INSPECTING EXECUTION OF BEST PRACTICES IN SOFTWARE DEVELOPMENT WITH MICROPROCESS TECHNOLOGY

A THESIS PROPOSAL SUBMITTED TO MY THESIS COMMITTEE

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

By
Hongbing Kou

Thesis Committee:

Philip M. Johnson, Chairperson

February 26, 2006
Version 1.0.0

Abstract

Adopting and deploying best practices is a common practice in software organizations to improve development capabilities and process maturities. Best practice contains "a set of guidelines and recommendations" for doing something in either software development or business management [7]. For instance, design patterns are depicted in Unified Modeling Language (UML) diagram to illustrate the generically accepted designs to similar software problems. In the development of software engineering discipline, researchers and practitioners came up with best practices waterfall model, software review, extreme programming and aspect programming etc. out of software practices. As one of the most well-known practice, waterfall model plays a vital role in the history of software engineering and it still exists in many modern software projects' development. Best practice helps to yield good software processes and improve software development. Extreme programming, one of the most famous agile process, consists of 12 best practices [23].

Best practice varies from complicated and heavy practice such as waterfall model to lightweight practice such as uniform coding style in a software project. In software engineering, best practices are summarized and abstracted from successful development experiences and they are constantly being improved by practitioners. Meanwhile, a process model may be developed to enforce the best practice discipline as extreme programming shows. Despite its importance, developers and software organizations often ignore and discard best practice intentionally or unintentionally. One important reason for this is up to the nature of best practices – they usually do not have solid theoretical foundation; on the contrary, best practices are very descriptive and narrative. Lacking of strict execution plan gives practitioners the flexibility to interpret and customize best practices in their environments but it also brings uncertainness and vagueness. On one hand, researchers and practitioners do not know how well best practices are being deployed; on another hand, developers are probably not aware of what they are doing. Janzen articulated that "Measuring the use of particular software development methodology is hard. Many organizations might be using the methodology without talking about it. Others might claim to be using a methodology when in fact they are mis-

applying it. ” [20] Without good understanding, mentor and consultation, practitioners will lack the ability to conclude whether they misconduct best practice or not, and it is hard to tell whether discipline is maintained or not as well.

In our research work, we introduced a substantial step forward to help developers and organizations retrospectively inspect software development process with Hackystat[27]. On the top of Hackystat platform, we designed and implemented software development stream analysis (SDSA) framework to evaluate execution of best practices in software development. In SDSA framework, procedure and key steps of best practices are represented as rules to study software process. It measures microprocess to inspect individual developer’s development work on micro level from bottom up starting with development activities. Microprocess is light weight on the contrary to the traditional documentation and management oriented heavy processes such as waterfall model and Unified Process (UP). In my thesis work I will use SDSA framework to automatically measure and evaluate best practice Test-Driven Development (TDD), one of the most well-known extreme programming (XP) practices.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Software organizations and developers are continuously confront the needs to improve their software processes: the software product should meet customers' requirements better; it should have less bugs; development process needs to be more stable and predictable; productivity should be improved to kick the software out of door as early as possible and so on. Unlike other disciplines such as civil engineering and mechanic engineering, software engineering is still too young to have mature software process and quality control process. Software is very cognitive complicated and software development is a very creative process; thus, each software is unique and there are no two projects that are exactly the same. It's challenging to improve software development process.

To a software development organization there are two ways to improve its development process: one is to identify problems in current process and solve the problems internally; another one is to adopt successful practices from others. Internal software process improvement program is often through process quality control and improvement program. It is heavily emphasized by SEI's capability maturity model (CMM) and standard ISO9001. Personal Software Process (PSP) and Team Software Process (TSP) are two processes proposed by SEI to achieve internal process improvement continuously. Adopting best practices from others is another way to improve software process by borrowing others' successful experiences. Usually it comes with the introduction of new technologies in current development process. For example, another development tool can be used because it has elegant features and powerful capabilities; continuous integration can be adopted to maintain a working version and detect problems early before they sneak into the final product; software review can be adopted to improve software quality by peer inspection. Of the two im-

provement programs, internal process improvement is gradual and steady, while adopting external best practice may bring dramatic changes to the development process.

Best practices are from prior successful experiences and they provide a set of guidelines and recommendations to improve software development. Although they proved to be successful elsewhere, they may or may not be appropriate in a new organization. Managers in the new organizations are either persuaded by consultants or inspired by other projects' successful stories to exercise a new best practice. Introducing best practice is more or less a trial-and-error approach, and normally project leaders manage and provide supports to best practice deployment. To software developers, adopting new best practice is passive, and they may have different opinions. Everett Rogers modeled new technology adoption process as adoption curve [43], in which there are five kinds of adopters when a new technology is introduced: innovators, early adopters, early majority, late majority and laggards. When a new technology is introduced there is always the learning process too. Developers may have slow start at the beginning before they understand the best practice well. Adding these factors together, there will have a lot uncertainness when a new best practice is exercised. Studying and evaluating the applicability of a new best practice needs thorough consideration and understanding of development process. It is fairly reasonable to conclude that the evaluation process will be more error-prone if the practice itself is hard to be understood or it has high discipline requirements.

Because of the nature of software development, it is expensive and cumbersome to do qualitative empirical software best practice and process research in software engineering. Good tools are demanded to facilitate empirical software engineering research.

