Technology-Driven Development:
Using Automation and Techniques to Grow an Agile Culture

Hiroyuki Ito
IT Department, Rakuten, Japan

Abstract

I have leaded the new agile project by “Technology-Driven Development”. The word “Technology-Driven Development” has 3 meanings: the mechanism to make the work more effective, to develop cooperative relationships with stakeholders like the business analyst or managers, and to drive learning of the team members by technical practices and methods such as Continuous Integration [5] / Continuous Delivery [8] (hereinafter called the “CI/CD”), TDD (Test-Driven Development) and BDD (Behavior-Driven Development).

I used “Technology-Driven Development” not only as a technology base for developing new smartphone applications, but also as a driver of developing engineering skills for my team consisted of many young and immature members. It made juniors develop software and solve problems as well or better than seniors with support of stakeholders.

In this paper, first I present the concrete mechanism of “Technology-Driven Development” I have introduced to my team. Then the results of learning, cooperation and product development by the method I proposed. In addition, the problems, possibilities and future of it discussed in the latter part.

Keywords

Continuous Integration, Continuous Delivery, TDD (Test-Driven Development), BDD (Behavior-Driven Development), XP

1. Introduction

At the end of April 2013, I started to support one new project as an Agile Coach. The objective of the project was to develop a new smartphone application for Android and iPhone.

1.1. Team and Members

The project team was consisted of a business analyst, UI/UX designers, and Developers. We were able to work closely and collaboratively from the start of the project.

The team members were young and immature. The average age of the team members was under 30. Especially, the average age of the Android developers was around 25. Most of the team members (including me) did not have any experiences of smartphone development. Additionally, the duration of the project was 6 months. Most of the team members did not have any experiences of the “big project” (over a half of year with over 10 members) like this. It was the first time for most of them to work such a long period with over 10 members.

The team was distributed to two locations. There were tons of miscommunications between each locations.

1.2. Problems

We produced the application by reusing and enhancing the existing one. The original application had no automation mechanism for test and release. It took around 1 week to test all features manually before releasing the original one.

The team members did not have any experiences of Agile, XP and Scrum. (So I entered the team as an Agile Coach for supporting them.) There were no technical backbones like CI/CD and test automation. Many agile apprentices tend to introduce agile processes and mindsets at first without any technical backbones and fail. So I needed to add technical backbones from the start of the project.

1.3. What I Did

Initially I started to use XP practices such as CI/CD, TDD and BDD to make our work more effective, and to support agile processes and mindsets. But I found that these practices drove learning and growth of our team members.

We released the Android and iPhone applications successfully. Other teams started to adopt our mechanism partially. A lot of developers and managers in our company expressed considerable interest in our model as the new learning model. So I organized this mechanism and named it “Technology-Driven Development”.

2. CI/CD: Aiming to All Stakeholders

At first I introduced CI/CD rather than test automation or any other agile practices due to the following reasons.

1) Make our work more effective soon
Main developers were so much tired due to manual tests and release operations. Additionally the recent projects also had exhausted them. I needed to make our work more effective immediately so that I make the team sustainable.

2) Easy to implement
As I mentioned earlier, we produced the application by reusing and enhancing the existing

one. There were some batch scripts, however there was no automation mechanism. I was able to implement release automation easily by calling them via Jenkins.

3) Provide values to the stakeholders from the beginning
Many software projects fail to adapt any agile practices without gaining the cooperation of the stakeholders like the business analysts or managers. There were some barriers to gain the cooperation of the stakeholders in the project.

- It took long for the stakeholders to test the application. The stakeholders needed to ask developers to install the application to their devices manually for each change. It took around 5 to 10 minutes for each stakeholders.
- The stakeholders did not have the clear vision and requirements from the beginning because the product was entirely-new. Executable software would help them clarify vision and requirements.
- The stakeholders were not able to get the progress information in a timely manner. Executable application would give stakeholders the progress intuitively.

