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

HIROYUKI ITO, DEVELOPMENT PROCESS OPTIMIZATION DEPARTMENT, RAKUTEN, JAPAN

In this experience report, I present a new mechanism called "Technology-Driven Development". "Technology-Driven Development" stands for three purposes: The first one is to make the work efficient. The second one is to develop cooperative relationships between the team members and stakeholders. The third one is 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).

The "Technology-Driven Development" mechanism has been chosen not only as a technical foundation for developing new smartphone application, but also as a driver for the team that consisted of young and immature members to learn new skills. This way the team members learned the skill for developing software and solving problems comparably or even better than the senior members. Moreover, this mechanism grew the voluntary and supportive culture in our team.

1. INTRODUCTION

Over the years, the main purpose of automation techniques has been considered as a way of making the work efficient. Certainly automation can reduce manual operations, operational errors, and work hours. Originally we used it to work more efficiently. Although making the work efficient is valuable, there is more to software product development, namely learning and collaboration. Learning is necessary to create the software right. Collaboration is the key factor to create the right software with the team members and stakeholders. At the end of April 2013, I started supporting one new project as an "Agile Coach", a dedicated role in our company to educate the team members on agile practices, techniques and the mindset through working jointly [9][10]. Through this project, I discovered additional possibilities of automation and development techniques that drive learning and accelerate collaboration. These techniques have been established as a new model for creating an agile culture in our company.

In this paper, first I explain our challenges and the approaches to solve them. Second, I present the concrete mechanism of "Technology-Driven Development". Third, I clarify the results in terms of making the work efficient, of learning, and of collaboration. Finally, the results, problems, possibilities and the future are discussed.

1.1 Conditions and challenges

At first, I got a request from a 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 iOS. In this project the team faced the following conditions and challenges:

(1) At that time, none of the team members had any experience with agile. Agile was adopted because we needed to create a whole new product and could not define all specifications up-front. However, there were unrealistic expectations toward agile: For example, it was expected that we could create the appropriate product just by following agile practices like Scrum, without any technical and cultural foundation, and without investigating in the problems on our own.
(2) There had been many manual operations: Testing and releasing the products was done manually before I joined. The products had many errors and the team members were often over-worked. There wasn't any slack time in order to think about improving the work.
(3) The project team basically consisted of three roles: business analyst, UI/UX designer (hereinafter called "designer"), and developer. The team members were able to work closely right from the start of the project. But they shared no common goals and objectives. The business analyst asked the developers to implement everything he requested. The designers proposed new designs without considering implementability.
(4) The role of the stakeholders was two-folded: The business analyst in our team and the managers belong to both our company and the customer's one. So they also acted as stakeholders. This led to a lot 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 about 25. They had

no adequate skills and knowledge about architecture, languages, or the domain in order to solve the problems by themselves.

(6) The duration of the project was six months. Most of the team members did not have any experience with a "big" project (over half of a year with more than ten members). They were not able to handle such a project by themselves at that time.

(7) The team was divided into two locations. One location (where I resided) built the Android application. Another location built the iOS application. The team members who belonged to the latter location tended to be proud and authoritative because this location is the origin of the organization. The iOS team members always thought of them as being correct and of the others as being wrong without any proof. There was a lot of miscommunication and mistrust between the two locations.

Yet, we had to work as one united team without delaying the delivery.

1.2 The approach

To overcome these conditions and challenges, I decided to implement the automation and technical practices step by step using the following approach:

(1) CI/CD

At first we focused on implementing CI/CD in terms of making the work efficient and starting to collaborate closer with each other. We used CI/CD in order to make releasing easier and to support test automation. We also aimed for the usage of working software as a measure for creating a shared understanding among all team members and stakeholders right from the beginning of the project.

(2) TDD

After implementing CI/CD, we used TDD for leveraging test automation and learning. At that time, we did not have enough knowledge for implementing the Android application. We hoped TDD would help us drive the learning for the development of the Android application. However, we faced many troubles and barriers while adapting TDD for Android.