1.2 State-of-art of Empirical Software Process Study

Considering various type of data in software development Torii et al developed Ginger2 [65] system to support computer-aided empirical software engineering. Ginger2 is an integrated environment to do *in vitro* (laboratory) study by integrating eye-tracking system, 3D motions and skin resistance level measurement altogether. Torii, et al modeled debugging process, analyzed developer's behaviors when a bug was generated and studied collaborative debugging between two developers with Ginger2. It is costly to setup a Computer-Aided Empirical Software Engineering (CAESE) environment like Ginger2 and clearly it will distort test subject's behaviors. The post-experiment data analysis work is massive and labor intensive.

Cook and Wolf implemented a system called Balboa [10] to discover and validate formal software process with finite state machine. Balboa is an event-based model of process actions and it infers a formal model from the events of process execution. Unlike Ginger2, data collection and process discovery are automated and they compared three methods RNet, KTail and Markov in generating finite state machine of the process execution with development event stream [10]. They three generated different finite state machine models to ISPW 6/7 process with some degree of similarities. The models could be accurate to represent the real software development process but it has too many states such that it is extremely hard to interpret the process models and process expert's knowledge are needed. Jensen and Scacchi automated the discovery and modeling of open source project NetBean's development by pulling out the web activities and interactions among developers through email, message board and instant messaging[24]. Instead of inferring software process blindly as Cook they used the prior knowledge of software development to discover software process. They drew out the hyperlinked picture of NetBeans project's requirements and release process to have process fragments and assembled them into PML descriptions. They suggested "a bottom-up strategy for process discovery, together with a top-down process meta model"[24] to automate process discovery and modeling.

In empirical software engineering discipline, survey, experiment and case study are three most widely used research methods. Researchers prefer to case studies because there are too many internal and external variables to control in the experiment. Ginger2 environment implements a complete set of data collection tool to study development activities. Drawback of Ginger2 is that it is limited to *in vitro* setting and it is impossible to apply it on *in vivo* experiment with many subjects to have controlled experiments. The process discovery and automation sounds great but it lacks of the in-depth knowledge of development process. It may take tremendous amount of efforts to have the model, interpret it and find ways to improve software process. Instead of looking for tool support in empirical study, researchers often observation or test subjects' self report to conduct experimentation. Perry, *et al* studied time usage in software development with both direct observation and self-reporting in their research [1]. Researchers often do Personal Software Process (PSP) in curriculum and it is part of course requirements to submit PSP report to ensure students do PSP in their course project development [8, 18]. Müller and Hagner had a case study on Test-Driven Development in University of Karlsruhe[38]. The experimentators asked the test subjects several times during the experiment to make sure they did Test-Driven programming.

The tool support is still not there yet to facilitate empirical researchers to study phenomenon associated with software development. We are aiming to develop a highly automatic

tool to have not only in-depth knowledge of software development process but also the evaluation of process execution to reduce the impact of internal validity problem in empirical software process study. Following sections covers how we designed the software development stream analysis (SDSA) framework to serve get to this goal.

1.3 Software Development Stream Analysis (SDSA)

With my research work I introduce a bottom-up approach to derive and evaluate best practice execution in software development automatically with the help of Hackystat platform [27]. It supplies a quantitative measure to best practice and generates reports to help developers have more stable and disciplined software process. There is almost no human intervention except for installing Hackystat sensors and pulling automated generated report of best practice evaluation.

My strategy is to analyze software development microprocess with software development stream analysis (SDSA) framework. Microprocess is a meta unit of software development. Developer has one task on hand only in a microprocess and all development activities serve to accomplish this task. We can come up with a few example to see what microprocess is. The development activities to fix a bug in software maintenance form a bug fix microprocess. The procedure to implement a feature or story is a development microprocess. In Test-Driven Development (TDD) an red/green/refactor cycle is also a microprocess. Generically, we define microprocess as a group of continuous activities in software development in order to fulfill a certain task. It is up to researchers to decide what activities account for a microprocess based on research purpose.

SDSA has four components data collection, development stream construction, microprocess tokenization and microprocess identification.

1.3.1 Data Collection

SDSA is built on top of Hackystat so it can utilize the services provided by Hackystat[27] to relieve the cumbersome data collection overhead. It already has more than 10 sensors to collect software metrics data from most popular development tools. In Hackystat, software metrics are automatically collected by attaching sensors to development tools, metric data are sent to a dedicated server for data storage and analysis. SDSA is a web-based system to analyze sensor data collected by Hackystat to inspect software execution process.

1.3.2 Development Stream Construction

Hackystat sensors collect development activities, object-oriented metrics, unit test invocation, compilation, commits and so on and stores these data in Hackystat server. We reduce sensor data into development activities, continuous activities make of development stream in SDSA framework. Development stream is a collection of all development activities over a period. Table 1.1 is part of the software development stream whilst I worked on a stack data structure implementation. Stack works on the principle of Last-In-Last-Out(LILO). This development depicts how I

23:43:57	TestStack.java	ADD CLASS	package org.sci
23:43:57	TestStack.java	ADD IMPORT	import junit.framework.TestCase
23:43:57	TestStack.java	MOVE CLASS	org.sci –>TestStack.java
23:44:19	TestStack.java	ADD METHOD	void testEmptyStack()
23:44:38	TestStack.java	TEST EDIT	6sec
23:44:39	TestStack.java	COMPILE	Stack cannot be resolved
23:45:07	Stack.java	ADD CLASS	Stack.java
23:45:07	Stack.java	BUFFTRANS	FROM TestStack.java
23:45:34	Stack.java	ADD METHOD	Stack()
23:45:56	Stack.java	PRODUCTION EDIT	18sec
23:46:02	TestStack.java	BUFFTRANS FROM	Stack.java

Table 1.1. Development Stream of Stack Implementation

implemented stack with best practice Test-Driven Development (TDD). I started it with the creation of test class **TestStack.java** to drive the implementation of stack. Test case **void testEmptyStack()** is added right after because it is an easy task to test and implement. The test class can not compile because object **Stack** does not exist yet. Then I went ahead to implement stack object to solve the compilation error. The development stream empowers us to retrospectively play back detailed development process.

