LEARNING EFFECTIVE TEST DRIVEN DEVELOPMENT
Software Development Projects in an Energy Company

Wing Kum Amy Law
TransCanada, 450 – 1 Street S.W., Calgary, Alberta, T2P 4K5, Canada

Keywords: Software engineering, software development methodology, agile, test driven development, automated tests, junit tests, software design pattern, requirement analysis, software maintenance and reliable software.

Abstract: The tests needed to prove, verify, and validate a software application are determined before the software application is developed. This is the essence of test driven development, an agile practice built upon sound software engineering principles. When applied effectively, this practice can have many benefits. The question becomes how to effectively adopt test driven development. This paper describes the experiences and lessons learned by two teams who adopted test driven development methodology for software systems developed at TransCanada. The overall success of test driven methodology is contingent upon the following key factors: experienced team champion, well-defined test scope, supportive database environment, repeatable software design pattern, and complementary manual testing. All of these factors and the appropriate test regime will lead to a better chance of success in a test driven development project.

1 INTRODUCTION

TransCanada is a leader in the responsible development and reliable operation of North American energy infrastructure. TransCanada's network of approximately 41,000 kilometres (25,600 miles) of pipeline transports the majority of Western Canada's natural gas production to key Canadian and U.S. markets. A growing independent power producer, TransCanada owns, or has interests in, approximately 6,700 megawatts of power generation in Canada and the United States.

To support this enterprise, TransCanada Information System (IS) department has delivered many software solutions. Two of the solutions were Project X and Project Y. These projects were different in requirements, customers, budget, and timeline. With varying degrees, both of these teams wrote automated tests before implementation. This practice can have many benefits (McBreen, 2002):

- Identify early mistakes prior to user acceptance testing.
- Reduce time to locate mistakes in the code.
- Analogous to documentation on how to use a class.
- Increase confidence that changes in one place have not broken functionality in another place.

The question becomes how to effectively adopt test driven development. As a programmer on these industrial projects, the lessons learned will be presented to address the following how-to questions:

- How do you start a project focusing on testing first?
- How do you establish a test scope?
- How do you configure an effective test database?
- How do you reuse automated test components?
- How do you guarantee system quality?

2 TRANSCANADA IS PROJECTS

Table 1: Project Profile.

	Project X	Project Y
Programmer	8	12
Test Driven Skills	Adopter	Advance
Project Duration	8 months	3 years
Database	Sybase	Oracle
# of Packages	55	96
# of Classes	1,324	1,560
# of Unit Tests	1,000	4,350

As shown in Table 1, Project X and Project Y were developed by two different teams. Project X was initiated to re-engineer backend components of an existing Java web-based system to enable integration with other systems. The objective of Project Y was to replace a legacy mainframe application with a new Java web-enabled system according to prioritized business functions. These projects shared a common attribute in which both teams adopted test driven development, an agile practice built upon sound software engineering principles.

3 FOCUS ON TESTING FIRST

How do you start a project focusing on test driven development? It is difficult to introduce test driven development to programmers who are not formally trained in this area. Through the lessons learned from Project X and Project Y, a few clues will be provided on the first step towards test automation.

3.1 Project X: Early Adopters

In 2001, TransCanada IS department started to adopt agile practices in several software development projects. The programmers in Project X were the early adopters to apply test driven development in TransCanada.

Without prior experience in using test driven development practices, the programmers in Project X had to start from the beginning on every aspect. They relied on Internet articles and books to explore test driven development techniques. The programmers with fast reading speed gained advantages. They could read, absorb, and apply test driven skills through the self-learning media. The challenge was how to effectively share the knowledge.

To leverage knowledge sharing, the team adopted pair programming practice. Although pair programming is not a part of test driven development, it leads to blending expertise.

Due to personality differences, pairing was not very popular in the team. Some programmers did not passionately believe in test driven development and preferred to write code prior to writing test. They did not have automated tests for all of their code. The diversified team culture reduced the practical application of test driven development.

