TEST-DRIVEN DEVELOPMENT RECOGNITION AND EVALUATION

A THESIS PROPOSAL SUBMITTED TO MY THESIS COMMITTEE

DOCTOR OF PHILOSOPHY

IN

INFORMATION AND COMPUTER SCIENCES

By
Hongbing Kou

Thesis Committee:

Philip M. Johnson, Chairperson
...

September 16, 2004
Version 1.1.0

Abstract

"Test-Driven Development (TDD), also called Test-First Design (TFD), is a software development practice in which test cases are incrementally written prior to code implementation[2]". The rational of TDD is to "Analyze a little, test a little, code a little and test a little, repeat." This work is to recognize TDD process consisting many Red/Green/Refactoring iterations and evaluate TDD by analyzing dynamic metrics of artifacts created in TDD process. It will answer whether software developers are doing Test-Driven Development by analyzing on-site activities and whether Test-Driven Development will yield 100% code coverage if developers follow TDD process exactly.

Chapter 1

Related Work

"Test-Driven Development (TDD) is a software development practice in which test cases are incrementally written prior to code implementation."[2] Here test case is the unit test, which is a piece of code written by a developer that exercises a very small, specific area functionality of the code being tested. The rational of TDD is to "Analyze a little, test a little, code a little, and test a little,repeat." The goal of TDD is to write "clean code that works"[1] and it has two basic rules. [1]:

1. Write new code only if an automated test has failed.
2. Eliminate duplication (Refactoring).

They imply an order to do programming task [1]:

1. *Red*
   Write a little test that does not work, and perhaps does not even compile at first.
2. *Green*
   Make the test work quickly, committing whatever sins(for example, constant, fake implementation) necessary in the process.
3. *Refactor*
   Eliminate all the duplications created in merely getting test work.

In book "Test-Driven Development by Example", Ken Beck claimed that red/green/refactor rhythm is the mantra of TDD, once you have an automated suite of tests you will never go back. It gives incredible confidence in your code. He also said TDD will naturally generate 100% code coverage and a coverage tool is not necessary at all if you do TDD perfectly.

Unit test is important in software development because it is easy to check whether the program is functioning properly or not with unit test suites. It allows the programmer to refactor code at a later date, and be sure the model is still functioning properly[6].The automatic unit test suite created in the development process could be used as regression tests too. “A unit test behaves as executable documentation, showing how you expect code to behave under the variation condition you’ve considered.”[5]

In common software practices unit test process is not disciplined and is usually done as an afterthought. Occasionally no tests are created at all, especially with tight schedules. With TDD, before writing implementation, the automated unit tests are written first and they are created on the ground of requirements, which leads to creating better and effective test suites.

However, software developers are conditioned to do afterward tests not test first design thus TDD requires discipline, training and good tool support. Kent Beck suggested the “xUnit” framework when he proposed Test-Driven Development. “xUnit” has many variances and it has been ported to more than 30 languages’ support[7]. But to some applications like Graphic User Interface, database querying etc it is hard to write small unit test to deploy them or not practical. TDD practitioners suggest mocking technique to forge the time consuming operations. Two most famous mock tools are MockObject and EasyMock. With mocked object developer can write test and program faster thus writing and executing unit test suites will not take significant amount of time.

Boby George’s analysis and quantification of test-driven development concluded that both students and professional TDD developers appear to have higher code quality[4]. E. Michael Maximilien and Laurie A. Williams found that defect rate of project IBM Retail Store Solution was reduced by 50% compared to another system built with ad-hoc unit test approach [3] and TDD developers passed 18% more functional black-box tests than non-TDD developers.

Chapter 2

Thesis Statement

I am going to use Hackystat to collect the software development data and analyze the data collected to study Test-Driven Development process. For my thesis work I propose to do the following work.