1.3.3 Microprocess Tokenization

Development stream has rich information on software process. It's easy to analyze a small portion of the development stream, but it is not feasible to look up the entire development stream with thousands of development activities. And it might not be necessary to analyze tons of activities as Cook [10] and Jensen [24] in their researches. Nowadays, software development is incremental and iterative, we can study the development process iteration by iteration without bother to look

up massive amount of data. We name this iterative development process as microprocess in which there are only limited number of development activities.

In software development, developers always have processes or best practices to follow on. They define a set of rules and recommendations to regulate software development. Software best practice either defines the order of development activities or adds new tool support to software development. For example, incremental build is the best practice to audit program's change and build system once there are changes to reduce compilation failures; unit test best practice emphasizes that developers should write their unit tests to improve the quality of programs instead of relying on quality assurance help at the later stage in the software development life-cycle. In incremental and iterative software processes, software best practices are repetitive. By separating development stream into iterations we will be able to demonstrate and examine the best practice deployment in each iteration. An iteration is also a microprocess and the separation method is called microprocess tokenization. Tokenizers need to be carefully designed to chop development stream into meaningful chunks.

1.3.4 Microprocess Identification and Evaluation

A microprocess will contain a series of activities with information about best practice execution. We chose rule-based system to identify and evaluate execution of best practice in a microprocess. In rule-based system knowledge is represented in the form of *if...then...* rules[33]. Inference engine in rule-based system can apply best practice rules and recommendation on microprocess activities to look for matched patterns and problems. Figure 1.1 is the conceptual model.

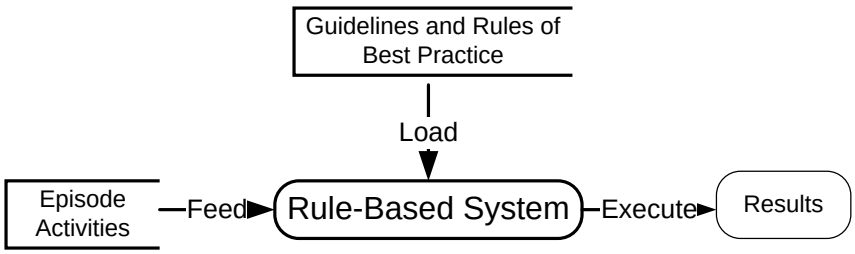

Figure 1.1. Microprocess Identification and Evaluation

Software Development Stream Analysis (SDSA) framework unites data collection, development stream construction, microprocess tokenization and microprocess identification and evaluation together to sustain empirical software process study. In my thesis work I will apply it on

Test-Driven Development (TDD), one of the most popular and well-practiced best practice among 12 Extreme Programming (XP) practices. I chose TDD as the starting point to exercise SDSA paradigm on empirical software process research because of the following reasons:

- TDD is a disciplined practice;
- TDD is iterative and has two simple rules only;
- Research conclusions of TDD are not inclusive;
- Professional software developers embrace TDD.

1.4 Test-Driven Development

1.4.1 What is Test-Driven Development

Test-Driven Development was first coined by Beck and his colleagues when they developed Chrysler C3 project, a payroll system in SmallTalk. In C3 project, they started a new practice to implement component test first and developed it as Test-First Programming, the precedence of Test-Driven Development. Some people prefer to calling it Test-First Design (TFD) because they think TDD is more about design than programming. With TDD developers *(1) write new code only if an automated test has failed; (2) eliminate duplication* iteratively in software development[6].

New code is added only when there is a failed unit test and TDD is to create "clean code that works"[6]. TDD development process is a stop-light pattern, a TDD cycle starts with failed unit test in red light, and it turns green after the code is implemented to make unit tests pass. TDD philosophy "clean code that works" means that developer writes only enough code to make unit test pass without committing too much work. The suite of unit tests gives developer confidence and courage to refactor the implementation code afterward to eliminate the duplication [6].

1.4.2 Widespread of Test-Driven Development

People often use "test infected" [29, 52, 44, 64] to describe developer who sticks to TDD after seriously practices it. The code will be 100% tested because all of the code are driven by unit tests. Their existence makes developers feel free to refactor code constantly in the development process. TDD is getting popular in software industry and some organizations even demand all developers stick to TDD. The extension of xUnit framework on programming languages and

development of mock testing technique make it feasible to test almost everything, which helps to boost the acceptance of TDD.

Test-Driven Development has its own site at http://www.testdriven.com and user group powered by Yahoo[54]. It is a platform for TDD developers to share information, exchange ideas and announce supporting tool releases [56]. A couple of books [6, 4, 31, 19] were published to disseminate testing and Test-Driven Development, and some consulting companies provide TDD training[49, 63, 40, 60, 58] to improve TDD practice in software industry.

