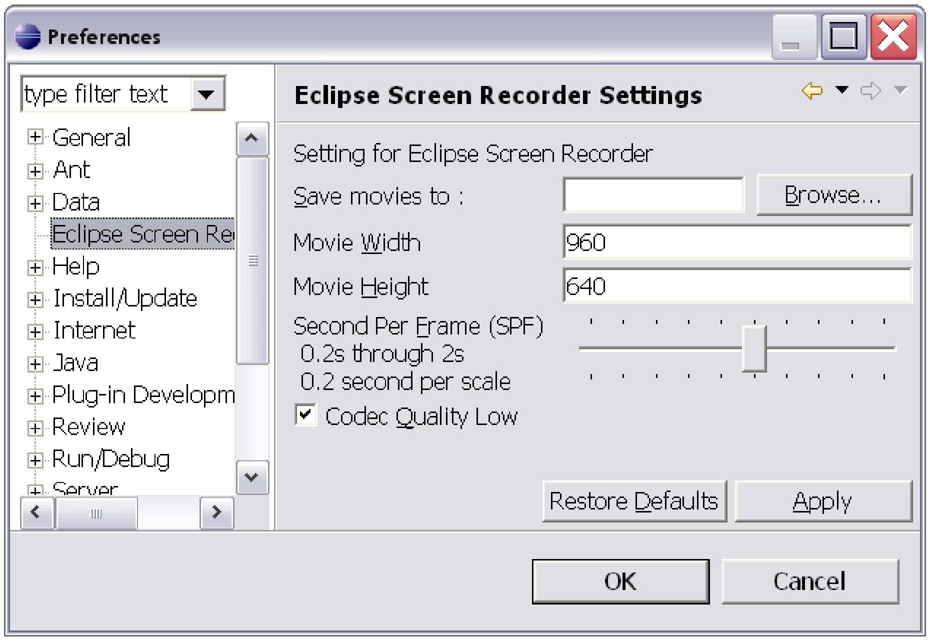
Figure 3.2. ESR Plug-in Structure

button and a red button only in Eclipse toolbar menu. Internally, ESR defines one abstract recorder and three concrete recorder implementations using Java Media Framework, Flash and Quick Time respectively as shown in figure 3.2.

3.1.3 Using ESR

We recommend QuickTime recorder because its video compressing rate is the highest one among three ESR recorders: the size of one hour's software development movie file typically ranges 5mb-10mb depending on main frame changes and resolution preference. The only down side is that a fast computer is desired for QuickTime recorder functioning well without causing noticeable delay of response to software development activities in Eclipse IDE.

ESR can be easily configured using Eclipse preference page as shown in figure 3.3. Practical use of ESR found that it can capture 1280*1024 Eclipse window in one frame per sec-


Eclipse Screen Recorder Settings

Setting for Eclipse Screen Recorder

Save movies to :

Movie Width 960

Movie Height 640

Second Per Frame (SPF)
0.2s through 2s
0.2 second per scale

☑ Codec Quality Low

Figure 3.3. Configuration of ESR Plug-in

ond rate, resize it to 960*680 image and create QuickTime movie without significant lagging on a 2GHz PC.

3.2 Pilot study

3.2.1 Subjects

3.2.2 Experiment setting

3.2.3 Procedure

3.2.4 Data analysis

3.2.5 conclusion and discussion

3.3 Classroom study

The pilot study (Section 3.2) proves that Zorro can collect enough developer behavior data to derive test-driven development in high accuracy, in the mean time it also confirms us that the validation study procedure works well. To be statistically correct, we will conduct an extended replication Zorro validation study in a software engineering class in fall 2006.

3.3.1 Goals and hypotheses

As an extended study of pilot study on Zorro validation, the goal of this study is to validate Zorro's capabilities on developer behavior data collection and test-driven development recognition. Hypotheses to test are:

- Hypothesis 1. *Zorro can collect enough developer behavior data to recognize test-driven development.*
- Hypothesis 2. *Zorro can correctly infer test-driven development with the collected developer behavior data.*
- Hypothesis 3. *Zorro can detect alternative processes when developers do not do test-driven development.*

Kent Beck, pioneer of test-driven development, ever claimed that developers would be "test infected" after they give it a serious try. With the capability of this study, we will evaluate this claim as well. The fouth hypothesis is:

- Hypothesis 4. *Students will get "test infected" and stick to tdd in their course projects development although test-driven development is elective.*

3.3.2 Subjects

Test subjects are students in a graduate level software engineering class. Prior to the study we will ask students to sign consent form (see appendix A) to inform them that this study is part of a scientific research, data collected will not be used against their gradings and participation is voluntary. Human subject clearence exemption of this reseach project was already granted by University of Hawaii Committee on Human Studies.

