Automation of Test-Driven Development Validation with Software Microprocess

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

Abstract

Test-Driven Development (TDD) is incremental and iterative with small steps in stoplight pattern. This simple principle implies high disciplines during software development and there is no effective method to validate TDD process so far. We designed and implemented Zorro system to automatically examine the microprocesses of TDD with rule-based system support. A pilot study is conducted in the University of Hawaii to study how and to what extent Zorro can understand and validate TDD microprocess, the red/green/refactor iteration.

1. Introduction

Software development is a very complicated process from requirement analysis through deployment. Traditionally this process is big and heavy, it is modeled as waterfall model, incremental process model, evolutionary process model or unified process [21]. Agile alliance panel brought up agile processes to uncover better ways to develop software by embracing individual and interactions, maintaining always working software, constantly collaborating with customers and responding to changes in the development process[1]. How an individual developer or a team implements software system becomes more interesting with the birth of agile processes along with Personal Software Process(PSP) and Team Software Process(TSP) [10].

Developers and software institutes may dedicate to well-defined software processes, or may customize software processes, or may develop software in ad-hoc manner. The execution of software process largely depends on process management and self-discipline. It is hard to measure the use of a particular software process[11], not mention to study how well processes are complied in a software organization. In the case studies and experiments of software processes, researchers often used observation or subjects' oral consensus to ensure process discipline. This manual process validation is inaccurate and very error-prone. In Collaborative Software Engineering Laboratory (CSDL) at University of Hawaii, we designed and implemented Hackystat, an in-process metrics collection system [15]. Hackystat automatically collects development activities and changing metrics of software. With the rich information on development process, we can reconstruct development process to discover and validate process execution in software development. We implemented a system called Zorro on top of Hackystat to automatically discover and validate Test-Driven Development (TDD), one of the 12 Extreme Programming (XP) practices. TDD is repetitive and iterative, each iteration starts and ends with test pass[3]. In Zorro, we put all kinds of development activities together to look for the incremental progress made during software development. It has four components data collection, development stream construction, microprocess classification and TDD microprocess validation. The data collection is powered by Hackystat system [15] to have in-process develop-

ment data seamlessly. Zorro reads in various kinds of development activities to build development stream; splitting it into TDD microprocesses – the red/green iterations; and validate development patterns. Unlike Balboa, we apply the mix of bottom-up and top-down approaches to study TDD microprocess. By assuming that developers follow Test-Driven Development principles we present them in rules to validate the execution process.

Figure 1 is the infrastructure of Zorro system. It collects development data to make development stream and looks for development microprocesses with episode identification. In current study we defined principles of Test-Driven Development as rules in CLIP syntax to examine development microprocess with JESS rule engine system[7] support.

2 Related Work

Basili commented that "we have long known that by using standard engineering and scientific principles, we can improve our profession. What we have not known is how to introduce these principles in a way that will convince engineers to consistently practice them." [10] Software process needs discipline and is continuously improved through "experimentation, measurement and feedback"[10]. In big software organizations, they may hire process experts to record, examine and improve their software processes. Individuals and small software teams are not likely able to afford the process improvement cost as big organizations do so they will have to conduct experiment, measurement and process evaluation by themselves. In PSP and TSP, developers record activity and defect log for process improvement. A pilot study of TSP in Microsoft found that defect rate is dropped from 25 defects/KLOC to about 7 defects/KLOC[28] after adopting TSP. However, the problem is that developers have to stop their on-hand work frequently to record log data with PSP/TSP, which brings lots of overhead to developers and software teams. Johnson worded that "You can't even ask them to push a button."[14] to state the urgency to have automatic data collection tool. Even developers are willing to regularly collect data manually, it is still not enough. Disney and Johnson found various kinds of problems in manually entered PSP data[5], which have great impacts on PSP analyses.

The recent trend in software process is agile process, which advocates incremental iterative development with rapid feedback. Extreme programming [2], one kind of agile process, has 12 practices on planning, designing, coding and testing. Among these practices, Test-Driven Development (TDD) is the most well-known one that is being widely discussed[24], supported[12, 29, 17], practiced[25, 26, 23] and studied[18, 8, 19, 6]. TDD is incremental and iterative. In a TDD microprocess, developers [3]:

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication.