1.4.3 Test-Driven Development Research

George and Williams ran through a structured experiment to compare TDD group and control group who developed in waterfall-like process [16]. The research found that TDD group yields superior external code quality compared to waterfall-like controlled group, although TDD group took 16% more time on development. Majority TDD developers thought that TDD is effective and improves programmers' productivity (80% and 78% respectively). In the mean time all developers were assigned as pairs randomly in the experiment, which makes it significantly different from other case studies. Researches in pair-programming indicates that pair-programming has social impacts on developers' behaviors because of pair pressure. Maximilien and Williams conducted another vertical experiment at IBM without pair-programming [34]. 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. There was somewhat productivity decrease in the experiment but developers inclined to continue using TDD in future software development.

Geras et al. designed an experiment to ask participants work on two projects with Test-First and Test-Last in different order in two groups [17]. There are only slightly differences on effort, tests per KLOC, customer test invocations and unplanned failures after delivery between Test-First and Test-Last processes. In the experiment developers worked on their own following the script without management and some of them even failed to submit the working program. Müller and Hagner had a similar case study in University of Karlsruhe[38]. The 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 concluded that TDD does not accelerate the implementation and the resulting programs are not more reliable except that it supports better program understanding.

These case studies give Test-Driven Development a shot and compare it against traditional ad-hoc or waterfall process. The conclusions are variant rather than unanimous. Researches realized that Test-Driven Development has high discipline requirement and used pair-programming,

management, script guideline or verbal confirmation in the experiments. These measures are helpful in removing internal validity problem but the development processes are not measurable and controllable, which may attribute to the differences on divergent conclusions on TDD. It is still challenging to adopt TDD [20] so we do not know how test subjects conducted software implementation differently in the studies. Florac [14] explained that software process can be measured and improved with statistics process control (SPC). My work is to improve the quality of empirical study on best practice and TDD specifically.

1.4.4 Test-Driven Development Microprocess Identification and Evaluation

As I discussed in section 1.3, Test-Driven Development is a good fit to Software Development Stream Analysis (SDSA) framework. In this section I am going to address issues that are connected with TDD.

Data Collection

In TDD, developer works on test code and production code interchangeablly in an iteration, test is executed frequently to guide code implementation and guard code refactoring. Essential data to be included are

- Test invocation and its result
- Test code implement
- Production code implementation
- Refactoring

Compilation error can also be collected to inspect microprocess in small step with compilation failure on test code caused by missing production code. Commit activity data is a plus when version control system is employed.

Test-Driven Development Stream

The SDSA framework need to be plugable so that we can just plug in the interesting activities into the development stream. To Test-Driven Development, we will include unit test substream, edit (on documentation, unit test and production code) substream, refactoring substream and compilation substream, and commit substream if applicable.

Test-Driven Development Microprocess Tokenization

Test-Driven Development is in stoplight pattern. It implies that a microprocess ends with successful unit test execution, which is green like traffic light to mean that the software system is under controlled and there is no failing tests. The green light marks the end of one microprocess so we can design *Test-pass* tokenizer to tokenize TDD development stream. Figure 1.2 is a TDD microprocess example.

03/15/2005 23:54:07	TestStack.java	BUFFTRANS	FROM Stack.java
03/15/2005 23:54:17	TestStack.java	ADD METHOD	void testPopEmptyStack()
03/15/2005 23:56:40	TestStack.java	TEST EDIT	122sec MI=+1, SI=0, TI=+1, AI=0
03/15/2005 23:56:40	TestStack.java	COMPILE	Syntax error on token "}", delete this token
03/15/2005 23:56:40	TestStack.java	COMPILE	Syntax error on tokens, delete these tokens
03/15/2005 23:57:57	TestStack.java	TEST EDIT	65sec MI=0, SI=+2, TI=0, AI=0
03/15/2005 23:58:02	TestStack.java	UNIT TEST	TEST FAILED
03/15/2005 23:58:14	TestStack.java	TEST EDIT	6sec MI=0, SI=0, TI=0, AI=0
03/15/2005 23:58:20	TestStack.java	UNIT TEST	TEST FAILED
03/15/2005 23:58:27	Stack.java	BUFFTRANS	FROM TestStack.java
03/15/2005 23:58:52	Stack.java	PRODUCTION EDIT	15sec MI=0, SI=+3
03/15/2005 23:59:02	TestStack.java	UNIT TEST	TEST FAILED
03/15/2005 23:59:17	TestStack.java	BUFFTRANS	FROM Stack.java
03/15/2005 23:59:27	Stack.java	BUFFTRANS	FROM TestStack.java
03/15/2005 23:59:47	Stack.java	PRODUCTION EDIT	9sec MI=0, SI=0
03/15/2005 23:59:50	TestStack.java	UNIT TEST	TEST OK

Figure 1.2. A TDD Microprocess Example

TDD Microprocess Identification and Evaluation

Test-pass tokenizer partitions development stream into many iterations that end up with successful unit test execution. In term of TDD, a test-pass episode can be either “test-driven” or “refactoring”. The significant property of refactoring is that there is no new test case in refactoring episode. With this character we can classify test-pass episodes of TDD into two categories, “test-driven” and “refactoring”. Moreover, a test-pass episode with test creation may or may not be “test-driven”. Test-Driven Development implies the order of development is to “test a little, code a little and repeat.”[6] such that it will be “test-last” if test is not created before code implementation. Figures 1.3 and 1.4 illustrate different kinds of “test-driven” and “refactoring” episodes respectively.