3.3.3 Experiment setting

Elements of Zorro validation are Zorro-compliant IDE, Hackystat sensor for developer behavior collection and ESR for development process recording. Currently, only Eclipse IDE has a sensor that is designed to collect development events suitable for Zorro processing. Hackystat Eclipse sensor and ESR will be used to instrument students' test-driven development process for colleting development data. In a nutshell, required experiment settings are:

- *Windows-based pc(≥1.8GHz CPU and ≥512MB RAM*)
- *JDK 1.4 or JDK 5*
- *Eclipse SDK 3.2*
- *Hackystat Eclipse Sensor (up-to-date version)*
- *ESR with QuickTime(up-to-date version)*

As discussioned in section 3.1, this experiment will not be conducted in laboratory setting which may deflect how students write program in actual situation. Instead, student participants will work on their own PCs at the time to their conveniences.

3.3.4 Experiment procedure

Experiment procedure of this study includes four steps — training, practice of test-driven development, enactment of test-driven development, and course project development in elective process.

Training

Software testing is a major topic of software engineering, but it used to be taught by instructors in a later time as a software quality assurance technique following the waterfall model.

Modern software development intends to emphasize on software testing at early stage of software process to improve software quality. Thus, lectures on software testing will be given at an early time in the software engineering class. Recommended lectures are:

- Lecture 1. *Software testing methods: system testing, validation testing, integration testing and unit testing. Reading materials: software testing chapter of textbook.*
- Lecture 2. *Methods and tools: xUnit, JUnit, JUnit in Eclipse, JUnit for ANT and Test-Driven Development. Reading materials: JUnit 3.8 Cookbook, preface and chapters 1-2 of book "Test-Driven Development by Example"[5].*
- Lecture 3. *Software metrics, Hackystat infrastructure, software project telemetry. Assignment: install Hackystat sensor, develop some simple code, and invoke Hackystat analyses.*
- Lecture 4. *Test-Driven Development, Zorro software system, and Eclipse Screen Recorder. Assignment: install and configure ESR, develop a trivial problem in TDD with Eclipse sensor and ESR instrumentation.*

Consent form (appendix A) will be distributed to students after we make sure that students can get along well with Hackystat and ESR.

Practice of test-driven development

It is needed and necessary to practice test-driven development for students from a lesson learned in the pilot study: although test subjects were explicitly told to do test-driven development with the supplied task list, 50% episodes are neither test-driven nor refactoring in pilot study. In addition, it is important to ensure that students are comfortable with JUnit, test-driven development and instrumentation tools — Eclipse sensor and ESR. Therefore, we will ask students to practice test-driven development on a well-known problem with the supplied tutorial chosen from the following three candidates:

- Roman numeral converter: *is a program that can convert any integer number between 1 and 50 to roman numeral.*
- Stack: *is a data structure that works in Last-In-First-Out principle.*
- Bowling game: *a single bowling game consists of ten frames. In each frame the object is to roll a ball at ten bowling pins.*

The practice of test-driven development will be instrumented by Hackystat Eclipse sensor and ESR, and the collected data will be used to validate Zorro. With this practice we will know that:

- *students understand red/green/refactor rhythm of TDD with hands-on experience;*
- *students can get along well with ESR, the screen recording tool.*

A follow-up survey (see appendix B) will be given to students to investigate their opinions on test-driven development after they finish the test-driven development assignment.

Enactment of test-driven development

In the previous step, students practice test-driven development on a well defined problem with step-wise tutorial, which might be too easy compared to actual problems in software development. Therefore, we will ask students to work on their course projects using test-driven development for a while. Purpose of this study is to: validate Zorro with actual software development data; and urge students do test-driven development seriously.

Same as last step, students will develop software in Eclipse IDE with the instrumentations of Hackystat Eclipse sensor and ESR. It required to do more than 3 hours' test-driven development in the first week after course projects are assigned. Development activity data and ESR video are collected for Zorro validation analysis.