(3) BDD

With CI/CD and TDD we were able to decrease the work hours and errors caused by manual operation. We developed the skills to implement the required software by improving the collaboration between the team members and the stakeholders. Yet on the other hand, the project started becoming chaotic, because the business analyst and the designers asked for more and more change requests without deep consideration. These requests increased the use-case bugs. We were in the need of a high discipline for restraining change requests, improving the domain knowledge in order to develop the software better, and for making the measurement of use-case tests easier. Therefore we adapted BDD quickly, in order to solve all of these challenges at once.

Though there were many successes and failures, our team was able to produce the proper software gradually by implementing the approaches described above. Through this project, I marshaled the ideas of making the work efficient, of learning, and of collaboration in the software product development team via technical practices like automation. Automation and the other techniques described provide indeed these additional possibilities. Currently "Technology-Driven Development" is the preferred way for growing the agile culture in our teams and both organizations.

2. CI/CD: AIMING FOR RELEASE AUTOMATION AND FOR COLLABORATION

When I joined the project, the performance was very low. When analyzing the project, I found that producing the release was done manually. There were about three change requests per week. Developers performed regression testing manually which took about four hours each time. They had to install the latest application on each stakeholder's device which took about half an hour each time. A total of 13.5 hours were consumed weekly for releasing manually at that time. Errors by manual operation increased and some developers became sick. It was necessary to reduce manual release operations immediately in order to sustain the team.

On the other hand, the team members and stakeholders had been going in circles. There was no clear vision and no requirements at the beginning because the product was a whole new one. Additionally, they were not able to get the information about the progress in a timely manner. Therefore I intended to use the working software for creating a shared understanding. I thought the working software would help to clarify the vision and the requirements, while providing status updates.

I intended to achieve both the reduction of manual release operations and the improvement of the collaboration by applying CI/CD. The team had some batch scripts that supported the release operations partially. Additionally, one designer had investigated in TestFlight [13], a tool for delivering beta smartphone applications for restricted users, which simplified the work. We combined both using Jenkins. The figure below shows the mechanism of CI/CD that we implemented.