1. Recognize Red/Green/Refactor(RGR) iteration with the onsite development activity data. Failed execution of unit test or test suite is the start a RGR iteration and a completed RGR iteration ends with green bar. Because of refactorings there may have more than one red bars inside a RGR cycle. We can characterize refactoring by tool-supported refactoring activities, for example, Eclipse supports class renaming and moving etc. The following activities and data are collected to support TDD analysis.

   (a) Editing activities (Including new file, delete file, edit file, rename file, remove file, buffer transition)

   (b) Compile, recompile, build and rebuild activities

   (c) Unit test invocation and test results

   (d) General metrics (For example, LOC)

   (e) Object metrics (Chidamber-Kermer metrics)

2. Evaluate software development practice with the RGR analysis results. After we can recognize and understand RGR iteration we can use it to evaluate how people are doing TDD in practice. We will be able to tell whether TDD practitioners are doing TDD and where/when they commit mistakes.

3. Testability and its effects on TDD

   One possible big adoption barrier of TDD is testability. It may take a big amount of time to execute a test or is too hard to create light-weight unit tests. Is it possible to achieve 100% coverage? Which methods or what kinds of methods and objects can be excluded from test set to achieve 100% coverage.

4. Verifies the claim that TDD generates 100% statement coverage. (Exclude some non-testable cases like main method.) Under what kinds of situation TDD can yield 100% coverage.

5. Test context/proximity study

   Unit test plays an very important role in modern software development practice. One reason is that unit test suite can be used as regression test to confirm system is not broken because of the maintaince changes. If there are tests failed some work has to be done to fix the problem to make them pass. With Hackystat buffer change data we can associate test class with related classes together.

Chapter 3

Data Collection with Hackystat

The IDE for TDD study is Eclipse because it has good refactoring and JUnit supports. Hackystat sensor plug-in for Eclipse is required to collect development activity data. In order to catch the possible fast editing work we will set the state change interval for Eclipse will be much shorter than thirty seconds.

Chidamber-Kermer metrics of the active java file are collected to study dynamic metrics behavior of TDD. Beyond the current metrics we can get super class of the active object is needed for TDD analyses to identify test classes.

Table 3.1. Hackystat sensors and Data to be collected

Hackystat Sensor	Data
Eclipse sensor	Editing, compilation, State Change, buffer transition and refactoring data
BCML sensor (in Eclispe sensor)	Chidamber-Kermer metrics
JUnit Sensor (in Eclipse sensor)	Test Invocation, duration and results

Chapter 4

TDD Recognition and Evaluation

As the leading factor that drives development, unit test should be light weight in TDD so that it will not be the reason that defers development process. Time required to run tests does not take significant amount of time. In TDD each step is subtle and the interval between testing and coding could be just a few seconds in certain conditions. In order to catch this sort of R/G/R transition we need to set the state change interval, defined by Hackystat, to a small value such as 10 seconds or less to record developers' programming activities accurately. Also the existing Hackystat analyses are created based on 5 minutes most active file abstraction mechanism which is coarse and not applicable to TDD recognition and evaluation. The substituted solution is to use raw sensor data to conduct analyses.

4.1 Graphic Representation of TDD

TDD practice contains lot of iterations of RGR iterations, each of them starts with a failed test and ends with green bar. The red bar opens the start of iteration and green bar closes the iteration. Between green bar and red bar there must have some activities associate with the unit test. To turn the bar from red and green developers will work on either implementation or test code. After test passes it is the time to remove the duplication (refactoring). The token of refactoing is that it starts with green and ends with green and it is also possible that there may not have any refactoring at all in a RGR iteration.

As long as we can recognize the RGR iteration of TDD we can represent TDD process graphically as as Hakan proposed.

Requirement:

1. Development activity data include file editing and refactoring activities

2. Unit test invocation and test results
3. Object Metrics such as number of children, line of code

Limitation: We are not sure whether 1 minutes or 10 seconds state change interval is applicable to TDD or not at this moment. The start and end of RGR iteration could be hard to define.