Another survey (apprendix C) will be conducted thereafter to readdress student's opinions on test-driven development and intention to continue using it in their future software development.

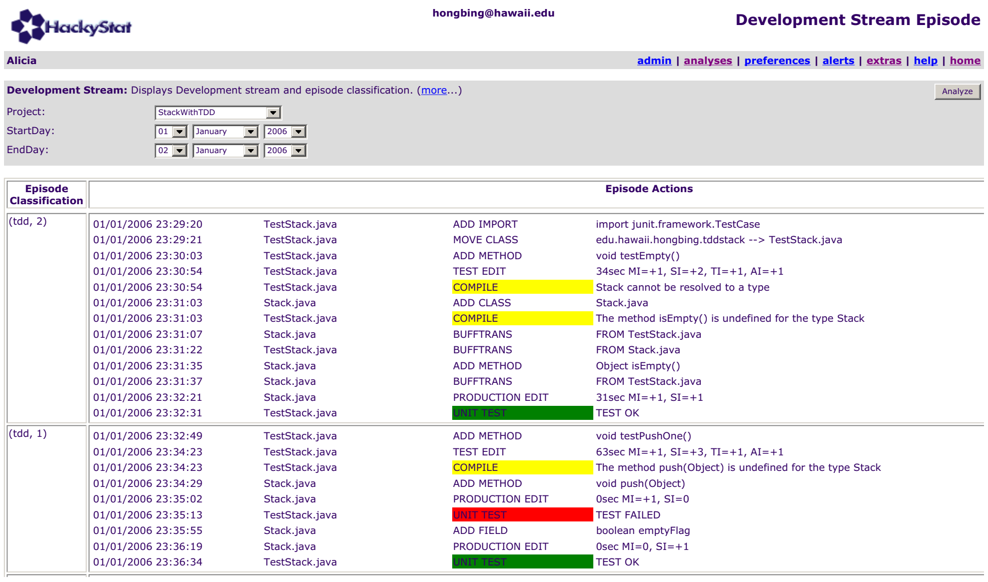
Course project development in elective process

In the rest of course project, we will no longer ask student to do test-driven development and record development process with ESR. In turn, students can choose any development method that works best for them and their development process will be instrumented by Eclipse sensor only.

3.3.5 Data analysis

Test-Driven Development recognition with Zorro

Hackystat Eclipse sensor collects development activity data and send them to Hackystat server automatically. A Hackystat project can be defined for each student to recognize test-driven development with collected development event data. Figure 3.4 demonstrates Zorro recognition results for project "StackWithTDD". Zorro divides the software development stream into episodes


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

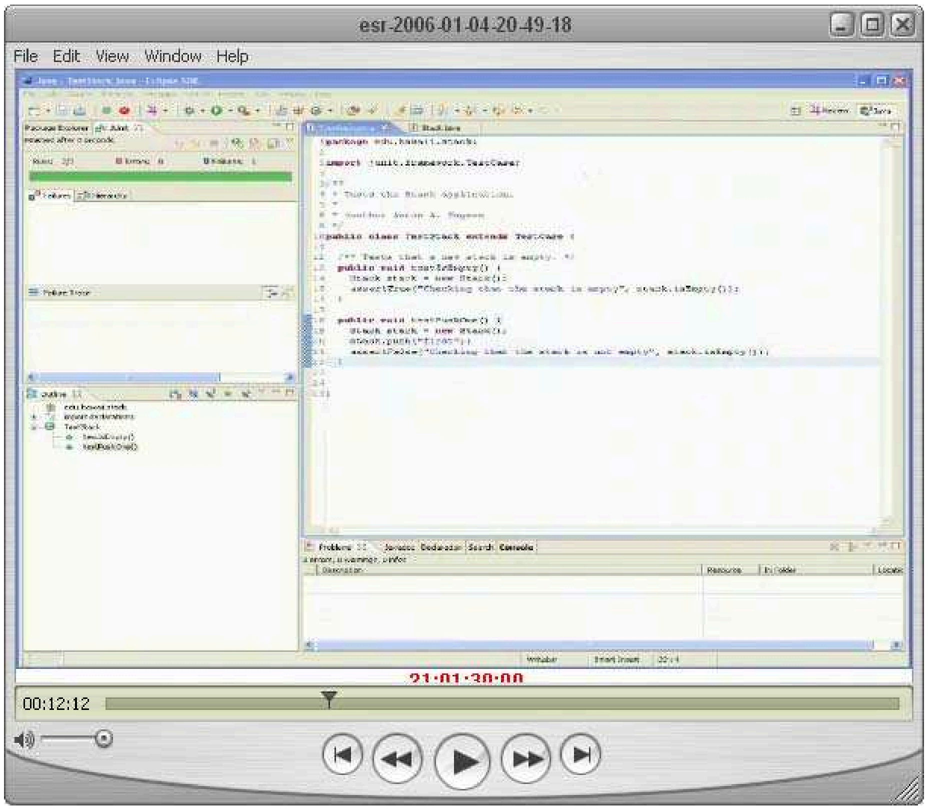
Episode Classification	Episode Actions			
(tdd, 2)	01/01/2006 23:29:20	TestStack.java	ADD IMPORT	import junit.framework.TestCase
	01/01/2006 23:29:21	TestStack.java	MOVE CLASS	edu.hawaii.hongbing.tddstack --> TestStack.java
	01/01/2006 23:30:03	TestStack.java	ADD METHOD	void testEmpty()
	01/01/2006 23:30:54	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