3.2 Project Y: Team Champion

At the other end of the spectrum, the programmers in Project Y leveraged their collective practical experience and test utility in using test driven development to facilitate their work.

They had previously been exposed to the test driven development and adhered to these practices. The team exercised pair programming to share test driven techniques. As shown in Figure 1, two programmers paired at a computer and monitor with two separate keyboards. The influence of pair programming was to cross train between programmers. With this background, automated tests were indeed written prior to implementation.

Project Y had several keen and experienced experts who built a solid foundation of test framework and set good examples for others to follow. With the supportive team culture, the automated tests typically would not be broken for longer than a day. These experts were the team champions, and they motivated everyone to adopt the practical application of test driven development.

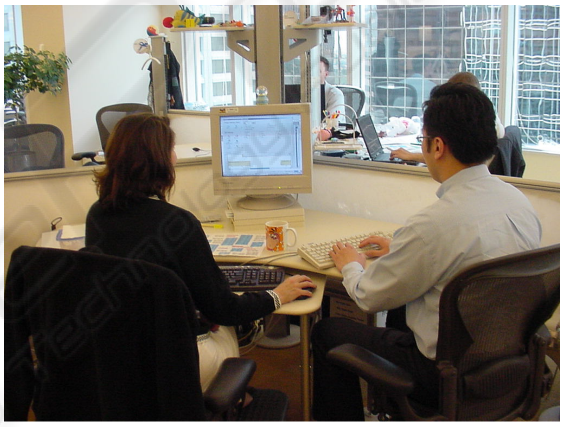

Figure 1: Pair Programming.

3.3 Lessons Learned

Learning test driven development is not easy, but there are a few titbits. A champion in test driven development is a useful guide when a team is challenged. The past experience of the champion could help the reuse and extension of test utilities. Without experienced champion, programmers could become discouraged with tests that were not working for a long period of time. It is simply easier to learn new knowledge from someone who has done it before and passionately believes in it.

And yet, not every project has the luxury to find and fund an experienced team champion. When a champion is not available, reference books and Internet articles are easy-to-access learning media. Furthermore, various research and case studies are recently conducted and documented shaping the best practices. These can be conveniently circulated.

The problem with them is that they do not provide an opportunity for team collaboration.

Test driven development can be enriched through pair programming. Pair programming is like blending colours together on a paint pad, where the colours mix and influence the overall resulting colour. This is a metaphor for the blending of expertise between paired programmers; however, these benefits can be tempered somewhat where personality differences arise.

In most cases, test first guru focus on writing tests first. On the other hand, others may adapt test driven development such that both test and code are implemented in parallel. The key point is that automated tests are indeed written to ensure any changes in functionality at one place would not impact functionality at another place. This continues to provide confidence to customers who see repeatable tests pass.

4 ESTABLISH A TEST SCOPE

Assuming your project applies test driven development, how do you establish a test scope? It is impossible to test everything, and it is also suicidal to test nothing. Therefore, the fundamental principle is test things that might break (Beck, 2000). Several types of automated tests will be discussed.

4.1 Project X: Basic Principle

The programmers in Project X followed the fundamental principle, and they only wrote automated tests for things that might break. If they knew that the code was simple and it was unlikely to break, then they did not write automated tests for it. As early adopters in 2002, they had limited choices in test frameworks. The team began with only unit tests or Junit tests. A Junit test is an automated test to verify a single program or a portion of a program.

Half-way through the project, the programmers realized there was a need to validate the integration of unit components. In response, integration tests or HttpUnit tests were developed. A HttpUnit test emulates browser behaviour and allows automated tests to examine returned pages. Since the integration test concept was introduced at a later stage, the team only implemented a few integration tests. Project X was a relatively simple application, and so about 1,000 automated tests were developed.

4.2 Project Y: Test for Design

By writing tests first, the programmers in Project Y captured customer requirements and scenarios in the tests. They better understood the requirements through the realistic customer's test data. When the tests passed, they knew that they completed the requirements. Therefore, the tests were written for design. The team began with unit tests and integration tests.

