Studying Micro-Processes in Software Development Stream

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

Abstract

In this paper we propose a new streaming technique to study software development. As we observed software development consists of a series of activities such as edit, compilation, testing, debug and deployment etc. All these activities contribute to development stream, which is a collection of software development activities in time order. Development stream can help us replay and reveal software development process at a later time without too much hassle. We developed a system called Zorro to generate and analyze development stream at Collaborative Software Development Laboratory in University of Hawaii. It is built on the top of Hackystat[11], an in-process automatic metric collection system developed in the CSDL. Hackystat sensors continuously collect development activities and send them to a centralized data store for processing. Zorro reads in all data of a project and constructs stream from them. Tokenizers are chained together to divide development stream into episodes (micro iteration) for classification with rule engine. In this paper we demonstrate the analysis on Test-Driven Development (TDD) with this framework.

1. Introduction

Software development is a very complex process from requirement analysis to project deployment. It requires developers understand domain knowledge well and have enough skills to produce high-quality code following a certain process. Traditionally this process is heavy and evaluated by standards such as ISO9001 or Capability Maturity Model (CMM). The recent trend in software process is agile process, which advocates light incremental iterative development with rapid feedback. Extreme programming [1] is one kind of agile process. Unlike plan-driven processes such as Rational Unified Process(RUP), Team Software Process (TSP), extreme programming depends on developers' self-control and internal discipline for process compliance. In eXtreme Programming (XP) pair-programming provides support for process compliance because of pair-pressure and pair-learning [18]. It will entirely depend on individual developer's self control and discipline while pair-programming is not applied. Personal Software Process (PSP), the foundation of Team Software Process(TSP) [9] suffers the same problem as extreme programming because developers have to stop on-hand work frequently to record process data manually or semi-automatically, which lowers data quality as Disney and Johnson discovered [4].

There are many case studies on Test-Driven Development, one of the core component of extreme programming. George et al [7] and Müller [14] et al did researches on Test-Driven Development (TDD) [2], one of the central practices of extreme programming. George et al found that TDD developers produced higher quality code [7] while Müller et al concluded TDD does not accelerate the implementation and improve the product quality [14]. There are other studies drew either positive [15, 5], neutral [8] or negative [16] conclusion on quality and productivity. In these experiments, researches conducted the studies based on their understandings of TDD and test subjects were told to do Test-Driven Development. Final projects were submitted for comparison study with projects developed by controlled group. George et al pointed out that test subjects had limited training on TDD and pair-programming in their study [7]. Test-Driven Development is very disciplined and writing test first is not always an easy task even with the help of xUnit [2] framework. Discipline of TDD was often ignored in the experiments, which may deteriorate the conclusions.

Hackystat[10], an automatic in-process unobtrusive metric collection system, was designed and developed to collect development process data as well as ongoing metrics of the artifact being worked on to lower process data collection overhead. Hackystat sensors [13] are installed in Integrated Development Environment(IDE) such as JBuilder, Eclipse and Emacs, and also build utilities such as ANT, Unix Shell, Version Control System (VCS) and issue tracking system to collect development process data automatically. IDE activities such as file edit, refactoring, unit test execution, debug and build data etc are all interested data to Hackystat sensors. These data are sent to the centralized Hackystat server for analyses such as in-time development management support software telemetry [12] and continuous Goal-Question-Metric (cGQM).

We developed a system called Zorro on top of Hackystat to drill down into fine-grained process data to study micro-process of software development. In Zorro we generate development stream out of sensor data. The stream is tokenized and classified for process compliance study.

2 Related Work

3 System Infrastructure

Zorro is built on top of Hackystat platform. Development activity data is grouped together for development streaming, stream tokenization and episode classification. Development stream is divided into small episodes with the help of tokenizer. Each episode is a series of continuous activities that are isolated by token activities. For instance, test-pass episodes are created when there is a successful unit test invocation. All development activities happened between two continuous successful test invocations belong to this test-pass episode. We evaluate episode with pre-defined rules using JESS [6] rule engine system.

3.1 Development Stream

Hackystat sensors collect both process of software development and state data of projects. To eclipse IDE sensor collects most development activities such as new project, open project, new file, file open, file close, file edit, refactoring, unit test etc. Same kind of activities are grouped together to make sub development streams. They merge together to make development stream as shown in figure 1. Activities irrelevant to the project are filtered out in merging process.