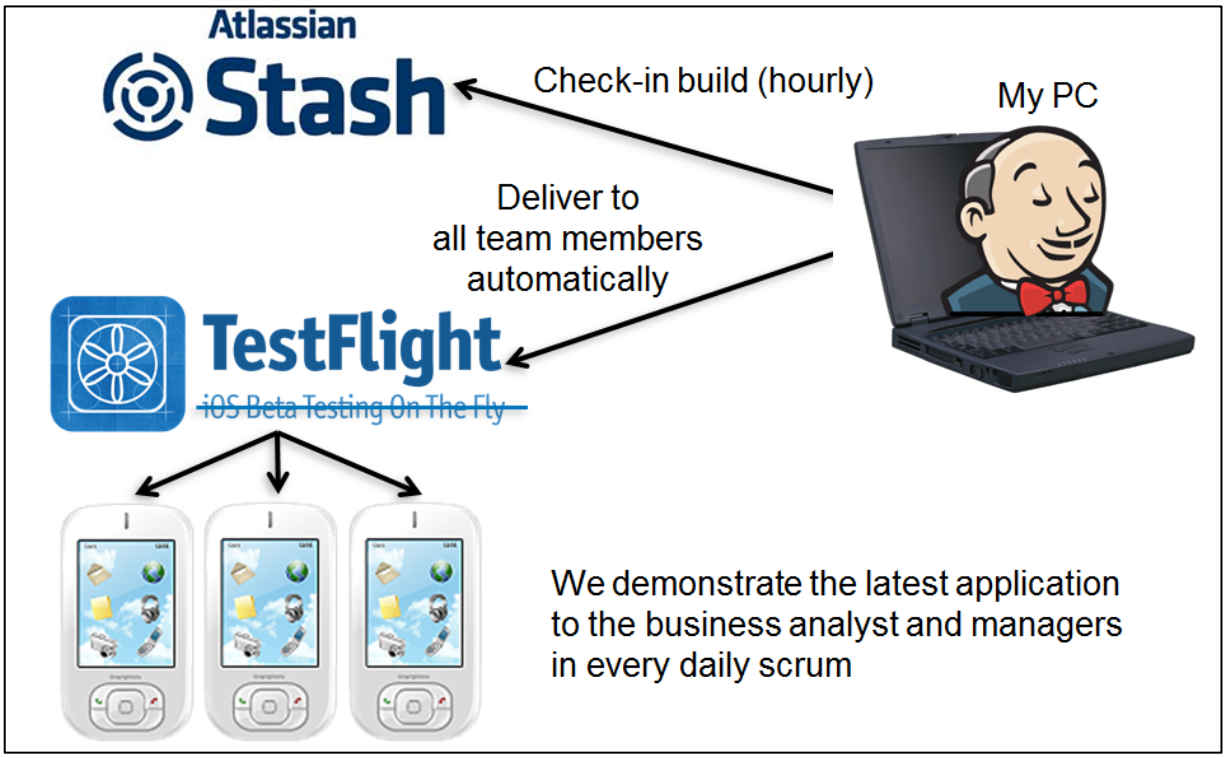

Fig. 1. The implementation of CI/CD: Making the work effective and using working software as a measure for creating a shared understanding.

As can be seen in Fig. 1, I installed and ran Jenkins on my PC for providing a quick implementation. If a developer commits a program to Stash (comparable to GitHub), Jenkins detects it, builds the application, and releases it to all stakeholders' devices automatically via TestFlight. This means all team members and stakeholders can use the latest application anytime, anywhere. We demonstrated the latest application at the daily scrum every morning. We were able to get fast feedback from the business analyst and the designers. The stakeholders and the team members were able to know about the progress via the working software. Thus, we used the working software as a measure for creating a shared understanding.

After implementing CI/CD, it took only fifteen minutes per week for releasing the application. Obviously we made the work efficient. Moreover, the stakeholders provided their support for agile and automation. We could start with implementing test automation. Additionally, one developer taught the designers how to use Stash. After that, the designers could also push the new design through the CI/CD mechanism. This was a good example for voluntary collaboration.

We also faced some challenges: We knew about our capability of delivering working software every day. But we did not know when we would complete what. It was insufficient just using the working software. We needed to visualize when we would deliver what. Using a Kanban board helped us a lot. Yet, Apple bought TestFlight. This meant we could not deliver the Android application with TestFlight anymore. These were really dire straits.

3. TDD: LEARNING ANDROID DEVELOPMENT VIA UNIT TESTS

After implementing CI/CD, we started using TDD for leveraging test automation and learning. We did not know the architecture, how to access the database on the device, or how to implement the UI. I thought that TDD would help us to learn how to develop the Android application. On the other hand, we chose a three-tier architecture consisting of UI, Controller, and DAO. It took about five days on average to implement one function because we were not able to test each component independently. I intended to use "Test Double" [11], a technique to replace the dependent component with the test-specific component, with TDD for streamlining both implementation and test.

But there we faced many troubles and barriers while adapting TDD for Android. Soon we discovered that it was very difficult to apply unit testing on the Android platform. The Android SDK provides its own test harness based on JUnit (hereinafter called the "Android JUnit"). Android JUnit requires an emulator or a device for unit testing. The Android JUnit starts a heavy lifecycle for each unit test case. Moreover, it is difficult to use the "Test Double" for tests on the component. It takes too long to get useful feedback.

Eventually we adopted a new test harness based on Robolectric [12] instead of Android JUnit. Robolectric enabled us to do unit testing without any emulator or device. Robolectric also emulates the lifecycle mechanism on Android. Additionally, Robolectric can easily be used with Mockito [2], which enabled us to use the "Test Double".