There are many case studies on Test-Driven Development. George et al. found that TDD developers produced higher quality code [8] than controlled groups while Müller et al concluded TDD does not accelerate the implementation and improve the product quality [18]. Other studies drew either positive [19, 6], neutral [9] or negative [20] conclusion on quality and productivity of TDD. In the experiments, researchers gave tutorial and guidelines of Test-Driven Development to test subjects and asked them to follow guidelines in their development process, final projects are collected for comparisons against projects developed by controlled groups. The drawback of this research method is apparent on the fact that Test-Drive Development has high discipline requirement. The experiment controls are not good enough to ensure that the stoplight development pattern[27] of TDD is followed. George et al. stated that one limitation of their experiment is that test subjects had limited training on TDD and pair-programming[8], and this problem exists in all case studies.

Process automation is to build process model out of activity data for process discovery, validation and improvement. Cook & Wolf[4] developed Balboa to collect development event data with finite state machine (FSM) to discover the existence of ISPW 6/7 formal software process. Their work is the complement

to process planning and has the potential to be used for process improvement. Jensen & Scacchi[13] studied requirements and release process of open source project NetBeans by analyzing community activities and describing the process in PML to study the release process execution. Their work emphasizes on software process from macro point of view and it is often neglected how developers implement software substantially. The problem in personal software process and agile process studies is that it relies on individual's self control and discipline [10]. With our work we propose and implement a framework to study the microprocess of software development. Zorro system is the implementation of this framework developed with rule-based system support.

3 Software Development Stream Analysis Framework

Software development stream analysis (SDSA) is designed on the rational of iterative incremental development (IID), that is to say, it is for microprocess. SDSA framework contains four components data collection, development stream construction, microprocess tokenization, and microprocess classification or evaluation.

3.1 Data Collection

SDSA is built on top of Hackystat and it uses service provided by Hackystat[15] sensors to collect in-depth development activities such as file edit, compilation, unit test invocation, refactoring and debug. SDSA reads in software metric data collected by Hackystat sensors and derives development actions out of them.

3.2 Development Stream Construction

Hackystat defines sensor data type (SDT) to abstract metric data collected by sensors. We derive development actions by checking the changes of software metrics and the continuous actions merge together to form development stream.

Figure 1 depicts the conceptual structure of development activity streaming of SDSA. It ends up with development activity stream, which is very similar to time-series or real-time data.

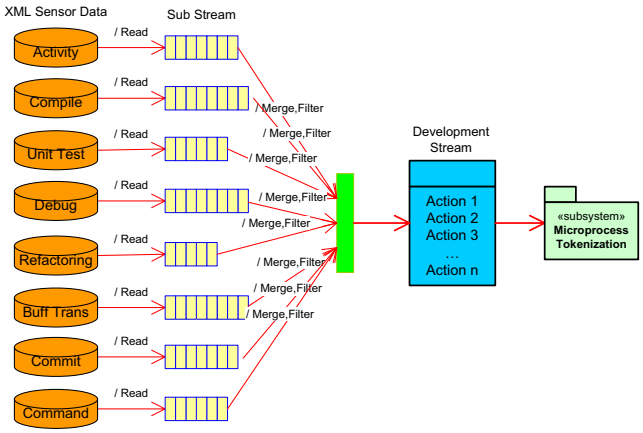

Figure 1. Development Stream Construction

3.3 Microprocess Tokenization

Cook [4] and Jensen [13] studied development stream with finite state machine (FSM) and PML respectively. Nowadays, software development is incremental and iterative, we could and probably should divide development streams into microprocesses to study modern software development process. Depends on what software processes are employed, there are many ways to tokenize the development stream into microprocesses. In our explorative study we introduce *commit tokenizer*, which marks the end of one iteration with commits to version control system, and *test-pass tokenizer* is designed to reflect the rhythm of Test-Driven Development. In situation that one tokenizer is not good enough, SDSA allows the combination of multiple tokenizers.