Half-way through the project, the team adopted user acceptance tests or Canoo tests. Canoo is an automated test to validate workflow. The Canoo test results are shown in colour-coded pages. The green colour represents tests passed, whereas the red colour represents tests failed. Since these Canoo tests were added at a later time, these tests were written retroactively on existing functionality and based on business priorities. Therefore, the tests were only written for a few main features. With the help of the colour-coded pages, the customers reviewed the tests in order to sign off on a release. Since Project Y had a wealth of business rules, about 4,350 automated unit tests were developed.

4.3 Lessons Learned

There are many types of software tests, such as the unit tests, integration tests, function tests, regression tests, system tests, and acceptance tests (Humphrey, 1989). It is recommended that different types of automated tests be applied to provide a wide coverage for system validation. On the other hand, it does not mean to write tests in every possible case. For example, writing tests to verify every "setter" and "getter" in a domain object is a waste of time.

Before programmers decide to implement another automated test, they should ask themselves if they gain additional business values by having it. Before they decide to stop testing, they should ask themselves four questions (Bertolino, 2001):

- What is the probability of finding more problems?
- What is the marginal cost of doing more testing to detect these problems?
- What is the probability of users encountering these problems?
- What is the impact of these problems to the users?

At a minimum, the programming team should write automated unit tests and integration tests because these tests validate the core business logic, database transactions, and interface of the overall system. Where possible, the team should also develop automated tests to validate end-to-end system behaviour on critical features and perform manual tests to cover other areas.

The customers see the positive results from repeatable tests. This saved the customer time from extensive manual testing and became an overall cost saving. Having said that, test driven development does not deliver software more cheaply than manual approach. Therefore, establishing a test scope and seeking a balance between automation and manual approach is essential to control project cost. This is not a trivial exercise, and this topic may actually be a general interest as a paper into itself.

5 DATABASE CONFIGURATION

After you determine a test scope, how do you configure an effective test database to accommodate the test requirements and ensure test suites do not take too long to run? Database resources, administration overheads, data collision avoidance, and flexibility are key considerations in setting up a test database as examined below.

5.1 Project X: Single Database

The programmers in Project X shared a single Sybase test database. They only needed to refresh one database when there was a change in the database structure. This minimized administration overhead and database resources. However, test data could collide with one another when they executed automated tests against the same database. Hence, the tests might not pass and unwanted test data might remain in the database. The issue became exponential when multiple developers ran the automated tests at the same time as shown in Figure 2.

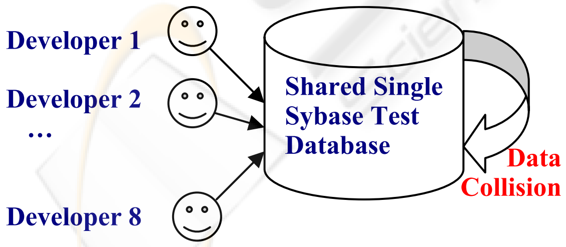

Figure 2: Single Test Database.

Although multiple test schemas in a database could accommodate the concurrency requirement, Sybase database had a technical limitation. Specifically, Sybase did not have the concept of schemas. In order to simulate multiple schemas, the team had to create multiple databases. This was not acceptable to the operational team. Thus, the team shared a single test database.

To reduce the data collisions, the programmers took an advantage of their co-location. They were seated in an open co-located area, where everyone could hear one another. They addressed this database insufficiency by announcing when a developer was about to run the automated tests such that the other programmers would not run the tests at the same time. With discipline, the team could execute all 1,000 automated tests in a single test database under thirty minutes.

Another technique was to use a private database to overcome data collision. Some developers used a private MySql open source database to setup, execute, and clean up automated test data in their own workstation. However, the team ran with the risk that the MySql open source database behaved differently than the Sybase database.

