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

HIROYUKI ITO, DEVELOPMENT PROCESS OPTIMIZATION DEPARTMENT, RAKUTEN, JAPAN

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.

1. INTRODUCTION

Over the few years, a number of software engineers use automation techniques as a way of streamlining their work. Certainly automation can reduce manual operations, operation mistakes, and work hours. Originally I had also begun using it to make our work more effective. Although the streamlining work is valuable, there is more than streamlining to do in software product development – learning and collaboration. Learning is necessary to create the software right. Collaboration is the key factor to create the right software with team members and stakeholders. At the end of April 2013, I started supporting one new project as an Agile Coach. Through this project, I found the additional possibilities of automation and techniques to drive learning and to accelerate collaboration. They have been becoming established as a new model of agile culture in our organizations. I organized this mechanism and named it "Technology-Driven Development". In this paper, I show why and how to organize the "Technology-Driven Development" mechanism through lots of our challenges, thoughts and actions.

1.1 Conditions and challenges

At first, I got request from one new agile project to support them as an Agile Coach. The objective of the project was to develop a new smartphone application for Android and iPhone. There are tons of conditions and challenges in the team as follows.

(1) All team members had not have any experiences of Agile then (hereinafter called them the "Agile apprentices"). They adopted Agile because they needed to create the whole new product and they could not define all specifications up-front. But they also had the too much and unrealistic expectations to Agile. They imagined they could create appropriate product by just following the agile practices like Scrum, without any technical and cultural backbones, and without investigating their problems by their own.
(2) There had been tons of manual operations. They had tested and released their products manually then. They often had mistaken operations and needed to work overtime. There had been no slack to think of improving their work.
(3) The project team was basically consisted of 3 roles: business analyst, UI/UX designer (hereinafter called them the "designer"), and developer. They were able to work closely from the start of the project. But they had not had the common goals and objectives. The business analyst just said "implement all things what I said". The designers proposed new designs without considering implementability. There had been little collaboration at first.
(4) The most of team members had been young and immature. The average age of the team members was under 30. Especially, the average age of the Android developers was around 25. They had not had adequate skills and knowledge of their architecture, languages, and domain to solve problems by themselves

(5) 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. They had not been able to handle their project by themselves at first.
(6) The team was distributed to two locations. The distributed team without me always said "we are correct" and "you are wrong" without any material proof by tradition. There were a lot of miscommunications and distrusts between them.

At that time, we had needed to be stronger and to unite as one team as fast as we could.

1.2 The approach

To overcome these conditions and challenges, I decided to implement the automation and technical practices step by step through the following steps

(1) CI/CD
At first I focused on implementing CI/CD in terms of streamlining our work and starting collaboration with each other. I used the CI/CD to make the release operation easier then and to support test automation later. I also aimed to use the working software as a measure to create shared understanding among the all team members and stakeholders from the beginning of the project.

(2) TDD
After implementing CI/CD, I selected TDD for leveraging test automation and learning. At that time, team members and I had not had the enough skills to implement Android application. I thought that TDD would help us drive learning how to develop the Android application. But there were many troubles and barriers adapting TDD for Android.

(3) BDD
We were able to decrease work hours and operation mistakes by CI/CD and TDD. We got skills to develop required software gradually collaborated with members and stakeholders. By contrast, our project had started becoming chaotic. Because change requests from business analyst and designers were increased without considering deeply. These requests increased usecase-level bugs more. We had needed the discipline to restrain change requests, the domain knowledge to develop software more properly, and the measure to make usecase-level tests easier. So I adapted BDD to solve these challenges simultaneously in a hurry.

Though there were lots of successes and failures, our team could have become producing the proper software gradually through these approaches described above. Through this project, I had marshalled the ideas to drive streamlining, cultivation and collaboration in the software product development team via "technical base" like automation. It is really the additional possibilities of automation and techniques. Currently it is usual to use the idea "Technology-Driven Development" as a way of growing the agile culture in our teams and organizations.

2. CI/CD: AIMING RELEASE AUTOMATION AND THE START OF COLLABORATION

When I joined the project, it was very slow in whole. I investigated the project at first and found that there were so many manual release operations. There had been around 3 change requests per week then. Developers needed to do regression test manually and it took around 4.0 hours each time. Developers needed to install the latest application to each stakeholder's device and it took around 0.5 hours each time. A whopping 13.5 hours had been consumed every week for manual release at that time. Operation mistakes had been increasing and some developers had fallen sick. It was necessary to reduce manual release operations immediately to make the team sustainable.

On the other hand, team members and stakeholders had clearly argued in a circle. They did not have the clear vision and requirements from the beginning because the product was a whole new one. Additionally, they were not able to get the progress information in a timely manner. So I intended to use the working software as a common base for shared understanding among them. I thought the working software would help them clarify vision and requirements, and give them the progress intuitively.