3.4 Microprocess Classification and Evaluation

In microprocesses, developers conduct development activities following certain methodology so we will be able to find repetitive patterns. In stead of looking for patterns in the microprocesses SDSA employs top-down method to classify and evaluate software microprocess. Knowledge of development methodology is represented in *if...then...* rules with rule-engine support to look for matched patterns. Figure 2 illustrates how to apply rules on microprocess activities.

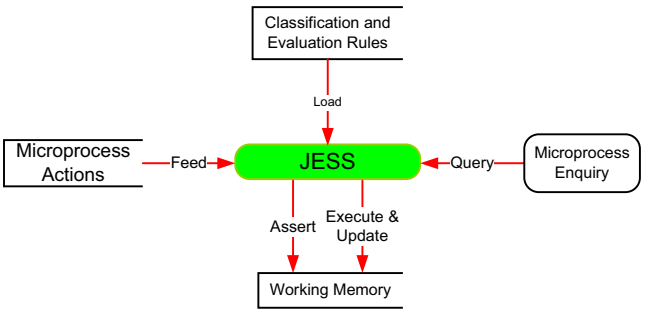

Figure 2. Microprocess Classification System

3.5 Test-Driven Development

In Test-Driven Development (TDD), Developers write failed test first before production code. Implementation is driven by test code and the progress is incremental [3].

> Red/Green/Refactor is the mantra of Test-Driven Development. It implicates the order of programming.
>
> 1. *Red* – Write a little test that doesn't work, and perhaps doesn't even compile at first.
> 2. *Green* – Make the test work quickly, committing whatever sins necessary in the process.
> 3. *Refactor* – Eliminate all the duplication created in merely getting the test to work.

Even though TDD is so simple, there is no better way than pair-programming and informal verbal confirmation to enforce TDD discipline in TDD research as we discussed in section 2. TDD practitioners also do not know how well they are performing in their software development. We categorizes Test-Driven Development *test-pass* microprocesses into *Test-Driven* and *Refactoring* in our study.

3.5.1 Test-Driven

In a test-driven episode developer writes a test based on requirement analysis. The test may not even compile at first because the test target does not exist yet. There should have compilation failure if developer compiles it or project was configured to be compiled automatically. Production code is created to get rid of compilation error. Execution of this test will probably fail when developer invokes it, which is the red bar pattern. The rest work of this episode is to have just enough code to make test pass. This is the scenario of a typical test-driven episode. Even though developers can be required to follow typical test-driven strictly it is lame to have this discipline requirement. In some cases there is no point to let developer do it rigiously.

- Test code compilation will definitely fail because it tests non-existed object or method.
- The production code to make test pass is trivial. Generating a fake implementation to make test fail will be just a waste.

The key to TDD is the test case creation and the substantial work to make test pass. They are the skeleton of a test-driven episode. Depending on the existence of compilation error and test failure, a test-driven episode can be one of them in figure 3.

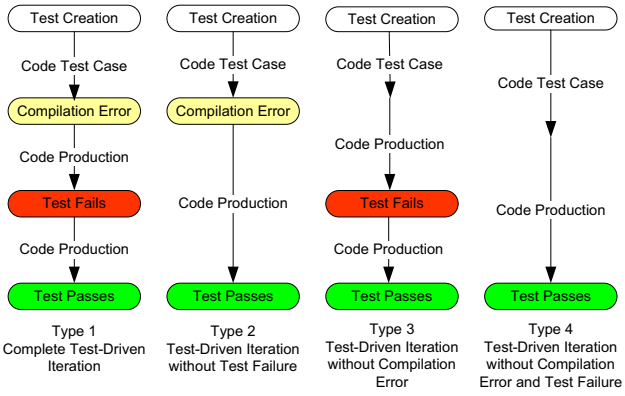

Figure 3. Test-Driven Microprocess

3.5.2 Refactoring

Refactoring is the term describes operation to alter a program's internal structure without changing its external behaviors in software development [22]. New feature is introduced by new test cases in TDD such that a test-pass microprocess is refactoring as long there is no new test. Refactoring episode also has four types. In one side refactor can happen either to test

code or production. On another side refactoring operation may or may not fail the existed tests. Figure 4 depicts the algorithm of this categorization. In types 3 and 4 there may have some work on test code without new test created.