Figure 3.4. Zorro Recognition Results

and reports the recognition results of episodes on left column in values—"tdd", "tld", "refactor", or "validation". Developer behavior data drived from development events and metrics for test-driven development inference are displayed on the right column. Each activity includes timestamp when the activity occurs, file that it is associated with, activity type, and supplemental metric data.

Development process video analysis

In the development process video analysis, we will play the collected videos with Quick-Time player, write down the script of the development process with a book keeping tool such as Microsoft Excel, and compare them against development event data collected by Zorro for validation. Figure 3.5 is a screen copy of the development movies. The main screen is Eclipse window which records development activities, and a time line track is attached to the movie at the bottom for data synchronization. Table 3.1 illustrates how we use Excel to do book keeping and Zorro validation. One by one comparison between bookkeeping data and Zorro developer behavior data will give us insights whether Zorro data collection mechanism is correct and whether the collected data are good enough to infer test-driven development. A beauty of the video analysis is that it can give us insights how software process is executed by developers, with which we can not only validate Zorro but also find ways to improve it.



```
package edu.hawaii.stack;

import junit.framework.TestCase;

/**
 * Tests the Stack application.
 *
 * @author [illegible]
 */
public class TestStack extends TestCase {

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

  public void testPushOne() {
    Stack stack = new Stack();
    stack.push("first");
    assertFalse("Checking that the stack is not empty", stack.isEmpty());
  }


}
```

Figure 3.5. Video of Development Process

Table 3.1. Zorro and ESR video comparison data sheet

Subject id: XXX-XX-XXX		Movie file: esr-YYYY-mm-DD-hh-mm-ss.mov				
Episode #	Zorro	Video	From	To	Activity	Annotation
1	(tdd, 1)	(tdd, 1)	23:28:32	23:28:43	New project	Create new project HelloWorld
			23:28:45	23:29:21	New TestStack	Create unit test TestHello in package edu.hawaii.ics.rainer
			23:29:55	23:30:08	Add testAloha	Add a empty test case for Hello
			...	...	...	...
			23:32:26	23:32:32	Run TestHello	Test passes
2	(tdd, 1)	(tld, 1)	...	...	...	...
			...	...	...	...

“Test infected” claim verification with telemetry

Software telemetry [23] is a new approach of software project management and telemetry report can give retrospective and in-process analysis on software development process. Zorro defines three telemetry reduction functions:

- *MemberTestDriven* Computes Test-Driven Development percentage of a project member over the given time interval.

- *ProjectTestDriven* Computes Test-Driven Development percentage of a project over the given time interval.

- *ZorroEpisode* Reports number of episodes of a specific episode type, or total number of test-pass episodes over the given time interval.

By combining telemetry reports and students survey, we can verify whether students get “test-infected” with strong evidences.

3.3.6 Anticipated Results

- Zorro collects development necessary behavior data correctly to derive test-driven development.

- Zorro recognizes test-driven development in acceptable accuracy.