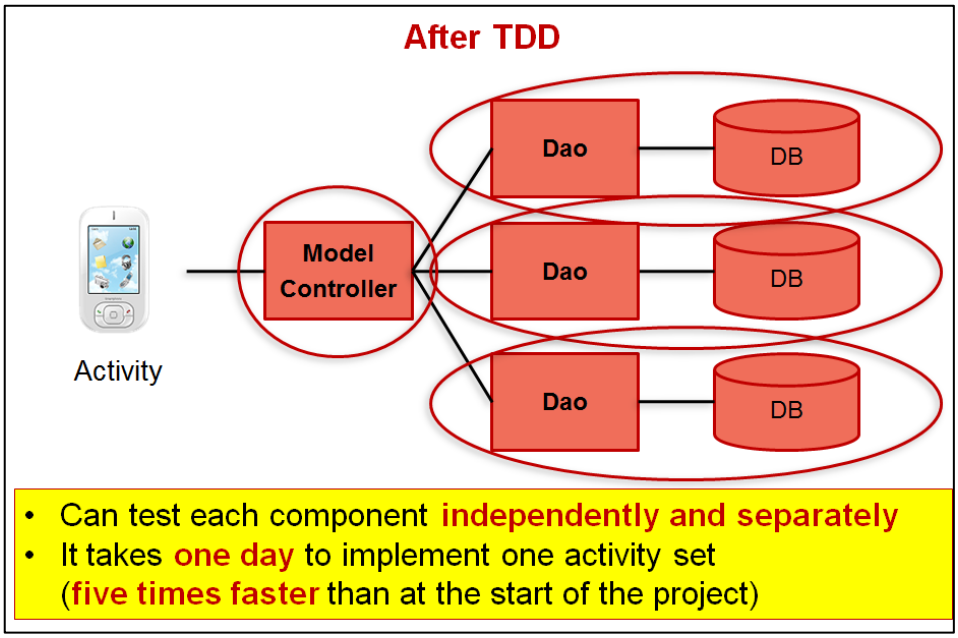

Fig. 2. The result of implementing TDD for Android: This mechanism enabled us to learn more about the system by using tests.

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 the team members how to implement each component by using these. As is shown in Fig. 2, we were able to build an application architecture that was less dependent by using Mockito. In general, defining the database is one of the biggest tasks in software product development. In our team, we could define all database tables and functions in only three days by using TDD and pair programming. It took only one day on average to implement one function after using TDD. It's five times faster than at the start of the project. The unit tests written by using TDD were indeed "Technology-facing tests that support the team" [3].

There were additional positive effects by applying TDD. Developers started pair programming with TDD voluntarily. They refactored the software continuously without any direction. They felt responsible for the software voluntarily.

Yet, we faced some difficult challenges. We could not implement TDD for the iOS application. The iOS team was located at a different site. Traditionally, there was a big distrust between the iOS and the Android team. The iOS team was hesitant to adopt TDD. Additionally I could not procure a Mac PC to support them. This led to a number of bugs and troubles in the iOS application later on.

4. BDD: IMPROVING THE DISCIPLINE

TDD made our work more effective with CI/CD. We were able to build and release the working software faster and faster. Still, the project became chaotic. The team faced the following three challenges:

(1) The business analyst and the designers had been asking for more and more change requests without considering implementability and consistency. We clarified the specifications and functions step by step by focusing on the working software. We could not define all the specifications up-front, because we developed a completely new product. Although the developers were able to build the software faster, making the development efficient led to many requests by the business analyst and the designers. They believed it was possible to ask the developers to implement anything they could think of right away. We needed to come up with a mechanism quickly for restricting unrealistic or ad hoc requests.
(2) Use-case bugs increased. Although the raising number of change requests by the business analyst and the designers was the main cause, developers could not decline ad-hoc requests. The reason was that the developers did not have enough domain knowledge in order to argue with the business analyst and the designers. Though TDD helped the developers to learn more about the architecture and the system, it was insufficient for becoming knowledgeable in the domain. It was desired to find a measure so the developers would learn more about the domain and the use-cases.
(3) We were not able to detect use-case bugs and regressions promptly. We implemented component tests by applying TDD, but could not detect the use-case bugs. Due to many interactions inherent to the smartphone, the screen functions tended to be complicated. The smartphone application has many interactions among screens, gestures, external services and so on. At that time, it took about three days to detect and fix the bugs. This was too much time. Therefore, it was necessary to build an additional mechanism in order to detect bugs and regressions on the use-case.