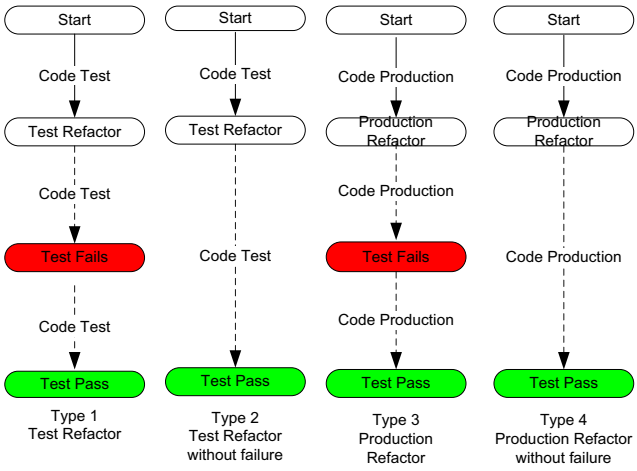

Figure 4. Refactoring Microprocess

3.5.3 Test-Last and Validation

Ideally a test-pass microprocess is either test-driven or refactoring in Test-Driven Development. The allowance is that developer may create multiple test cases for the production code to test various inputs. We call this Test-Last Development contrary to Test-Driven Development in that test code is created after production code. It is also the canonical programming habit of most non-TDD developers. Regression or validation is to run the existed tests to make sure system work well without changing anything, which happens very often in software development.

4 A Case Study on Test-Driven Development

The Software Development Stream Analysis (SDSA) framework views software development from micro-level to support both *in vivo* and *in vitro* empirical software process study. Zorro system is our implementation of this framework to explore its capability on evaluating Test-Driven Development in particular. To study how and to what extent we can automate the TDD evaluation in software development, we conducted a case study at a software engineering research lab in University of Hawaii.

4.1 Experiment Setup

We required test subjects have substantial knowledge of Java programming language and good understanding of unit tests. Eclipse IDE is the development platform we currently have comprehensive sensor support so we asked subjects use Eclipse and install Hackystat Eclipse sensor [16] before the experiment. We observed subject's programming activities and recorded them to compare our observation with the evaluated results from Zorro.

4.2 Tutorial and Guideline

In the experiment we provided a brief introduction and gave instructive guideline on how to implement a stack data structure in Test-Drive Development fashion. Stack is an essential data structure and requirements to stack is well-known to computer science students.

5 Lessons Learned and Discussion

References

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

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

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

[4] J. E. Cook and A. 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.

[5] A. M. Disney and P. M. Johnson. Investigating data quality problems in the PSP. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, pages 143–152, Lake Buena Vista, FL, November 1998.

[6] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30. ACM Press, 2004.

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

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

[9] 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.

[10] W. S. Humphrey. Pathways to process maturity: The personal software process and team software process. <http://www.sei.cmu.edu/news-at-sei/features/1999/jun/Background.jun99.%pdf>.

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

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

[13] C. Jensen and W. 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.

[14] P. M. Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashville, TN, December 2001.

[15] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. 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.

[16] P. M. Johnson and T. Yamashita. Hackystat sensors. <http://hackystat.ics.hawaii.edu/hackystat/controller?Page=help&Subpage%=toc>.

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

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

[19] M. Olan. Unit testing: test early, test often. In *Journal of Computing Sciences in Colleges*, page 319. The Consortium for Computing in Small Colleges, 2003.

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

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

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

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

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

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

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

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

[28] Microsoft's pilot of tsp yields dramatic results. <http://www.sei.cmu.edu/publications/news-at-sei/features/2004/2/featur%e-1-2004-2.htm>.

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