I intended to achieve both reducing manual release operation and creating the baseline of collaboration with each other by CI/CD. The team had some batch scripts which were able to support release operation partially. Additionally, one designer had investigated TestFlight [13], a tool for delivering beta smartphone applications to the restricted users, for making her work easily. I combined them via Jenkins. The figure below shows the mechanism of CI/CD that I implemented.

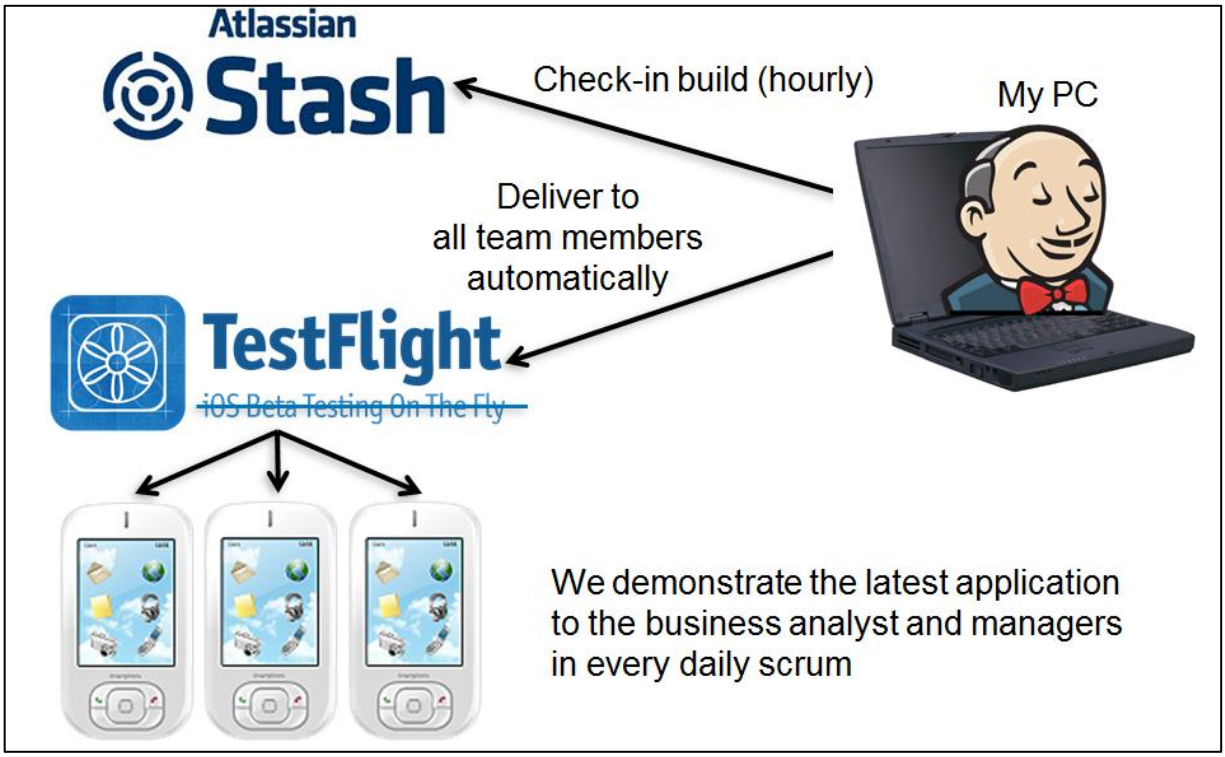

Fig. 1. The mechanism of CI/CD I implemented in our team.

I installed and ran Jenkins on my PC from the standpoint of quick implementation. If the developer commits programs to Stash (the same as GitHub in our company), Jenkins detects it, builds application, and releases it to all stakeholders' devices automatically via TestFlight. It means every members and stakeholders can use the latest application anytime and anywhere. We demonstrated the latest application at the daily scrum every morning. We were able to get fast feedback from the business analyst and designers. Stakeholders and members were able to know the progress via working software. We really used the working software as a measure for shared understanding.

After implementing CI/CD, it took only 15 minutes per week for releasing application. It was obviously effective for streamlining our work. I also gained the cooperation of stakeholders to proceed agile and automation more. I could go ahead with test automation. Additionally, one developer taught designers how to use Stash. After that, designers could also push the new design through our CI/CD mechanism. It was a good example of voluntary collaboration.

There were also challenges. We could know that we were delivering the working software every day. But it did not mean when we would complete what. It was insufficient just using the working software. We needed to visualize when and what by other way like Kanban board. Moreover, Apple bought the TestFlight. It means we could not deliver Android application through TestFlight more. It was really dire straits.

3. TDD: TO LEARN ANDROID DEVELOPMENT VIA UNIT TESTS

After implementing CI/CD, I selected TDD for leveraging test automation and learning. At that time, team members and I had not had the enough skills to implement Android application. I thought that TDD would help us drive learning how to develop the Android application. But there were many troubles and barriers adapting TDD for Android.

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.