We adapted BDD to solve these challenges quickly. We chose Calabash-Android [7], the wrapper of Cucumber [4] for Android to implement BDD. As can be seen in Fig. 3, the Calabash-Android allows to writing use-case scenarios as test cases. We used test scenarios to elicit the ideas and requirements from the business analyst and the designers. At first, I wrote the test scenarios while asking the business analyst and the designers, and showing these scenarios for finding out if my understanding was correct. Thus, we communicated via test cases [1]. These executable test scenarios also enabled the developers to clarify what they should develop and provide to the users. Business analyst, designers, and developers used test scenarios as a common language. Additionally, executable test scenarios could detect use-case bugs and regressions promptly and automatically. Finally, we used BDD as “Business-facing tests that support the team” [3].

At that time, there were a lot of change requests, bugs and regressions in one function. Therefore, we implemented the BDD test scenarios for that function. Specifically, we covered all bugs and regressions found in the function with the BDD scenarios. After that, we implemented more and more test scenarios for the product. Afterwards it took only five hours to detect and fix use-case bugs and regressions. The number of bugs and regressions were decreased by 60%. In addition, the test scenarios made the team members more self-confident. They considered the test scenarios for bugs and regressions as proofs of their efforts.

On the other hand, the business analyst and the designers didn’t write any BDD test scenarios. Initially, I indented to make the business analyst and the designers write the test scenarios with the Calabash-Android in order to restrict their ad-hoc requests. Yet, most of them didn’t have any experience with writing test scenarios. We tried to support them to write the test scenarios, but it didn’t work out. We could not restrict their unrealistic or ad hoc change requests by BDD only. Therefore, we decided on a deadline for change requests for each user story. The deadline was based on the implementability and the milestones. If the business analyst or the designers missed the deadline, we declined the request. This worked out well. The number of change requests was decreased by 70%.