5.2 Project Y: Multiple Schemas

On the other hand, the programmers in Project Y took an advantage of Oracle database to overcome test data collision. Each programmer had a private Oracle database schema in the same database instance. Hence, anyone could create test object, execute automated test, and clean up test object at any time as shown in Figure 4. In some cases, the team used a mock object to simulate results as if a database call was made.

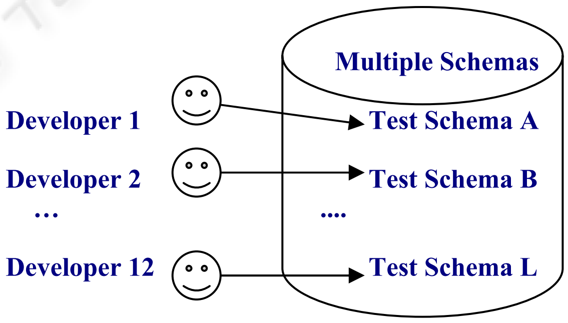

Figure 3: Multiple Test Database Schemas.

To expedite a unit test cycle, the team distributed automated unit test suites among four separate test database schemas. Each test cycle consisted of 4,350 automated Junit tests, and the team can execute each cycle under fifty minutes.

Two additional test database schemas were created to run HttpUnit and Canoo tests at night. This procedure was used to preserve the correct system behaviours without consuming the intensive CPU power during daytime. If a test failed, an email notification was sent out to the team.

As a result of the added convenience, the team incurred schemas administration overhead. The team accumulated up to twenty-five test database schemas. When there was a database structure change, all of schemas required a refresh.

5.3 Lessons Learned

Automated tests could generate many test databases. Some of these databases became unused due to turnover. Alternatively, a pool of test databases can be developed. When programmers are ready to execute the tests, they select available databases from the pool. After use, they can be released it back to the pool.

These databases would not be identical at all times to accommodate different programming needs. The programmers could effectively execute the tests without data collision, maximize resources without increasing large administration overheads, and achieve concurrent programming. With adequate resources, automated tests should be executed every time new code is checked into the central source code repository.

6 REUSE OF TEST COMPONENT

Another interesting topic surrounding test driven development is how do you reuse automated test components? Experts do not solve every problem from first principles. Instead, they built on previous experiences making designs more flexible and ultimately reusable. A number of techniques will be addressed concerning reuse.

6.1 Project X: Design Pattern

The programmers in Project X made use of software design patterns, such as the Factory Pattern and Singleton Pattern. Since 1997, these software design patterns were well documented (Gamma, 1995). However, the maturity of software design pattern in test driven development was in the childhood stage in 2002. There were no documented reusable objects for automated tests. Perhaps it was an excuse for programmers who lacked passion to test driven development, Project X did not use any repeatable software design pattern in their automated tests. Each automated test was always a fresh new test in its own right. This saved time for the programmers by reducing learning curve to existing tests and potentially increased code readability.

6.2 Project Y: Test Design Pattern

The programmers in Project Y applied "Object Mother" software design pattern (Schum, 2001) to reuse the test data setup. The "Object Mother" pattern is a creational pattern. It aims to simplify, standardize, and maintain test object. Using this pattern, test objects can be conveniently used in any automated tests because they are created from public static methods. In case new requirements surface, any future changes can be centralized in the "Object Mother" and propagated to the children test objects. A sample of the "Object Mother" test object is shown below. In addition to "Object Mother" design pattern, the programmers made use of class inheritance such that common test methods could be shared by different automated tests.

```
public static Pipe createPipe( ) {
 Pipe pipe = new Pipe( );
 pipe.setId = unquieRandomNum();
 …
 pipe.setCreatedBy = "ObjectMother";
 return pipe;
}
```

6.3 Lessons Learned

One of the challenges of test driven practices was to design, build, and effectively maintain data setup for the automated testing. The data setup involved the creation of objects required to satisfy the data constraints and test scenario. For example, a repeatable software design pattern should be used, such as the "Object Mother", to reuse test objects. The software design pattern reduced the complexity of individual tests. It also encouraged programmers to reuse test objects in subsequent tests.