So I started providing the executable software from the beginning of the project to gain the cooperation of stakeholders.

The figure below shows the mechanism of CI/CD that I implemented.

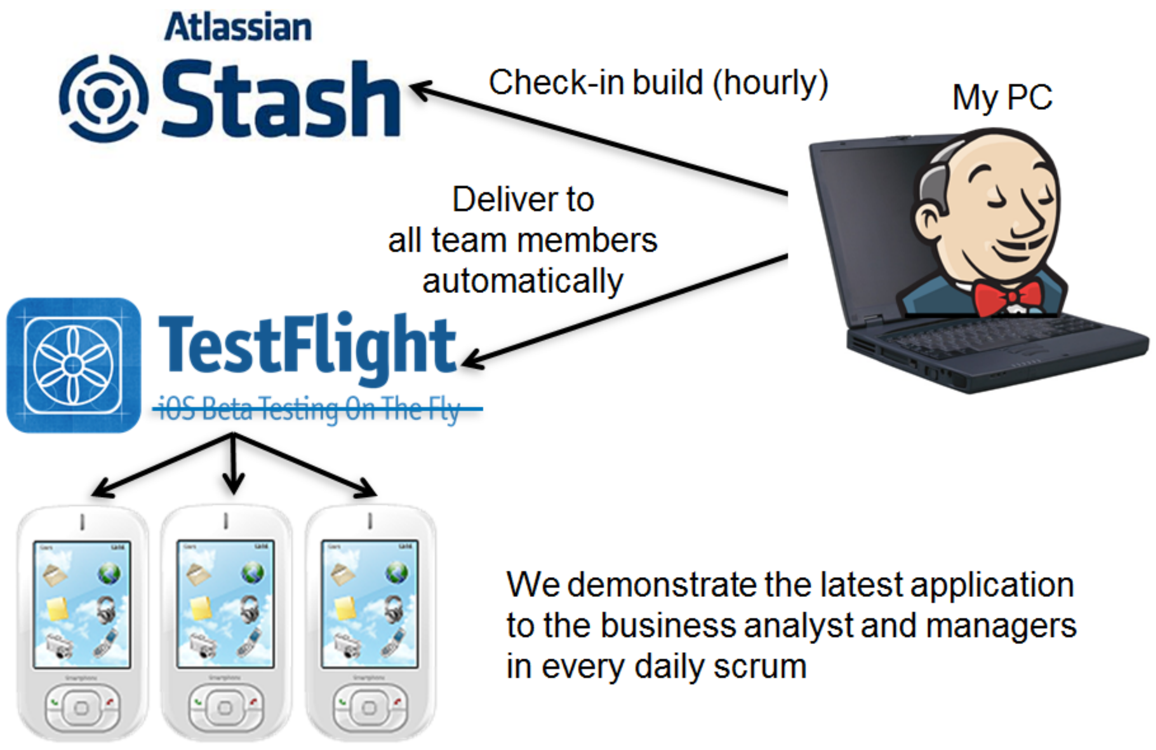

Atlassian Stash
Check-in build (hourly)
My PC
Deliver to all team members automatically
TestFlight
~~iOS Beta Testing On The Fly~~
We demonstrate the latest application to the business analyst and managers in every daily scrum

Figure 1: The mechanism of CI/CD I implemented in my team.

I automated manual release processes via Jenkins and TestFlight [13] (a tool useful for delivering beta applications to the restricted users). The latest application would be delivered automatically to all stakeholders' devices soon after developers committed programs to Git. We demonstrated the latest application at the daily scrum every morning and we were able to get fast feedback from the stakeholders. After implementing CI/CD, I gained the cooperation of stakeholders to proceed agile and automation more.