- Students get “test-infected” and they continue using test-driven after Zorro validation study.

3.4 Study in TDD community

Case study with students in classroom setting is insufficient for generalization of conclusions as the matter of fact that students are novice programmers and TDD developers. I plan to conduct an off-site case study of Zorro by recruiting experienced TDD developers from test-driven development community, the user group of TDD.

3.4.1 Goals and hypotheses

The objective of this study is to validate Zorro on developer behavior data collection and test-driven development process recognition with experienced TDD developers. Beyond this objective, I also aim at improving test-driven development recognition capability of Zorro in actual software development process. Propositions with regard to these goals are:

- Hypothesis 1. *Zorro can collect enough developer behavior data to recognize test-driven development.*
- Hypothesis 2. *Zorro can correctly infer test-driven development with the collected developer behavior data.*
- Hypothesis 3. *Zorro helps experienced tdd developers stay on the track of test-driven development.*

3.4.2 Subjects

Test subjects are experienced TDD developers from the community of test-driven development, user group of [53]. On the account that I only have limited experience on recruiting experienced/professional developers as test subjects, it will be wise to take Benestad's advice [6]:

> first, practical constraints must be defined when defining the target population of software developers; second, participants must be offered flexibility and value using a planned communication strategy; third, high professional and ethical standard must be employed.

There principles are largely for recruiting professional developers from software organizations for controlled experiments, but we can borrow the idea to assist recruiting test-driven development community members who likely belong to some software institutes.

1. *Practical constraint must be defined when defining the target population of software developers.*

   The constraint is that participants must be okay with Java programming in Eclipse IDE in Windows OS. JUnit 3.8 is recommended.

2. *Participants must be offered flexibility and value.*

   Test subjects have the flexibility to choose the problem to tackle in test-driven development to their conveniences. Process Instrumentation and data collection overhead are maintained in the lowest level. As the payback, participants can use Zorro in their organizations for test-driven development process improvement and I will provide technique support. Appendix D is the participation solicitation letter for recruiting experienced developers from test-driven development community.

3. *High professional and ethical standard must be employed.*
   Clearence of human subject exemption was already granted by University of Hawaii Committee on Human Studies. Participants' individual skills will not be evaluated, instead their inputs will be used to evaluated Zorro software system. The participantion of this study is voluntary and test subjects can withdraw from the study at any time. A consent form (appendix E) will be signed by participants and they can keep a copy of it for reference.

3.4.3 Experiment setting

This study offers the flexibility to allow participants work off-site at their own working environment. Basic requirements are:

- *Windows-based pc(≥1.8GHz CPU and ≥512MB RAM*)
- *JDK 1.4 or above*
- *Eclipse*
- *JUnit 3.8*
- *Hackystat Eclipse Sensor*
- *Eclipse Screen Recorder*
- *Quick Time*

Java 5 features of JUnit 4.x are depressed for this study because Eclipse sensor can not tell on-going metrics of Java 5 style code. In order to help participants configure the test environment, we wrote a DocBook chapter for reference in HTML at [64].

3.4.4 Data collection

The Java development in Eclipse IDE in test-driven development will be instrumented by Hackystat Eclipse IDE and ESR. Once installed, Hackystat Eclipse sensor can collect data unobtrusively. Developers start ESR recording by pressing green button and stop it by pressing red button. It takes test subject's manual intervention to send the recorded movie file to reseachers for data analysis.

3.4.5 Procedure

Recruition of test subjects

A participation invitation email will be sent to test-driven development user group to recruit test subjects. The first contact email will briefly address objective of this study, introduction of Zorro software system and what participants will do in the study. If some people are interested in participation, I will send them consent form and instruction guideline for the study.

Test environment setup

Participants will install Eclipse IDE if they've not done it yet, and process instrumentation utilities — Hackystat Eclipse sensor and Eclipse screen recorder (ESR) under help of Zorro user guide[64].

Development and data collection

Developers can choose the problem they want to work on either from the selected problem sets or elect their own software development in TDD.

- A well-known problem: stack, roman numeral or bowling game.
- Another interested problem: money, sudoku, or spreadsheet.
- 1-3 hours' personal software development in TDD.

The development process is instrumented by Hackystat Eclipse sensor and ESR for data collections. Developers will send their recorded process video to me using email for data analysis and answers to a short survey on usfulness and usability of Zorro software system.