Another technique for reusing functionality in automated tests is class inheritance. This approach can group common test methods in the parent class, whereas the children classes can make use of them. Both recommendations enable programming tests less tedious to implement and change. The reusability and consistency outweigh the time invested to avoid learning curve to existing tests. There are now books, training, and internet web sites with many patterns for effective automated tests.

7 SYSTEM QUALITY

People with passion on test driven development claim that automated tests bring various benefits including system quality. So, how do you guarantee system quality? Testing can be used to show the presence of bugs, but never their absence [4]. In spite of how much testing is performed, the team can never guarantee that an application is free of defects. The possible combinations of the input and the execution paths are too many to perform exhaustive testing. Program testing is not a simple process.

7.1 Project X: Manual Test Plan

Since the programmers in Project X did not write automated tests for the entire system, they developed a 35-page comprehensive manual test plan to validate system behaviours. The manual test plan contained test criteria and expected results to guide the users. In response to traditional software development practice, the customers performed an extensive manual user acceptance testing regimen for project sign off.

7.2 Project Y: Customer Sign Off

The programmers in Project Y had a series of automated tests that targeted to validate a wide range of business logics. These tests became a precondition for project sign off. Hence, the customers performed selective manual tests rather than extensive manual tests. As a result, the influence of test driven development practice reduced time from manual testing.

7.3 Lessons Learned

According to PMBOK (PMI, 2004), quality is the degree to which a set of inherent characteristics fulfil requirements. It is documented in requirement specifications. It can be measured by a combination of automated tests and manual tests

The automated tests should be executed as frequently as possible to reduce repetitive tests manually. They fill in the gaps incurred from manual testing. This is especially the case when the team stress level surfaced or human judgment started to degrade.

During software maintenance stage, changes to the automated tests should be made prior to changes to the source codes. Therefore, tests are always kept up-to-date with the specifications and code. This approach requires strict discipline and familiarity to the automated test architecture. Regardless of how extensive automated tests are developed, manual tests must still be performed, to some extent, in order to assure the look and feel of the system. This also increases system usability. As a result, any tuning requirements can be identified and completed prior to the production release.

Testing requires team experience and customer involvement. Therefore, the trick is to know when to stop testing, while at the same time keeping the likelihood of having the application fail post-deployment to under the target reliability objective. A balance of pair programming, code reviews, inspections, traditional manual testing, and user acceptance testing provide a complementary mechanism to test driven development for finding defects and deliver quality and reliable software.

8 CONCLUSION

There is no such thing as instant success in test driven development. However, there are clues which can enable positive results. This paper used the lessons learned from two teams to address questions surrounding test driven development. Any software development team can leverage these lessons learned and develop their own version of test driven development techniques to fit into their unique team environment. Under these conditions, we have a better chance of success in applying test driven development.

REFERENCES

Beck, K., 2000. *Extreme Programming Explained*, Addison Wesley Professional, p 116-117.

Bertolino, A., 2001. *Software Testing, Guide to the Software Engineering Body of Knowledge*, Software Engineering Coordinating Committe, IEEE.

Caputo, W., 2004. *TDD Pattern*: *Do not Cross Boundaries, http://www.williamcaputo.com/archives/000019.html*

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design Patterns,* Addison Wesley.

Humphrey, W., 1989. *Managing the Software Process – SEI Series in Software Engineering*, Addison Wesley.

McBreen, P., 2002. *Becoming a Software Developer Part 2: Test Driven Development with Ruby*, *http://www.informit.com/articles/article.asp?p=26339&seqNum=5*

Mock Object, Project Description and Goals, 2003. *http://www.mockobjects.com*

PMI, 2004. *A Guide to the Project Management Body of Knowledge*, ANSI.

Poppendieck, M., Poppendieck T., 2003. *Lean Software Development – An Agile Toolkit*, Addison Wesley.

Schum P., Punke S., 2001. *Object Mother*, XP Universe Conference.