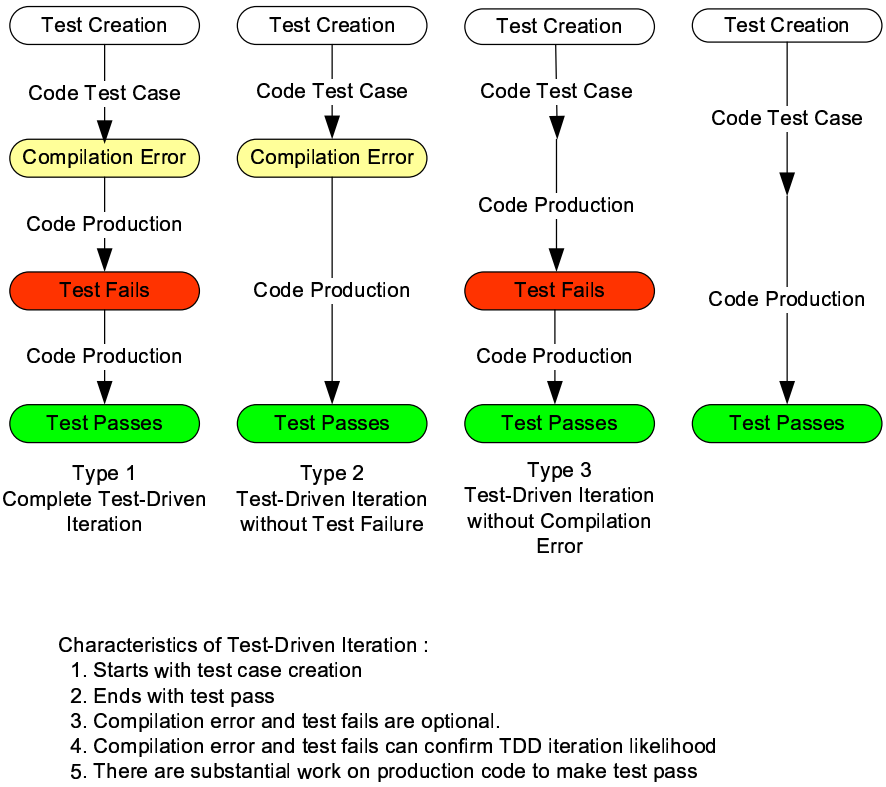

Figure 1.3. Test-Driven Episode Classification

1.5 Research Statement

Software engineering is an empirical discipline and best practice is a very important component in this field. Lacking of good tool support greatly impacts validity of the research conclusions. My research is to design and develop a paradigm to facilitate empirical software research on iterative software processes and best practices effectively by the introduction of software development stream analysis (SDSA) framwork. I am going to study how this framework can support Test-Driven Development empirical study in my thesis work and the weakness of this automation approach. Upon success this research will power empirical researchers both qualitative evaluation and quantitative measure of software best practice executions.

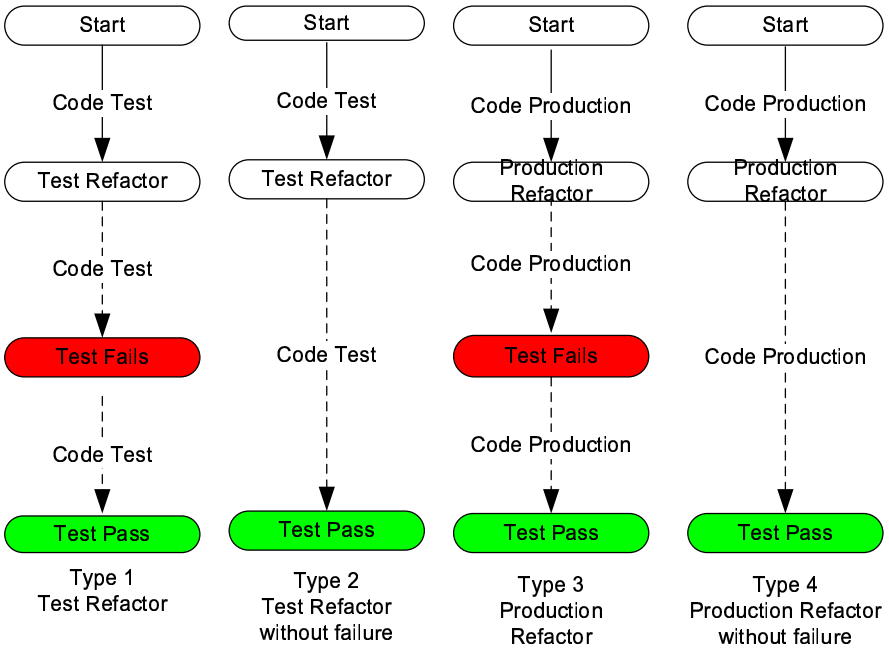

Figure 1.4. Refactoring Microprocess Classification

1.6 Research Methods

1.7 Roadmap

This thesis work is to evaluate best practice execution in software development with development stream. Chapter **??** is the related literature work on Hackystat, software process automation, event stream and Test-Driven Development. The system design structure and implementation details are described in chapter **??**. The framework is tested against Test-Driven Development with a series of experiments in chapter **??**.