3.4.6 Data analysis

Zorro validation analysis

We will be conducting similar analysis as in classroom case study 3.3 to validate Zorro's data collection and TDD recognition capability.

Usefulness and usability of Zorro

Feedback from experienced TDD developers is helpful on identifying issues regarding to test-driven development discipline in practice and verifying hypothesis 3 made on Zorro usefulness.

Appendix A

Consent form of classroom case study

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

As part of this research, you will be asked to develop or modify a program and part of your course project using test-driven design practices and the Eclipse development environment. While you are working on the program, we will be collecting 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, and so forth. The development or modification activity will typically take no more than a few hours of work, depending upon how consistently you work on it.

There is no “right” or “wrong” way to do software development in this research. Just develop the software as best you can. We are interested in observing what real programmers do when working in a test-driven development setting.

The data that we collect will be kept anonymously, and there will be no identifying information about you in any analyses of this data.

Your participation is voluntary, and you may decide to stop participation at any time, including after your data has been collected. If you are doing this task as part of a course, your participation or lack of participation will not affect your grade.

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.

Thanks very much!

____

Your name/signature

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

Appendix B

Pre-survey on test-driven development

Your email: ....

Please circulate your answers to the following claims made on test-driven development.

1. Test-driven development helps me develop code in less time:
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

2. Test-driven development drives me understand requirements and specification better.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

3. Test-driven development improves code quality.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

4. Test-driven development helps to yield simple design.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

5. Test-driven development saves time on debugging.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

6. Test-driven development is more effective than other methods.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

7. I tried to do test-driven development all the time.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

8. It is hard for me to get used to test-drive style programming.

(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

Appendix C

Post-survey on test-driven development

Your email: ....

Please circulate your answers to the following claims made on test-driven development according to your experience.

1. Can you estimate how much percent of your development is in test-driven development?
(1) 90-100% (2) 75-90% (3) 50-75% (4) 20-50% (5) less than 20%

2. It is hard for me to do test-driven development?
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

3. Test-driven development helps me develop code in less time.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

4. Test-driven development drives me understand requirements and specification better.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

5. Test-driven development improves code quality.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

6. Test-driven development helps to yield simple design.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

7. Test-driven development is more effective than other methods.
(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

8. I will continue using test-driven development in the rest of my course proejct development.

(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

9 What are the drawbacks of test-driven development with your experience in this class?

. . . .

. . . .

Appendix D

Participation solicitation letter

Hi, Dear TDD developers,

I am writing an invitational letter of participation in a TDD study of validation of Zorro, a software system that infers existence of TDD with development event data.

As members of test-driven community, we are proud of being test-infected and we appreciate the confidence that test-driven development brings to us. But, outside of our community, there are still a lot of doubts and questions on test-driven development. A significant one of them is that it is impossible to do test-driven development all the time, even most of the time. In another word, are we disciplined in doing test-driven development?

In my Ph.D research work, I developed a software system called Zorro to infer existence of test-driven development with developer behavior data automatically in an obtrusive manner(silent event data collection provided by Hackystat http://hackydev.ics.hawaii.edu and http://www.hackystat.org). In this paper (http://csdl.ics.hawaii.edu/techreports/06-02/06-02.pdf), we presented Zorro software system and a pilot study on its validation. Zorro correctly recognizes 90% of development episodes, which are equivalent to iterations of test-driven development.

I am planning to conduct a replication case study on Zorro with experienced test-driven developers. Your experience and expertice on test-driven development will help us improve our understanding on test-driven development and refine Zorro, an open source software that can recognize test-driven development. Please refer to http://hackydev.ics.hawaii.edu/hackyDevSite/external/docbook/ch07s06.html for present analyses provided by Zorro.

We will be very appreciated for your participation in this study. Please email me at hongbing@hawaii.edu if you are interested

Yours Sincerely,

Hongbing

Appendix E

Consent form for experienced develoeprs

Thank you for agreeing to participate in our research on understanding test-driven development practices. This research is being conducted by Hongbing Kou as part of his Ph.D. research in Computer Science 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 practice and Eclipse IDE. While you are working on the program, we will be collecting 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, and so forth. The development or modification activity will typically take an anour to no more than a few hours of work, depending upon how which program you choose to work on.