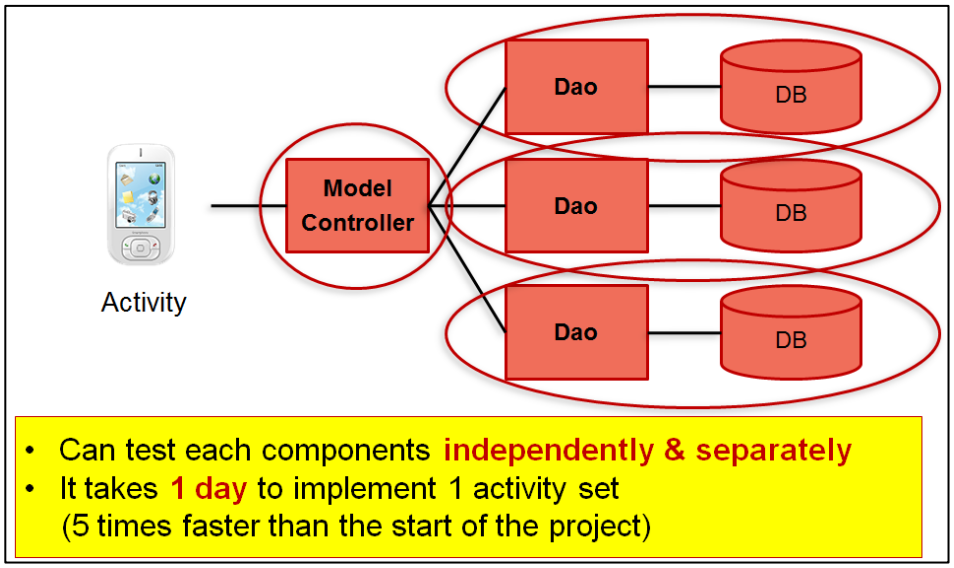

Fig. 2. The result of TDD for Android in my team.

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].

★Difficult to do pair programming with distributed office (due to cultural issue…)

4. BDD: DISCIPLINE IN A CHAOTIC PROJECT

We were able to decrease work hours and operation mistakes by CI/CD and TDD. We got skills to develop required software gradually collaborated with members and stakeholders. By contrast, our project had started becoming chaotic. Because change requests from business analyst and designers were increased without considering deeply. These requests increased usecase-level bugs more. We had needed the discipline to restrain change requests, the domain knowledge to develop software more properly, and the measure to make usecase-level tests easier. So I adapted BDD to solve these challenges simultaneously in a hurry.

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.

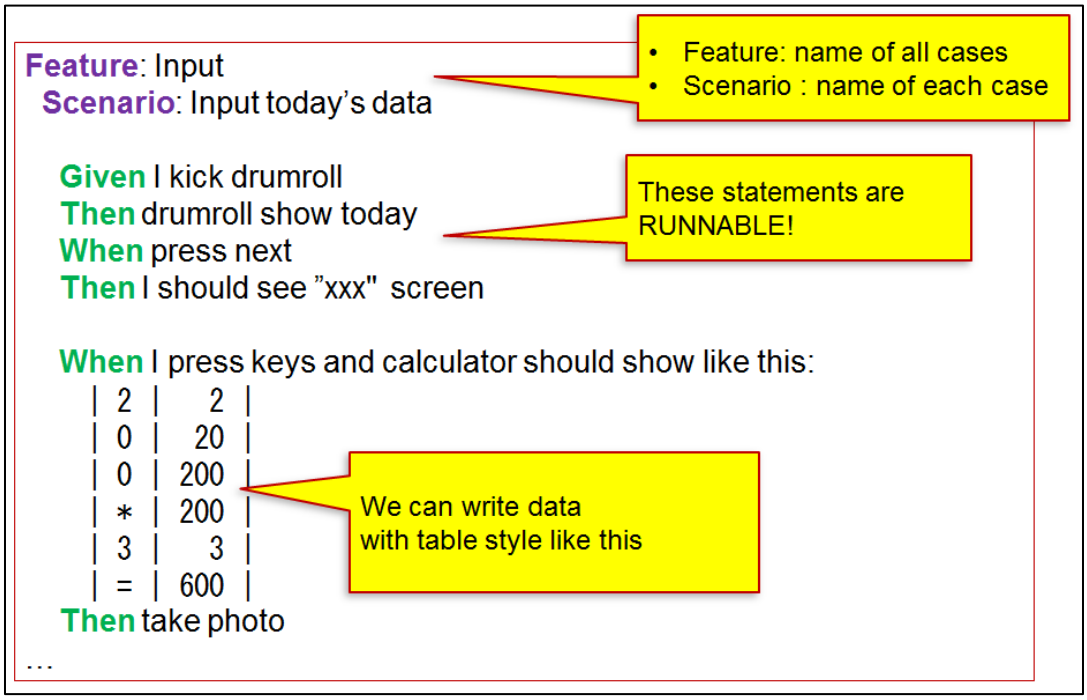

Fig. 3. Example of BDD test case.

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".

★Failed to create test cases by BA and designer

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.

★Act as coordinator, rather than mere workforce.

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/.