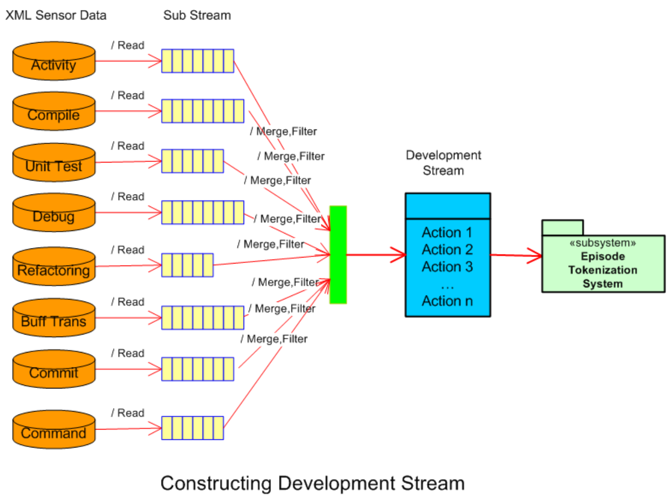

Figure 1. Hackystat Sensor Data Streaming

Development stream is the collection of all development activities occurred in chronological order. It is similar to time-series data and real-time stream because they all consist of a series of data in time order. Development stream is also different from time-series data because it consists of heterogeneous activity, different from real-time steam because each activity has well-defined descriptive data structure. These differences make classical time-series and stream analysis methods hard to be applied on development stream. In our work we developed a tokenization mechanism to divide development stream into micro-processes to simplify analysis.

3.2 Stream Tokenization

Activities in development stream are heterogeneous and each of them can last from milliseconds to hours. The interval between two consecutive activities also varies from milliseconds to hours. They make development stream very irregular and stochastic. Since developers follow different processes development streams vary from developer to developer. To find patterns in programming we implemented tokenization system to divide development stream into micro-processes, which are small programming units. There are four tokenizers in Zorro.

- *Commit tokenizer* ends an episode when it encounters a bunch of file commit activities. It can be used to inspect what developers do before integration.
- *Command tokenizer* ends an episode when there are some consecutive command build activities to deploy system in local environment.
- *Test Pass tokenizer* ends an episode when there are successful unit test invocations. We implemented it to find the iterations in Test-Driven Development.
- *Buffer transition tokenizer* starts an episode when it encounters consecutive buffer transition activities. It sums what developers did to the working buffer.

Basically tokenizers abstract development stream for better understanding of the development process. We applied them on development stream and found that the iterations are either too big such as commit episode, or too detailed such as buffer transition episode. In the mean time an iteration may contain too many kinds of activities to be analyzed. These motivated us to develop tokenizer chain algorithm to iteratively apply tokenizers on the development stream. Figure 2 is the tokenization system data flow. Tokenizer 1 applies on development stream to generate type 1 episodes. They are passed down to be processed by tokenizer 2 when there are more activities except for token activities.

To study Test-Driven Development we put commit, command, test pass and buffer transition tokenizers in a row to work on development stream iteratively. At the end the long development stream will be tokenized into many small episodes that can be classified easily as in figure 3.

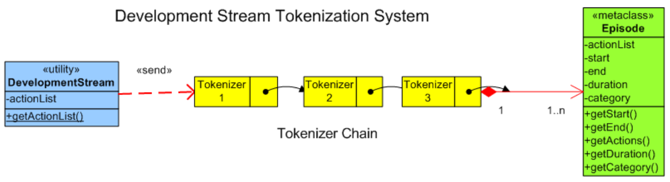

Figure 2. Development Stream Tokenizer Chain

3.3 Classification of Episode

Episode includes activities to accomplish a certain task. It defines iteration in software development such that this approach fits to most modern software development processes because all of them are iterative. In Test-Driven Development each iteration is either a new test case implementation or refactoring if developer follows TDD rational strictly. Because the assumption may not hold we implemented the tokenizer chain to make our solution be general enough to most development streams. If the development stream is not Test-Driven Development we will end up with a bunch of buffer transition episodes.

Test-Driven Development iteration can be elaborated as following[2]:

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

Clearly an TDD iteration (cycle) contains one or more test pass iterations which are either Test-Driven or refactor. In order to fine-tune progress of work we enhanced Hackystat Eclipse sensor to report not only edit work but also the on-going metrics changes such as number of methods, number of statements as well as number of test cases and number of assertion statement to test class. These fine-grained data make it possible to find incremental small changes on programs to deduce micro-process iterations accurately.

Finiate State Machine (FSM) is widely used to study sequential data such as process execution [3] or language. It has the advantage to find the hidden patterns in complicated process. Cook discovered ISPW 6/7 process with RNet, KTAIL, and Markov Method [3] but it ends with a complicated process diagram that needs process expert's knowledge to interpret it. We chose rule-based system over FSM in our study of Test-Driven Development because TDD emphasizes on small and simple iteration instead of complex process and our interest is on process compliance not process discovery. Rule-based system is also stable in the existence of noise.