There is no “right” or “wrong” way to do software development in this research. Just develop the software as best you can. We are interested in observing what real programmers do when working in a test-driven development setting.

The data that we collect will be kept anonymously, and there will be no identifying information about you in any analyses of this data.

Your participation is voluntary, and you may decide to stop participation at any time, including after your data has been collected. If you are doing this task as part of a course, your participation or lack of participation will not affect your grade.

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.

Thanks very much!

____

Your name/signature

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

Appendix F

Problemsets used in validation study

F.1 Stack

This tutorial gives step-wise guideline on how to implement a stack data structure with Test-Driven Development. Stack works in last-in-first-out (LIFO) principle. In a stack, new value will always be pushed onto the top of the stack and the topmost value is always the first one to be popped off from the stack. Stack includes four basic operations: 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* operation returns the topmost 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.

F.2 Roman numeral

Roman numerals are written as combinations of the seven letters in the table **??** (excerpted from [43]). 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:

- VIII = 5 + 3 = 8
- IX = 10 - 1 = 9

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

Table F.1. Roman Numerals

??

- XL = 50 - 10 = 40

You are about to write a conversion program to translate integer numbers 0 - 50 to roman numeral. See more explanation and bi-directional conversion at [44].

F.3 Bolwing game

A single bowling game consists of ten *frames*. The object in each frame is to roll a ball at ten bowling pins arranged in an equilateral triangle and to knock down as many pins as possible.

For each frame, a bowler is allowed a maximum of *two rolls* to knock down all ten pins. If the bowler knocks them all down on the first attempt, the frame is scored as a *strike*. If the bowler does not knock them down on the first attempt in the frame the bowler is allowed a second attempt to knock down the remaining pins. If the bowler succeeds in knocking the rest of the pins down in the second attempt, the frame is scored as a *spare*.

The score for a bowling game consists of sum of the scores for each frame. The score for each frame is the total number of pins knocked down in the frame, *plus* bonuses for strikes and spares. In particular, if a bowler scores a *strike* in a particular frame, the score for that frame is ten plus the sum of the next two rolls. If a bowler scores a spare in a particular frame, the score for that frame is ten plus the score of the next roll. If a bowler scores a strike in the tenth (final) frame, the bowler is allowed *two more rolls*. Similarly, a bowler scoring a *spare* in the tenth frame is allowed *one more roll*. The bonus is only used to calculate the last frame and it won't be treated as a normal frame.

The maximum possible score in a game of bowling (strikes in all ten frames plus two extra strikes for the tenth frame strike) is 300.

Figure **??** and **??** are two bowling scoreboard. Each frame except for the tenth has one little squares in the upper right, where scoring for the frame's own throws is kept. The first throw is written outside the little square; the second is recorded inside the little square; and the cumulative score goes in the big part of the square.

In the score board, a solid square stands for a strike and solid triangle stand for a spare. You should only work on the bowling game model so that it can compute the game score correctly. Please pay attention this program does not require GUI; you write test cases to drive the implementation of the data model.

Bibliography

[1] Scott Ambler. *Agile Database Techniques : Effective Strategies for the Agile Software Developer*. Wiley, New York, NY, 2003.

[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. *Test-Driven Development by Example*. Addison Wesley, Massachusetts, 2003.

[6] Hans Christian Benestad, Erik Arisholm, and Dag Sjøberg. How to recruit professionals as subjects in software engineering experiments. `<http://www.simula.no/departments/engineering/.artifacts/recruitIRIS>`.

[7] Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`.

[8] Cppunit documentation. `http://cppunit.sourceforge.net/doc/1.8.0/`.

[9] Dbunit. `http://dbunit.sourceforge.net/`.

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

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

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

[13] Boby George. Analysis and quantification of test driven development approach. M.sc. thesis, North Carolina State University, 2002.

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

[15] 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.

[16] Junit. `http://www.junit.org/index.htm`.

[17] Testng. `http://www.beust.com/testng/`.

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

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

[20] Philip M. Johnson. Results from qualitative evaluation of Hackystat-UH. Technical Report CSDL-03-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, December 2003.

[21] Philip M. Johnson, editor. *Proceedings of the First Hackystat Developer Boot Camp*, May 2004.

[22] 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.

[23] 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.