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 [5]. 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. [5] 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 [5])

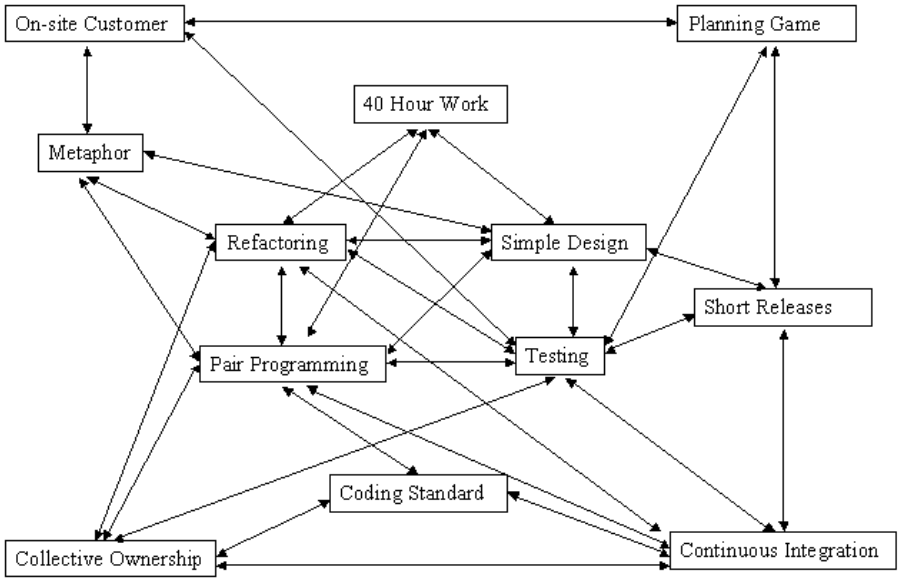

Figure 2.1. XP Support Network

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. [5]

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 [5] 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.[42]

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 [50]:

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 [13]. It has the following structure [6]:

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. [68] 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 ([58, 60, 40, 49, **?**]), technical reports, journals ([57]) and blogsphere ([55]). 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 [59]. It is totally okay to just add implementation stub or simply returns a constant value to fake implementation (p13 [6], p169 [4]).

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 [46]. 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 [6]. These two steps fulfil a complete TDD cycle. "Make it run, make it right." p24 [6] 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 [6]. 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."[6] 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 [6]). 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 [9]. 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 [6, 23] as figure 2.2 shows (Excerpted from page 139 [23]).

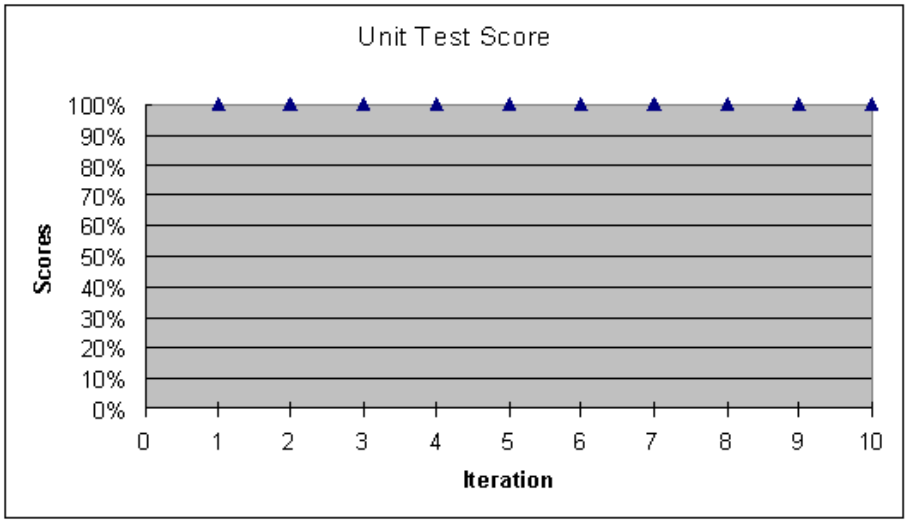

Figure 2.2. Unit Test Scores

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 [15]. 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 [38]. 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 [41]. 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 [52, 2].

Clear Design

XP community argues that TDD is not about test but design [53]. 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 [32]. 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 [6] 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[38]. 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 [16].

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 [51]. 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 [3]

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 [55, **?**]. 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 [6, 29, 64, 44].

Testability

Darach Ennis argued that there are a lot of fallacies blowing around various engineering organization and among various engineers [6].

- 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[67] or Cactus[30]. Mock object and Easy Mock can be used to test some complex operations by providing fake implemention to the real system [36].

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[61]. 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[21], SUnit for Small Talk[47], CPPUnit for C++[11], NUnit for C#.NET[39], VBUnit for VB[66], PYUnit for Python[45] etc. There are also some extention for JUnit to test some complex operation, for instance, HttpUnit for web[67], Cactus for Servlet[30], and DBUnit for database[12]. A new test tool called TestNG is developed to simplify unit test using annotation and configuration file [22].

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.[34] 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[6]. 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 [16]. 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 [38].

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 [48]. 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 [48]. 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 [35] :

- 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 [37].

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 [26] 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[26]. Initial case studies ([25], [28]) 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

Implementation

3.1 Graphic View of Software Testing Process

Software process is constructed by a series of analysis, design, development, testing and debugging activities. It starts from requirement analysis and ends after software products are delivered. Rational Unified Process (RUP), Personal Software Process (PSP), Team Software Process (TSP) and Extreme Programming (XP) are some well-defined software processes. These processes were defined by process pioneers from their best practice and critical thinkings on their development activities. All processes are constructed by a set of rules and advices from requirement analyses to testing. They exist in many kinds of software development organizations and the rules are enforced by development team leaders or managers. Software development process is thought as intellectual, non-repeatable and invisible.