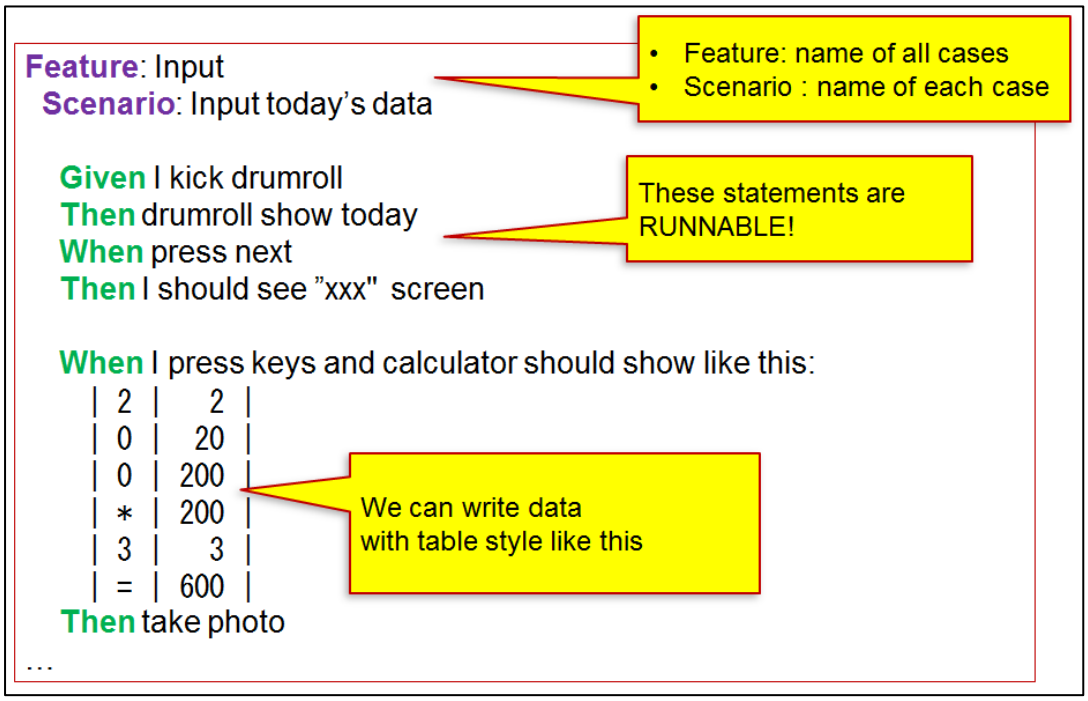

Fig. 3. Example of BDD test scenario with the Calabash-Android: It was used as a common language among the team members and as an executable specification to detect bugs and regressions promptly.

5. RESULTS

In the process of adapting CI/CD, TDD and BDD, we gained a lot of useful insights. We improved our work so that we could release the application successfully. Moreover, I learned several interesting lessons:

The first lesson is that automation and development techniques like CI/CD, TDD and BDD nurture the team members. If there were bugs or regressions, Jenkins and the according tests detected them swiftly and notified all team members automatically. TDD gave developers the knowledge of the Android architecture. BDD taught developers about the domain. BDD also taught the business analyst and the designers to explain what they thought. Finally, the young Android developers were able to develop the required software within five months, which was two months faster than the iOS developers. Their application performed better and with less bugs than the iOS application developed by seniors in another location. (It took about six months to develop the iOS application and it had twice as much bugs compared to the Android one.)

The second lesson is that continuous improvement with automation techniques leads to other voluntary collaborations. Two months after implementing TDD, developers started doing pair programming and refactoring without any direction. They exchanged their knowledge continuously while developing software. Additionally, one developer taught the designers how to use Stash. After that, the designers could also improve the product through the CI/CD mechanism. Furthermore, some team members started pulling tasks voluntarily. They found and solved problems in advance without any instructions. The team members were becoming more self-organized and self-confident.

The third lesson is that learning and collaboration make the work more efficient. Through pair programming and refactoring, developers improved the 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 BDD and the development about ten times faster than before. Moreover, the Android developers helped the iOS developers by using Android tests (TDD and BDD) as a measure for exploratory testing of the iOS application. The more the work improved, the more the slack time increased to improve further.

I have introduced a series of technical improvements which were valuable for most team members and stakeholders. Automation and development techniques like CI/CD, TDD and BDD made our work more effective. Moreover, these improvements also enabled learning and collaboration. This mechanism can grow a voluntary and continuously improving culture. This is the essence of "Technology-Driven Development".

6. PROBLEMS

Although I gained a lot of useful lessons in the process of adapting “Technology-Driven Development”, I also faced some big challenges:

The first problem is the organizational and/or cultural traditions which could not be solved by technical excellence and working software only. For example, changing the scope was very difficult inherent to the organizations of the team members. In the middle of the project, I found out that the business analyst had not accepted any scope change. On inquiry, I realized that all team members except me belonged to both our company and the customer’s one. In the customer’s company, they have to achieve everything that has been planned at the start of the project. As I mentioned earlier, the team members and stakeholders adopted agile because they could not define all specifications up-front in our company. However, this was not true for the customer’s company. It was exactly the opposite. Thus, it was impossible to solve this challenge only by collaboration with the team members. Also the managers could not support us. Finally, I told our company’s executive the whole story and asked to solve it. After that, the team was able to change the scope as necessary.

