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

Hiroyuki Ito, Development Process Optimization Department, Rakuten, Japan

I achieved a new agile project named “Technology-Driven Development” as an Agile Coach. The word “Technology-Driven Development” has 3 meanings: the mechanism to make the work more effective, to develop cooperative relationships with the team members and stakeholders, 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 technical based for developing new smartphone applications, but also as a driver for developing engineering skills for the 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. Moreover, this mechanism grew the voluntary and supportive culture in our team.

In this paper, first I present the concrete mechanism of “Technology-Driven Development” I have introduced to our 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.

# 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” (a role in our company to educate the team members about the agile practices, techniques and mindsets through working with them). 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.

## 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. They adopted agile because they needed to create the whole new product and they could not define all specifications up-front. However, they also had 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 a lot 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 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. Our stakeholders are slight complicated. A business analyst in our team and managers belong to both our company and customer’s one. So they also behave as stakeholders. It led to lots of challenges later.
5. Most of the team members were young and immature. The average age of the team members was under 30. Particularly, the average age of the Android developers was around 25. They had not had adequate skills and knowledge of architecture, languages, and domain to solve problems by themselves.
6. The duration of the project was 6 months. Most of the team members did not have any experiences of a “big project” (over half of a year with over 10 members) like this. They had not been able to handle the project by themselves at that time.
7. The team was distributed to two locations. One location (where I were) built Android application. Another location built iPhone application. The distributed team always said that “we are correct” and “you are wrong” without any material proof by tradition. There were a lot of miscommunications and distrusts between them.

Therefore we had to work as a united body without delay.

## 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.

1. TDD

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

1. 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. Therefore 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.

# 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 we 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. The 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.

# TDD: to Learn Android Development via unit tests

After implementing CI/CD, I selected TDD next for leveraging test automation and learning. At that time, team members and I did not have enough knowledge and experience to implement Android application. We did not know the architecture, how to access database on device, how to implement UI, and so on. I thought that TDD would help us drive learning how to develop the Android application. On the other hand, we chose the three-tier architecture consisted of UI, Controller and Dao. It had taken around 5 days to implement one function on average because we were not able to test each component independently and separately then. I intended to use “Test Double” [11] with TDD for streamlining implementation and test.

But there were tons of troubles and barriers adapting TDD for Android. Soon I found that it was very difficult to do unit testing on Android platform. 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. Moreover, it is difficult to use the “Test Double” for component-level test. It takes too long to get useful feedback.

Eventually I adopted to implement the new test harness based on Robolectric [12] on behalf of Android JUnit. Robolectric enabled us to do unit testing without any emulator or devices. Robolectric also emulate lifecycle mechanism on Android. Additionally, Robolectric is easy to use Mockito [2], which enabled us to use the “Test Double”.



Fig. 2. The result of TDD for Android we implemented in our team.

Robolectric and Mockito enabled us to get fast feedback from unit testing. They also made it easier to do TDD and pair programming. I built sample programs with unit tests for each layer and taught members how to implement each component by using them. Moreover, we could build the low-dependent application architecture by using Mockito. In general, defining database is one of the biggest tasks in software product development. In our team, we could define all database tables and functions only 3 days by using TDD and pair programming. It took only 1 day to implement one function on average after implementing this TDD mechanism. They were really ”Technology-facing tests that support the team” [3].

There were also positive effects by TDD. Developers started pair programming with TDD voluntarily. They refactored our software continuously without any direction. They became responsible for the software spontaneously.

We faced with some difficult challenges. We could not implement TDD for iPhone. The iPhone team was distributed. There was a big distrust between the iPhone team and Android team by tradition. The iPhone team was very passive to adopt TDD we proposed. Additionally I could not procure a Mac PC to support them. It led to a number of bugs and troubles on iPhone application later.

# BDD: discipline in a chaotic project

TDD made our work more effective with CI/CD. We got skills to develop required software gradually collaborated with members and stakeholders. We were able to build and release working software faster and faster. By contrast, our project started becoming chaotic. There were 3 challenges in our team.

1. Change requests from business analyst and designers had been increasing rapidly without considering implementability and consistency. We had been clarifying specifications and functions step by step with working software. That was because our product was whole new and we could not define all specifications up-front. Although developers were able to build software faster, streamlining development led to a lot of requests by business analyst and designers. They misunderstood that it was possible to make developers implement anything what they thought right away. I needed to build a mechanism to restrict unrealistic or ad hoc requests quickly.
2. Usecase-level bugs had been increasing. Although increase of change requests by business analyst and designers was the main cause, developers could not decline ad-hoc requests. That was because developers did not have enough domain knowledge to argue with business analyst and designers. Though TDD helped developers learn architecture and system, it was insufficient to learn domain. It was desired to find a measure to make developers learn domain and usecase more.
3. We were not able to detect usecase-level bugs and degrade promptly. We implemented component-level tests by TDD, but they could not detect usecase-level bugs. That was why screen functions tend to be complicated due to many interactions inherent in smartphone. In smartphone application development, there were many domain logics in interactions among screens, gestures, external services and so on. At that time, it took around 3 days to detect those bugs and fix. It was too long for us. Therefore, it was necessary to build additional mechanism to detect usecase-level bugs and degrade promptly and automatically.