In my thesis work I will focus on studying tests in software development, especially Test-Driven Development to see how tests are being created and exercised by developers incrementally. A test process view is going to be implemented to display and analyze how developers create and execute tests in their development. This tool is called TDPViewer, which stands for Test Development Process Viewer. With this view support I will be able to study development process to see whether developers follow a set of demanded rules.

3.2 Process Pattern and Quantification

Speaking of unit testing execution in software development it could be either test-first as specified by Test-Driven Development, test-last, or hybid mode of test-first and test-last. In

Hackystat we collect both the development activities including implementation, compilation, unit testing and debugging in Eclipse IDE so it is clearly feasible to study how unit tests are implemented in the development process. Software process rules can be used to generate development patterns to categorize how developers do unit testing in the real implementation. One thought here is to design a rule-based agent to study the development pattern [further research to be conducted] to do the categorization.

Chapter 4

Experiment Design and Claim Evaluation

This thesis work introduces software development stream and micro-process to study software development process. Adoption and learning curves of best practice in software development impair process development as well as best practice evaluation. SDSA framework aims at improving both experimental evaluation and practical execution of best practice in software development by recognizing micro-process and quantifying best practice execution.

We will undertake two separated experiments to study correctness and effectiveness of SDSA in software development best practice evaluation. In our study we choose best practice Test-Driven Development as the bechmark to evaluate our SDSA framework. As a well-known best practice welcomed by many developers, Test-Driven Development defines two simple rules only such that experiment subjects can comprehend it easily. Experiment one is best practice micro-process discovery and validation. The goal of this experiment is to evaluate how good SDSA can understand development process and best practice execution. It sets up the benchmark on how system performs to identify best practice and what kind violations it may detect. In experiment two we will use SDSA on a controlled experiment of Test-Driven Development. The controlled experiment will be the replication of already done experiment on Test-Driven Development in other organizations. The SDSA-powered experiment is expected to reveal more detailed information on how test subjects performs instead vaguely assuming best practices are there.

4.1 Software Development Stream Analysis Framework Validation

Automatic data collection, development stream contruction, episode tokenization and micro-process classification are four basic elements of SDSA framework. This experiment is to test whether it can tell the difference when developer execute the best practice and when they do not. Because experiment subjects may or may not have enough skills to write unit tests, not even to test first we will ask test subjects work on three problems to build and exercise Test-Driven Development skills. Task 1 is a simple programming task for us to understand test subject's development skills, developers should only spend 20 to 40 minutes to finish without experiment observer's assistance. An optional introduction of unit test will be provided before subjects work on second experiment. As an additional requirement we ask all test subject to produce more than 90% statement coverage. Experiment assistants will provide technique support to help test subjects achive high coverage. The coverage is evaluated with Clover. We will introduce Test-Driven Development before problem 3 and ask test subjects to work on it following best practice Test-Driven Development. Similarly as previous experiment we ask for 90% above statement coverage too in this experiment.

Objectives

Fine tune the development stream construction, episode tokenization and micro-process classification to measure Test-Driven Development precisely and differentiate development process quantitatively with SDSA framework.

Elements of Experiments

1. *Subjects* Test subjects are undergraduates who have finished one or two 300 level classes already and graduate students. Knowing Java and good understanding of Object-Oriented Programming are two basic skills required.

2. *Problem Sets*

   Problem 1 is a movie listing management system. It should be able to add a move, delete a movie, modify a move and print out list of all movies. Database usage is prohibited for the sake of simplicity and user interface is via DOS command line input. It is optional to have unit tests. (Approx 20-40 min)

Problem 2 is a stack data structure implementation. Elements in the stack are integer objects only. Implemention of stack must be in object-oriented style and it test coverage has to be at least 90%. (Approx 20 min)

Problem 3 is a bowling score system. It should tell the correct score given a set of bowling throws. We require test subjects write programs in Test-Driven Development fashion, and statement coverage must reach 90% at least. In the mean time we want to observe how developers conduct Test-Driven Development. Observes will record down Test-Driven cycles and violation of Test-Driven Development. We will look for an alternative solution to record development process without disturbing development work.

3. *Development Tools* Eclipse IDE is the only developement we will use in the entire development process. Although Eclipse experience is a plus we don't require test subject know Eclipse well to increase our chance to recurit approachable test subjects. Development environment will be set up by examiners beforehand and we only ask subjects do simple tasks only.

4. *Observer and Recorder* It is unsure whethere SDSA framwork can canonically reflect development process in the context of Test-Driven Development such that we include oberver role in our experiments. Observers must know Eclipse well and understand Test-Driven Development well. Upon request observers should be able to answer possibly sophisicated unit test and Test-Driven Development question. An a complement tool we may also monitor and record the development process without interfereing test subjects' development process.

5. *Pre-experiment Survey* Purpose of this experiment is to verify SDSA framework implementation so we want diverse test subject population. We will include undergraduate, graduate, and possiblely professional developers in our experiment. The pre-experiment is informational about test subjects.

6. *Post-experiment Survey* Mostly it will be about the test subjects' understanding of Test-Driven Development from experiments.

Experiment Setup and Procedure

Result Analysis

4.2 Evaluation of Test-Driven Development with Assistenance of SDSA framework

Our motivation to development SDSA framwork is to provide a generic framework to help best practice practitioners to undertake experiment and evaluation effectively. Delicate best pracice such as Test-Driven Development requires high discipline on test subjects, which is hard and expensive to be managed. The light-weight framework of SDSA powered by Hackystat will provide in-process development process data. We will replicate Test-Driven Developments experiments in our study to validate the arguments contradicted by previous studies.