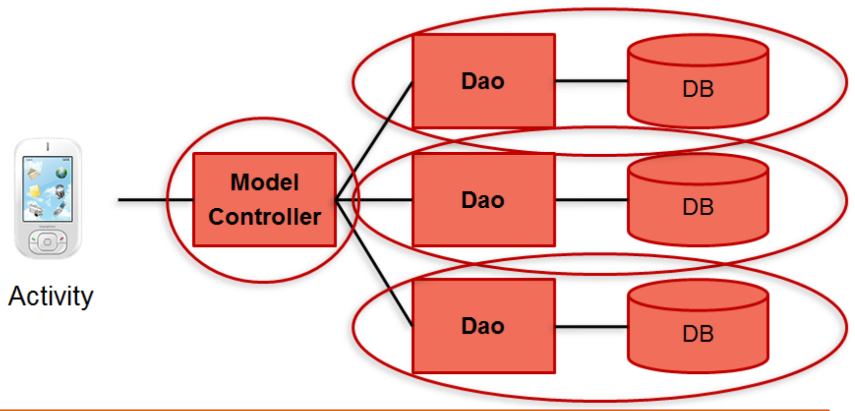
3. TDD: for Making Android Development Easy

After implementing CI/CD, I focused on implementing test automation next. I focused on implementing test automation for Android because Android developers were young and immature compared to iPhone developers. Additionally, it was easier to implement it for Android than iPhone since Android developers and I were in the same location.

Soon I found that it was very difficult to do unit testing of Android application. Android SDK has its own test harness based on JUnit (hereinafter called the "Android JUnit"). Android JUnit requires emulator or device to do unit testing. Android JUnit starts its heavy lifecycle for each unit test cases. It is difficult to use the "Test Double" [11] for component-level test. It takes too long to get useful feedback.

Furthermore, I also needed to grow the Android developers immediately. The average age of the Android developers was around 25. They did not have necessary skills for developing Android application. TDD including Pair Programming was appropriate to grow them quickly.

Eventually I implemented the new mechanism of TDD and unit testing for making Android development easy by Robolectric [12] and Mockito [2]. Robolectric enabled us to do unit testing without any emulator, devices, or lifecycle mechanism. Mockito enabled us to use the "Test Double" for Android development easily. They enabled us to get fast feedback from unit testing.



- Can test each components **independently & separately**
- It takes **1 day** to implement 1 activity set
  (5 times faster than the start of the project)

Figure 2: The result of TDD for Android in my team.

We used the three-tier architecture consisted of UI, Controller and Dao. Before implementing TDD, we were not able to test each components independently and separately.

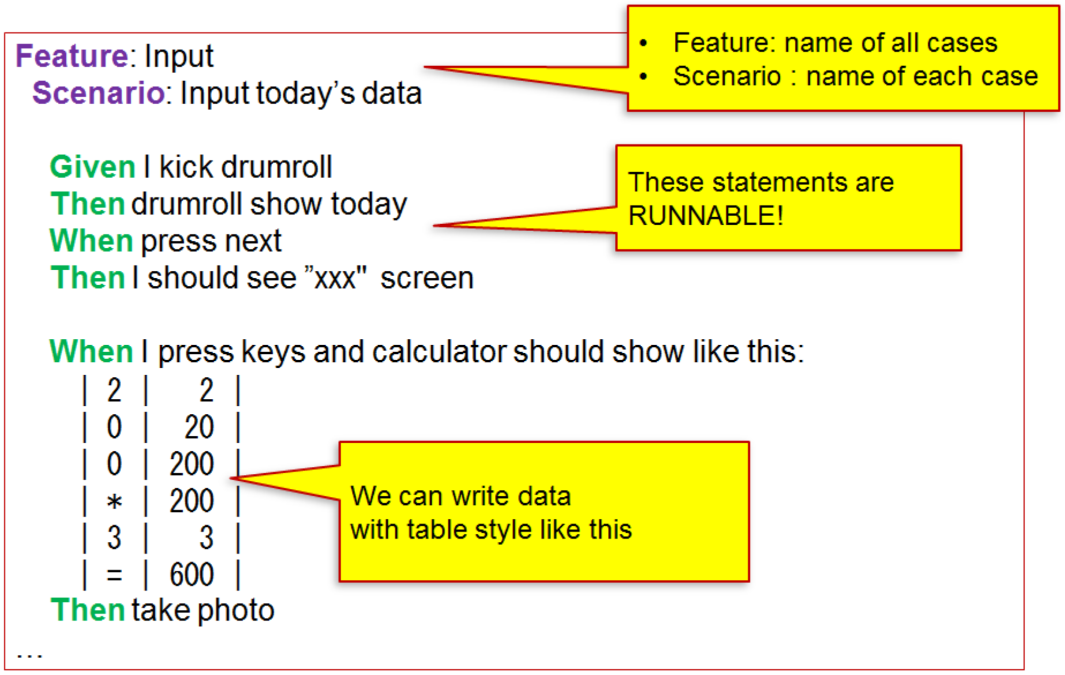
It had taken 5 days to implement one function. After implementing TDD, it took only 1 day due to testability improvement by Robolectric and Mockito. TDD also made it possible to implement all database functions with unit test cases in the early stage of the project. Additionally, all Android developers learned how to develop Android application by TDD and Pair Programming within 1 month. It was really the ”Technology-facing tests that support the team” [3].