I adapted BDD to solve these challenges simultaneously in a hurry. I chose the Calabash-Android [7], the wrapper of the Cucumber [4] for Android, to implement BDD in our team. That was why the Calabash-Android can write usecase-level scenarios as test cases. I used test scenarios to elicit ideas and requirements from business analyst and designers. At first I wrote test scenarios while asking them and showed scenarios whether my understanding was correct or not. It was really a communication via test cases [1]. Executable test scenarios also enabled developers to clarify what they should develop and provide to users. We used test scenarios as a common language among business analyst, designers, and developers. Additionally, executable test scenarios could detect usecase-level bugs and degrade promptly and automatically. To wrap up, we used BDD as the “Business-facing tests that support the team” [3].

At that time, there were a lot of change requests, bugs and degrade in one function. Therefore, we focused on implementing the BDD test scenarios for the function. Specifically, we covered all bugs and degrade found in the function by the BDD scenarios. After that, we increased the test scenarios gradually throughout the product. Whereafter it took only 5 hours to detect and fix usecase-level bugs and degrade. The number of bugs and degrade were decreased by 60%. In addition, test scenarios made team members confident. They thought of scenarios for bugs and degrade as proofs of their efforts.

On the other hand, business analyst and designers didn’t write any BDD test scenarios. Initially, I indented to make business analyst and designers write test scenarios with the Calabash-Android to restrict their ad-hoc requests. However, I couldn’t make them out that writing test scenarios were their tasks. Most of them didn’t have any experience writing scenarios like the Calabash-Android. Developers and I often tried to support them to write scenarios, but it didn’t work. We could not restrict their unrealistic or ad hoc change requests only by BDD. Therefore, developers and I put a deadline for change requests on each user story. Deadline was set by developers from the standpoint of implementability and milestones. If business analyst or designers missed the deadline, developers and I declined requests. It worked well. The number of change request was decreased by 70%.



Fig. 3. Example of BDD test scenario by the Calabash-Android. We used it as a common language among team members and an executable specification to detect bugs and degrade promptly.

# Results

In the process of adapting CI/CD, TDD and BDD, team members and I gained a lot of useful insights I mentioned above. Our young team members and I improved our work enough to release the applications successfully. Moreover, I learned several interesting lessons.

The first lesson I learned is that automation and techniques nurture the team members. If there were bugs or degrade, Jenkins and tests detected them swiftly and notify all team members automatically. The TDD gave developers the knowledge of Android architecture. The BDD taught developers the domain. The BDD also taught business analyst and designers how to explain what they thought. Finally young Android developers were able to develop the required software faster than iPhone developers 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 is that continuous improvement leads other voluntary collaborations. 2 months after implementing TDD, developers started doing pair programming and refactoring without any direction. They exchanged their knowledge continuously by developing software. Additionally, one developer taught designers how to use Stash. After that, designers could also improve the product through our CI/CD mechanism. Furthermore, some members started pulling tasks voluntarily. They found and solved problems in advance without any instructions. The team members were becoming more self-active and confident.

The third lesson is that learning and collaboration make the work effective yet. Through pair programming and refactoring, developers improved architecture, performance and maintainability continuously. One developer found and 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. Moreover, Android developers helped iPhone developers by using Android tests (TDD and BDD) as a measure of exploratory testing for iPhone application. These devices were from slack by continuous improvement.

★以下が重要。キチンと整理すべし。

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 and techniques.

It is the essence of “Technology­Driven Development”.

# Problems, Possibilities and Future

I gained another useful lessons in the process of introducing “Technology­Driven Development”.

●Problem

Scope change was very difficult inherent in the organization.

★Act as coordinator, rather than mere workforce.

Managersとは仲良く＃

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 “Technology­Driven 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 “Fail­Fast Approach”, invoking small failures intentionally to drive learning. Team members are able to cover intentional small failures by “Technology­Driven Development” and get more knowledge to proceed improvements.

# Conclusions

I have described “Technology­Driven 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.

A number of people who does not have any experiences of agile 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. “Technology­Driven 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 “Technology­Driven Development” as the new learning model, regardless of what they produced.

I found that I was able to improve “Technology­Driven 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 “Fail­Fast Approach” will lead the team’s growth. I intend to strengthen “Technology­Driven 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.* Addison­Wesley 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/calabash­android.

[8] Humble, J., & Farley, D. 2010. *Continuous Delivery: Reliable software releases* *through build, test, and deployment automation*. Addison­Wesley 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*. Addison­Wesley  Professional.

[12]  Robolectric.org. http://robolectric.org/.

[13]  Testflightapp.com. http://testflightapp.com/.

[14]  Virtualbox.org. https://www.virtualbox.org/.