[24] Junit test infected: Programmers love writing tests. `http://junit.sourceforge.net/doc/testinfected/testing.htm`.

[25] Cactus. `http://jakarta.apache.org/cactus/`.

[26] 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.

[27] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 273–281, Washington, DC, USA, 2004. IEEE Computer Society.

[28] 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.

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

[30] Jim Little. Up-front design versus evolutionary design in denali's persistence layer. In *Proceedings of XP/Agile Universe*, Raleigh, NC, 2001. Agile Alliance.

[31] 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.

[32] Metrics and the immature software pocess. `http://www.qpmg.com/pfp.php3?page=metrics.htm`.

[33] Mock objects. `http://www.mockobjects.com/FrontPage.html`.

[34] Carleton A. Moore. *Investigating Individual Software Development: An Evaluation of the Leap Toolkit*. Ph.D. thesis, University of Hawaii, Department of Information and Computer Sciences, August 2000.

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

[36] C#.net, nunit. `http://www.nunit.org/`.

[37] Benug workshop on test-driven development. `http://dotnetjunkies.com/WebLog/davidb/archive/2004/09/05/24474.aspx`.

[38] Chidamber & kemerer object-oriented metrics suite. http://www.aivosto.com/project/help/pm-oo-ck.html.

[39] Pair programming. <http://c2.com/cgi/wiki?PairProgramming>.

[40] Fastest developer in the west. http://mikemason.ca/2003/12/03/#029FastestWayToDevelop.

[41] Pyunit - the standard unit testing framework for python. http://pyunit.sourceforge.net/.

[42] Refactoring. http://www.refactoring.com.

[43] Roman numeral. http://www.yourdictionary.com/crossword/romanums.html.

[44] Roman numerals at sizes.com. http://www.sizes.com/numbers/roman_numerals.htm.

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

[46] Sunit. http://sunit.sourceforge.net/.

[47] Software metric. <http://en.wikipedia.org/wiki/Software_metric>.

[48] Test-driven development: Way fewer bugs. http://www.adaptionsoft.com/tdd.html.

[49] Test-driven development. <http://en.wikipedia.org/wiki/Test-driven_development/>.

[50] Tdd explained. http://homepage.mac.com/keithray/blog/2005/01/16/.

[51] Test driven development. http://www.objectmentor.com/writeUps/TestDrivenDevelopment.

[52] Test-driven development is not about testing. http://www.sys-con.com/story/?storyid=37795&DE=1.

[53] Test-driven development user group. http://groups.yahoo.com/group/testdrivendevelopment.

[54] Test-driven development weblogs. http://www.testdriven.com/modules/mylinks/viewcat.php?cid=20.

[55] Your test-driven development community. http://www.testdriven.com/.

[56] Test-driven development articles. http://www.testdriven.com/modules/mylinks/viewcat.php?cid=7.

[57] Test driven development workshop is a go! http://weblogs.asp.net/rosherove/archive/2004/04/25/119764.aspx.

[58] Work guidelines: Test-driven development. http://www.cs.wpi.edu/˜gpollice/cs562-s03/Resources/xp_test_driven_deve%lopment_guidelines.htm.

[59] Test-driven development with junit workshop. http://clarkware.com/courses/TDDWithJUnit.html.

[60] About the return on investment of test-driven development. http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf.

[61] Development upside down: Following the test first trail. http://www.axtelsoft.com/delphi/xp/development_upside_down.pdf.

[62] Unit-testing tools. http://www.testdriven.com/modules/mylinks/viewcat.php?cid=3.

[63] Test-driven development workshop. http://www.industriallogic.com/catalogs/activities/000002.html.

[64] Zorro: Recognizing and evaluating test-driven development. http://hackydev.ics.hawaii.edu/hackyDevSite/external/docbook/ch07.html.

[65] Test-first stoplight. <http://xp123.com/xplor/xp0101/index.shtml>.

[66] Test infected developers anonymous. http://dotnetjunkies.com/WebLog/seichert/archive/2003/12/03/4214.aspx.

[67] vbunit3 unit testing made easy. http://www.vbunit.org/.

[68] Httpunit. http://httpunit.sourceforge.net/.

[69] XP Software download. `<http://www.xprogramming.com/software.htm/>`.

[70] Robert K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Thousand Oaks, California, 2003.