Figure 3 is the episode tokenization and classification algorithm of TDD analysis.

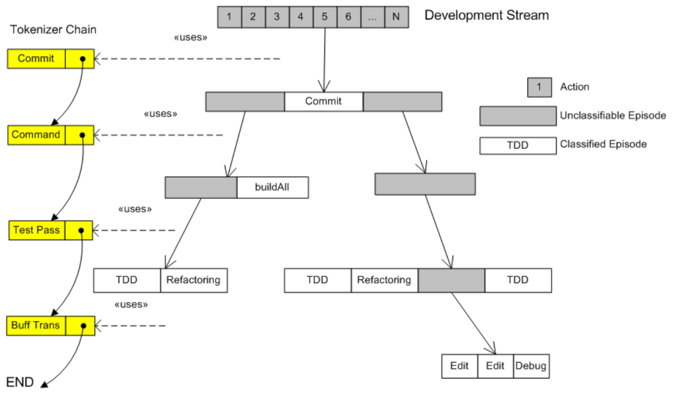

Figure 3. Episode Classification

The tokenizer chain on the left has commit, command, test-pass and buffer transition tokenizers. Commit tokenizer is applied on development stream to make commit episodes. Each commit episode represents one system integration after a new feature is added or a bug is fixed. Commit activities in the episode are stripped off before episode activities are passed to command tokenizer. Test-pass tokenizer will be applied on command episode to generate test-pass episodes. We assert activities of test-pass episode into rule engine to do classification. In figure 3 the left-most command episode contains two sub episodes, TDD and refactor. Buffer transition tokenizer will be applied on non-classifiable test-pass episode. In the end we get episode tree out of development stream with this algorithm.

Developers only make small change to let test pass in TDD. We define rules to check work both on test code and production code to inspect TDD process. Figure 4 is the interaction between Zorro and JESS rule engine system for episode classification with pre-loaded rules. We assert development activities in time order into JESS working memory as facts and run JESS engine to fire up rules.

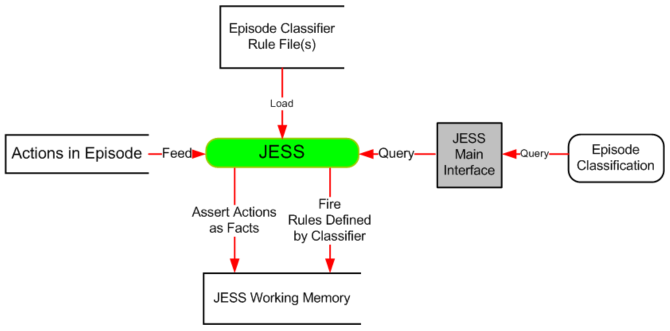

Figure 4. Classification Query

In Zorro we use term "action" to represent development activity, which is the meta operation in development. Here is the action template definition in CLIP language for JESS.

```
;; Action template.
(deftemplate Action
  "Common parts of all actions"
  (slot index) ;Order in episode
  (slot file)  ;Active file
)
```

Index is the order of the action in the episode. All actions inherit this template. We have following actions in Zorro:

- *Documentation Edit* is the edit action on documentation. It has duration inherited from abstract *Edit Action*.

- *Production Edit Action* is the edit action on production code which is java program at present. It defines attributes *method change* and *statement change* from the incremental work.

- *Unit Test Edit Action* is the edit action on unit test. It is similar to *Production Edit Action* but the work is on unit test code. In addition to number of method change and number of statement change it includes number of test methods change and number of test assertion change too.

- *Compilation Action* is not compilation invocation but a compilation error on the active file..
- *Unit Test Action* represents an unit test invocation which either succeeded or failed. Error message will be attached if it fails.
- *Buffer Transition Action* is the active buffer's change during development.
- *Unary and Binary Refactor* are two types of refactor operators. *Add* and *Delete* are unary while *Rename* and *Move* are binary. In addition, refactor operation can be on class, import, field or method.
- *Debug Action* is any debug activity include break pointer set/unset, step into, step over etc.

3.4 Test-Driven Development Episode Classification

Test-Driven Development (TDD) is the way on how to program. Developers write failed test first before production code. Implementation is driven by test code and the progress is incremental [2].

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.

Iterations of Test-Driven Development are usually less than ten minutes. There is no big progress in each iteration except for making test pass. This iteration can be casted into one to many test-pass episodes. Each episode will be either a complete TDD iteration or a portion of it. Episodes of TDD can be either Test-Driven or refactor depends on test progress.

3.4.1 Test-Driven Episode

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 5.