Procedure:

1. Get all sensor working well to include refactoring activities
2. Write analysis to recognize RGR iteration
3. Continue working on the analysis to understand the scenarios of RGR

4.2 Dynamic Coverage

Clover/JBlanket data will be used to study coverage evolution. As it is claimed the statement coverage should be 100% or close to 100% excluding untestable classes.

Requirements: ANT sensor for Clover or JBlanket. And we need to have continuous integration configuration to observe coverage of the developing projects.

Procedure:

1. Create Clover ANT task
2. Configure cruise control or manually check out projects to calculate project test coverage

4.3 Effort Distribution in TDD process

In most situations, one the code is implemented the work is almost done. Even though unit tests are required, sometimes developers may overlook or don't emphasize on unit test especially when the schedule is very tight[3]. From developer's and stakeholder's views unit test are not the end product so that they easily overlook the importance of unit test. TDD emphesize unit tests. "If you cannot write test for what you are about to code, the you should not ever be thinking about coding." L. William's research suggests TDD only takes 16% more effort than the ad-hoc process. With Hackystat we can get activity effort on test, test invocation and implementation to study the effort distribution in TDD practice.

Requirements: Hackystat Activity Sensor

Chapter 5

Thesis Executive Preliminary

As we discussed Eclispe will be used as the targeted IDE to evaluate TDD because it is easy and ready for use, popular in Java development/research community extensible and we have rich experience on using Eclipse. The good thing with a universal IDE is that we can eliminate the bias introduced by environment adaptive issue in the evaluation.

5.1 Hackystat Eclipse Activity Sensor

Hackystat activity sensor for Eclipse was originally written by Takuya, who is a research assistant and Eclipse evangelist in CSDL. It supports open file, close file, state change with the given interval, Chidamber-Kermer metrics of object editing, and unit test invocation with result in Eclipse IDE. In Test-Driven Development we also care the build, file renaming, deletion, creation and removing to observe what happens. The new Eclipse sensor will include the following activities:

1. Editing
2. Compile/build (Including Eclipse build/rebuild/build all/rebuild all)
3. Delete file (.java files only)
4. New file (.java file only)
5. Rename dile (.java file only)
6. Move file (From ... to ...)

We add these activities because they are part of or related with the refactoring. In TDD refactoring is to remove the duplications created in development process. I am not quite sure whether we can solve the refactoring identification problem with these new activities but we will find out with the project progressing.

5.2 What's the test case?

With JUnit framework we believe objects which extend TestCase will be test classes so we could separate test classes and implementation classes by gathering each object's super class. We can go ahead to add one more super class to Chidamber-Kermer metric set. It might be a solution but there is a better one. In TDD practice all tests cases are exercised and the Eclipse JUnit sensor knows which test casese are executed and test results. It means we can tell which object is the test class without matching the activity data and Chidamber-Kermer metrics. (*This solution is kind of lame. Basically we could add file to unit test sensor so that we can get which classes are test classes directly without using super classes.*)

Hackystat JUnit sensor was written for ANT batch execution use such that there is no file associated with test runs. Fully qualified test class names and test case names (test methods) are included in unit test sensor data type. Test class file name as a useful field for TDD analysis will be added into unit test sensor data type.

5.3 Refactoring

As to Kent Beck's explanation and demonistration in Test-Driven Development by Example [1] the refactoring in TDD is to remove the duplication (method, implementation) created. My understanding is that TDD refactoring includes the following cases:

1. Move method around (upward or downward)
2. Delete useless class
3. Eliminate outdated testcases (method)

Class file deletion is easy to be recognized with the proposed new activity sensor. It is not an easy thing to say whether adding/deleting method is refactoring. We will find more after we dig into the domain.

Bibliography

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

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

[3] Laurie Williams E. Michael Maximilien. Accessing Test-Driven Development at IBM, 2003.

[4] Boby George. Master's thesis.

[5] Andy Hunt and Dave Thomas. Pragmatic Unit Testing in Java with JUnit.

[6] Unit Test. <http://en.wikipedia.org/wiki/Unit_test/>.

[7] XP Software download. <http://wwww.xprogramming.com/software.htm/>.