4. BDD: for Supporting the Team More

After implementing TDD for Android development in terms of the technology-facing tests that support the team, I needed the “Business-facing tests that support the team” [3] next.

The team members including the business analyst and the UI/UX designers were poor at communicating each other at first. The UI/UX designers created tons of mock-ups and screen transition diagrams. The mock-ups and diagrams had much interaction information, though, they did not give developers what they should develop. We should elicit usecase-level information from all members and stakeholders to develop our application.

I also found that unit testing of user interfaces were ineffective in developing smartphone applications. Screen functions tend to be complicated due to many interactions inherent in smartphone. Tests of screen functions often become usecase-level ones. It was valuable and cost-effective for us to automate tests of user interfaces in terms of functional tests or acceptance tests.



```
Feature: Input
 Scenario: Input today’s data

  Given I kick drumroll
  Then drumroll show today
  When press next
  Then I should see ”xxx"  screen

  When I press keys and calculator should show like this:
    | 2 |   2 |
    | 0 |  20 |
    | 0 | 200 |
    | * | 200 |
    | 3 |   3 |
    | = | 600 |
  Then take photo
…
```

• Feature: name of all cases
• Scenario : name of each case
These statements are RUNNABLE!
We can write data with table style like this

Figure 3: Example of BDD test case.

I adopted the Calabash-Android [7] (wrapper of Cucumber [4] for Android) to implement

BDD for our team. The Calabash-Android enabled us to clarify what we should develop and provide to users. And I used Calabash-Android test cases to elicit ideas and requirements from the business analyst and the UI/UX designers. I drove communications via test cases [1].

5. Results

In the process of implementing CI/CD, TDD and BDD, I gained useful improvement I mentioned above. We were able to run the regression test and detect some degrades by them. Additionally I learned several interesting lessons.

The first lesson I learned is that automation nurtures the team members. If members committed any problematic programs, Jenkins and all tests would tell them what the problem was and how to fix them. The BDD test cases told young and immature Android developers what the users would require and what they should develop. Finally young Android developers developed the required application within 5 months. Their application was faster with less bugs rather than the iPhone application developed by seniors in another location. (It took around 6 months to develop iPhone application and it has more than double bugs compared to Android one.)

The second lesson I learned is that continuous improvement leads other voluntary improvements by the team members. One developer introduced the Genymotion [6], a very fast Android emulator that runs on VirtualBox [14]. It made our BDD and development around 10 times faster than ever before. Additionally, other members started to pull tasks voluntarily. They found and solved problems in advance without any instructions. These devisals were from slack by continuous improvement.