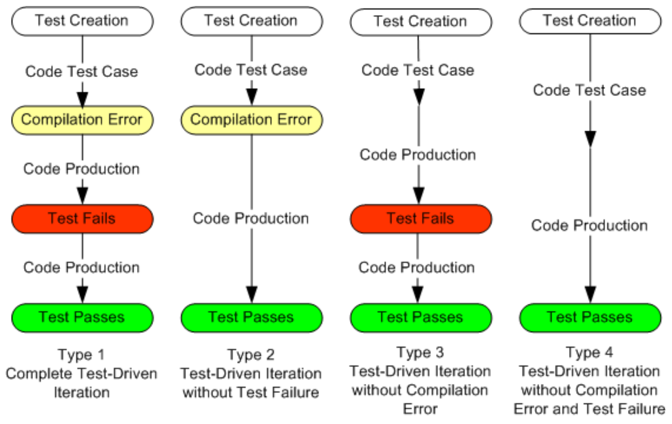

Figure 5. Test-Driven Episode Classification

3.4.2 Refactor Episode

Refactoring is the term describes operation to alter a program's internal structure without changing its external behaviors in software development [17]. New feature is introduced by new test cases in TDD such that an test-pass episode 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 6 depicts the algorithm of this categorization. In types 3 and 4 there may have some work on test code without new test created.

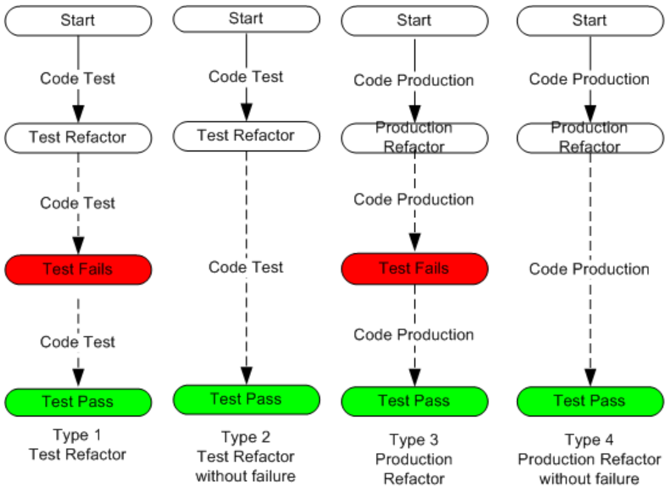

Figure 6. Test-Driven Refactor Classification

3.4.3 Test-Last Episode and Validation Episode

Ideally all test-pass episodes are either test-driven or refactor 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, which happens very often in software development especially in TDD since test cases from TDD serve as regression test suite as well. A test episode is validation as long as there is no substantial edit work on both production and test code.

3.4.4 Complicated Test-Pass Episode

TDD advocates developers write a simple test only each time and write enough code to make failed test pass without committing big bulk of code at once. This is the discipline required by TDD. It's also called Test-First Design because test cases define the interaction of the system with external code delegated by test code in development. In actual project development developers may breach this discipline because it is tedious to run unit tests in every small step. Our survey indicates that developers tend to run tests when they worry that system might break because of the new code. Although it is not welcomed by TDD we can not stop developers making big progress. In our study we claim a test-pass episode not classifiable and complexity high if there are more than two different production files. This kind of complicated test-pass episode is flagged and passed down to buffer transition tokenizer for further investigation.

3.4.5 Classification of Buffer Transition Episode

Buffer transition episode includes developer's activities on the working buffer. When a test-pass episode is complicated we apply buffer transition tokenizer on it to deduce developer's work.

- *Read* – There are only buffer transition activities in the episode.
- *New* – There are new class, methods or fields created.
- *Delete* – There are only buffer transition and delete activities on either test code or production code.
- *Edit* – Edit work on documentation, production code or test code.
- *Test* – There is failed test invocation with possible edit work on test or production code.

4 Experiment and Evaluation

5 Discussion

References

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

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

[3] 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.

[4] A. M. Disney and P. M. Johnson. Investigating data quality problems in the PSP. In *Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT'98)*, Orlando, FL., November 1998.

[5] 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.

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

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

[8] 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.

[9] 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>`.

[10] P. M. Johnson. Project hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis. Technical report, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, November 2001.

[11] 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.

[12] P. M. Johnson, H. Kou, M. G. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Software*, August 2005.

[13] P. M. Johnson and T. Yamashita. Hackystat sensors. `<http://hackystat.ics.hawaii.edu/hackystat/docbook/apa.html>`.

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

[15] 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.

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

[17] Refactoring. `http://www.refactoring.com`.

[18] L. Williams and B. Kessler. The effects of "pair-pressure" and "pair-learning" on software engineering education. In *CSEET '00: Proceedings of the Thirteenth Conference on Software Engineering Education & Training*, page 59, Washington, DC, USA, 2000. IEEE Computer Society.