The second problem is that some team members and stakeholders opposed to a series of improvements. The iOS team members were not cooperative. There was traditionally a lot of miscommunication and distrust between the iOS and the Android team. There was a project manager, who did not manage the project. He was responsible for multiple projects and one of those was in a crisis. Therefore, I decided to use the results of the technical improvements in order to overcome the absence of the project manager and to “bow” the iOS developers. Though, this didn’t work well. The iOS developers resisted emotionally. The project manager sent apocryphal progress reports to the stakeholders regardless of the real achievements. I should have communicated more sincerely and honestly with the iOS developers and the project manager to get their support, rather than forcing the results.

The third problem is that the team members and stakeholders sometimes thought of me as an additional workforce for the project. As I mentioned earlier, I led a series of technical improvements. I often used the results by the technical improvements to get the support from the team members and stakeholders. Through the process, the team members and stakeholders acknowledged my skills to develop software and to lead the project. They requested that I increase the ratio of developing software due to a severe lack in scope implementation and delivery. I accepted their request, however, this brought the project to a standstill. The more I developed, the more they requested. At times I lost my composure in order to see the whole project. I couldn’t have given the team appropriate advices. At that time, as a leader of the improvement strategy (or as an Agile Coach), I should have clarified the cause for the increasing requirements. If we use “Technology-Driven Development”, the Agile Coach would be better off acting as a coordinator, rather than a mere workforce.

7. POSSIBILITIES AND FUTURE

On another front, I discovered the possibilities and the future of “Technology-Driven Development” through the challenges I mentioned above:

Firstly, numerical measurement makes “Technology-Driven Development” more effective. Numerical measurement helps to find the problems the team wants to solve. Moreover, numerical measurement supports decision-making by the team. It also helps managers and stakeholders.

Secondly, we would be better off using “Technology-Driven Development” as a measure for total optimization. To optimize totally, it is necessary to see the whole project. Additionally, it is very useful to get the support from the managers for understanding the whole picture. Numerical measurement is a key factor for collaborating with managers.

Finally, it is judicious to use “Technology-Driven Development” as a measure to achieve the results by the team, not by the leader. The leader often feels the temptation to achieve personally. However, this “false” achievement-oriented attitude disables the team to solve the problems on its own. The more the team achieves, the more the growth sustains.

The automation and development techniques like CI/CD, TDD and BDD are powerful but difficult because they can achieve short-term results easily. However, short-term effects are not sustainable. To make effects long-lasting, it is necessary to grow the team continuously. That is to say, we ought to grow an agile culture. “Technology-Driven Development” has the possibility to grow an agile culture. On the other hand, it is judicious to improve the mechanism continuously by itself. This is the key factor to make this mechanism sustainable.

8. CONCLUSIONS

I have described “Technology-Driven Development”, a mechanism to drive making the work efficient, to learning, and to collaboration in a software product development team using automation and development techniques like CI/CD, TDD and BDD. We can drive each element spirally. This mechanism can also grow a voluntary and continuously improving culture. A number of people who do not have any experiences with agile tend to introduce agile processes and the mindset first without any technical foundation and therefore fail. The technical foundations like CI/CD and test automation enable effective learning and elicit voluntary improvements for the team members. “Technology-Driven Development” will be a good foundation for supporting and enhancing agile processes and mindset.

On the other hand, there is room for improving the mechanism. There are challenges which cannot be solved by technical excellence and working software only. We should not ignore the organizational and/or cultural traditions. We need to get support from all team members and stakeholders, rather than forcing the achievements. The leader of the improvement strategy should act as a coordinator, not a mere workforce. Numerical measurement will support improving the “Technology-Driven Development” mechanism.

Finally, our young team members released the Android and iOS application successfully. After that, some other teams started adopting the “Technology-Driven Development” mechanism.

I firmly believe that “Technology-Driven Development” helps to strengthen our team members, both organizations and company thoroughly and continuously. Furthermore, I want to improve “Technology-Driven Development” by itself iteratively and incrementally to make the mechanism more sustainable.

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/.