In the classroom setting we will randomly assign student in the class into two different groups. They will have the same lecture and teaching materials. We require everybody must reach 90% cover coverage at least. Group one students has the option to test-last or follow the traditional water-process. Group 2 students are asked to do Test-Driven Development. In order to improve the execution effectiveness we ask developer to write a well-known stack application to improve the understanding of Test-Driven Development via a tutorial session.

Chapter 5

Time Line

Table 5.1. Tentative Timeline

Task	Milestone
Pilot Study of TDD and TLD	Jan 7, 05
Development of TPDViewer	Feb 1, 05
Form Committee	Feb 1, 05
First Round TDD Experiment	Mar 3, 05
#1 Survey on TDD Acceptance	Mar 4, 05
Second Round TDD Experiment	Mar 20, 05
#2 Survey on TDD Acceptance	Mar 21, 05
Third Round TDD Experiment	Apr 15, 05
#3 Survey on TDD Adoption	Apr 16, 05
TDD Study on TDD Adoptors	Aug 8, 05
First Thesis Draft	Oct 10, 05
Submit thesis to committee	Nov 15, 04
Thesis Defense	Dec 3

Chapter 6

Conclusion and Discussion

Appendix A

Acceptance of Test-Driven Development

Last four digits of your SSN . . . . . .

1. I am confident my project is very robust and reliable.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

2. Test-Driven development helps me to produce simple and working design.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

3. I always do refactoring to make my code simple or logic clear.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

4. Test-Driven Development saves my time on debugging.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

5. My development time was increased because of Test-Driven Development
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree
   How much percent increase by your estimation? . . . . . %

6. I did all my work with Test-Driven Development.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

7. I will not do Test-Driven Development if it is not required.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

8. I will recommend Test-Driven Development to my colleagues.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

Appendix B

Test-Driven Development Post-study Questionnaire

Last four digits of your SSN . . . . . .

1. Please specify your development in last session.
   (1) I did all my work in Test-Driven Development fashion;
   (2) I did most of my work in Test-Driven Development fashion;
   (3) Half of my work was done in Test-Driven Development fashion;
   (4) I occasionally did my work in Test-Driven Development fashion;
   (5) I did NOT do Test-Driven Development at all.

2. Compared with last session I feel like my code quality dropped.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

3. I do Test-Driven Development only when I am NOT confident with my program.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree (6) N/A

4. I think writing tests before code implementation is NOT a natural way to develop software
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

5. I think unit test is NOT product. We should leave it to test specialists.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

6. I wanted to do 100% Test-Driven Development but I do NOT know how to write tests for some applications.
   (1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree (6) N/A

7. What are the drawbacks of Test-Driven Development from your view that prevent it from being widely used?

   … ……

   … ……

8. If you still do Test-Driven Development and plan to use it in your future software development please specifye reasons.

   … ……

   … ……

Bibliography

[1] *Trends in Software: Software Process*, chapter Understanding and Improving Time Usage in Software Development. John Wiley & Sons, 1995.

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

[3] Extreme js – js greenwood's web log on architecture, .net, process, and life ... `http://weblogs.asp.net/jsgreenwood/archive/2004/11/26/270503.aspx`.

[4] David Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, Upper Saddle River, NJ, 2003.

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

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

[7] Best Practice. `<http://en.wikipedia.org/wiki/Best_Practice#Software_engineering>`.

[8] William I. Bullers. Personal software process in the database course. In *CRPIT '04: Proceedings of the sixth conference on Australian computing education*, pages 25–31, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

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

[10] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

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

[12] Dbunit. http://dbunit.sourceforge.net/.

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

[14] William A. Florac and Anita D. Carleton. *Measuring the Software Process:Statistical Process Control for Software Process Improvement.* Addison Wesley, Reading,Massachusetts, 1999.

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

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

[17] 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.

[18] Lily Hou and James Tomayko. Applying the personal software process in cs1: an experiment. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 322–325, New York, NY, USA, 1998. ACM Press.

[19] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. Addison Wesley, Massachusetts, 2003.

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

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

[22] Testng. http://www.beust.com/testng/.

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

[24] Chris Jensen and Walt Scacchi. Process modeling across the web information infrastructure. In *Special Issue on ProSim 2004*, Edinburgh, Scotland, 2004. The Fifth International Workshop on Software Process Simulation and Modeling.

[25] 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.

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

[27] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Christopher Chan, Carleton A. Moore, Jitender Miglani, Shenyan Zhen, and William E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.

[28] 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.

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

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

[31] Johannes Link. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, San Francisco, 2003.

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

[33] George F Luger. *Artificial Intelligence*. Addison Wesley, 2002.

[34] 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.

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

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

[37] 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.

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

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

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

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

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

[43] Neil S. Potter and Mary E. Sakry. *Making Process Improvement Work.* Person Education, Boston, MA, 2002.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[61] About the return on investment of test-driven development. http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.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] Test infected developers anonymous. http://dotnetjunkies.com/WebLog/seichert/archive/2003/12/03/4214.aspx.

[65] Koji Torii, Ken ichi Matsumoto, Kumiyo Nakakoji, Yoshihiro Takada, Shingo Takada, and Kazuyuki Shima. Ginger2: An environment for computer-aided empirical software engineering. *IEEE Trans. Softw. Eng.*, 25(4):474–492, 1999.

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

[67] Httpunit. `http://httpunit.sourceforge.net/`.

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