The third lesson I learned is that a series of technical improvements lead solutions beyond existing organizations. After implementing CI/CD, I gained the cooperation of stakeholders such as the business analyst and managers to proceed automation more. One developer taught the UI/UX designers how to use Git and the UI/UX designers were able to commit their artifacts without any handouts.

I have introduced a series of technical improvements valuable to all members and stakeholders. These improvements also have a voluntary improving mechanism that is underpinned by automation. It is the essence of “Technology-Driven Development”.

6. Problems, Possibilities and Future

I gained another useful lessons in the process of introducing “Technology-Driven Development”.

The first problem I faced with is that I was not able to implement TDD and BDD for iPhone development. I implemented CI/CD for iPhone, though, I was not able to implement TDD and BDD due to lack of coaching resource and Mac PC. Distributed location also made it difficult to implement test automation and collaborate via tests.

The second problem is that most of team members did not have sufficient knowledge of quality assurance. Most of them were young and immature. They did not have any experiences of the “big project”. It was their first time to do quality assurance. I implemented test automation mechanism, though, I should also teach them more about the quality assurance’s point of view.

The third problem is that I was not always able to drive by success. In the middle of the project, the business analyst did not accept any scope change because developers always developed what the business analyst wanted against reason in past times. I needed to show the business analyst that unrealistic requirements would lead to no functions in one iteration.

To drive learning more by “Technology-Driven Development”, I need to improve it more and more. I need to learn TDD and BDD for iPhone development, and maybe Windows Phone. I would like to try to add a QA engineer as a Test Coach (like an Agile Coach) from the start of the project. Furthermore, I’m trying the new learning approach named “Fail-Fast Approach”, invoking small failures intentionally to drive learning. Team members are able to cover intentional small failures by “Technology-Driven Development” and get more knowledge to proceed improvements.

7. Conclusions

I have described “Technology-Driven Development”, a learning mechanism by CI/CD, TDD, BDD, and a series of improvements. I have introduced a series of technical improvements valuable to all members and stakeholders. I gained the cooperation of members and stakeholders. These improvements also have a voluntary improving mechanism that is underpinned by automation.

Many agile apprentices tend to introduce agile processes and mindsets at first without any technical backbones and fail. Technical backbones like CI/CD and test automation enable to lead effective learning and elicit voluntary improvements from team members. “Technology-Driven Development” will be a good backbone to support and enhance agile processes and mindsets.

Our young team members released the Android and iPhone applications successfully. Other teams developing smartphone applications started to adopt my mechanism partially. Additionally, a lot of developers and managers in our company expressed considerable interest in “Technology-Driven Development” as the new learning model, regardless of what they produced.

I found that I was able to improve “Technology-Driven Development” more and more. I should learn TDD and BDD for iPhone development. Adding a QA engineer as a Test Coach from the start of the project will make tests more effective. Other learning approach like “Fail-Fast Approach” will lead the team’s growth. I intend to strengthen “Technology-Driven Development” thoroughly to strengthen the team members and company for a better world.

References

[1] Adzic, G. (2011). *Specification by Example: How successful Teams Deliver the Right Software*. Manning Publications.
[2] Code.google.com. http://code.google.com/p/mockito/
[3] Crispin, L., & Gregory, J. (2009). *Agile Testing: A practical guide for testers and agile teams.* Addison-Wesley Professional.
[4] Cukes.info. http://cukes.info/
[5] Fowler, M. (2006) *Continuous Integration*. http://martinfowler.com/articles/continuousIntegration.html
[6] Genymotion.com. http://www.genymotion.com/
[7] GitHub. https://github.com/calabash/calabash-android
[8] Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley Professional.
[9] Kniberg, H. (2007). *Scrum and XP from the trenches*. InfoQ.
[10] Kniberg, H. (2011). *Lean from the Trenches*. The Pragmatic Bookshelf.
[11] Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional.
[12] Robolectric.org. http://robolectric.org/
[13] Testflightapp.com. http://testflightapp.com/
[14] Virtualbox.org. https://www.virtualbox.org/