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 first one is to streamline the work. The second one is to develop cooperative relationships with 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).

I have used the "Technology-Driven Development" not only as a technical base for developing new smartphone applications, but also as a driver for the team consisted of young and immature members to learn skills. This mechanism brought young members the skill for developing software and solving problems as well or better than seniors. Moreover, this mechanism grew the voluntary and supportive culture in our team.

In this paper, firstly I explain our challenges and the approaches to solve them. Secondly, I present the concrete mechanism of the "Technology-Driven Development" I introduced. Thirdly, I clarify the results of streamlining, learning and collaboration by the mechanism. Finally, I consider the "Technology-Driven Development" from the aspect of problems, possibilities and future.

1. INTRODUCTION

Over the few years, the main purpose of automation techniques has been considered as a way of streamlining the 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.

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. 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 before I joined. They often had mistaken and overworked. Therefore, 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.

伊藤 宏幸 14/5/6 21:35
コメント [1]: Revise the Agile2014 page later.

Hiroyuki Ito (The Hiro) 14/4/29 17:47
コメント [2]: #9
https://github.com/hageyahhoo/agile2014/issues/9
Moved the word "business analyst' later to clarify it.

Hiroyuki Ito (The Hiro) 14/4/29 17:48
コメント [3]: #3
https://github.com/hageyahhoo/agile2014/issues/3
Added it to explain more about the growth of agile culture.

Hiroyuki Ito (The Hiro) 14/4/29 17:48
コメント [4]: #12
https://github.com/hageyahhoo/agile2014/pull/12
Already adapted the ACM structure.

伊藤 宏幸 14/5/6 21:35
コメント [5]: Clarify the agenda of the paper.

Hiroyuki Ito (The Hiro) 14/4/29 17:55
コメント [6]: #8
https://github.com/hageyahhoo/agile2014/issues/8
Clarified the role of "Agile Coach".

伊藤 宏幸 14/5/6 21:38
コメント [7]: #18
https://github.com/hageyahhoo/agile2014/issues/18
序論に入れるには大きすぎる気がする。むしろ本論の気がするので、移動を考える。

Hiroyuki Ito (The Hiro) 14/4/29 17:50
コメント [8]: #8
https://github.com/hageyahhoo/agile2014/issues/8
Removed the word "agile apprentices".

(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 divided into two locations. One location (where I were) built Android application. Another location built iPhone application. The members who belong to the latter location tended to be proud and high-pressure because the latter one is the origin of the organization. The iPhone team members always said that "we are correct" and "you are wrong" without any material proof. There were a lot of miscommunications and distrusts between them.

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

Hiroyuki Ito (The Hiro) 14/4/29 17:56
コメント **[9]:** #9
https://github.com/hageyahhoo/agile2014/issues/9
Clarified that BA is both member and stakeholder.

1.2 The approach

Hiroyuki Ito (The Hiro) 14/4/29 17:56
コメント **[10]:** #7
https://github.com/hageyahhoo/agile2014/issues/7
Clarified the 3 meaning of Technology-Driven Development.

伊藤 宏幸 14/5/6 21:38
コメント **[11]:** #17
https://github.com/hageyahhoo/agile2014/issues/17
おそらくここの"approach" は、設定した課題を紐解くアプローチのことのはず。
現場での実証を通じて〜となるだろう。

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 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.

(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. 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 marshaled the ideas to drive streamlining, learning 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. Therefore 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 that 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.

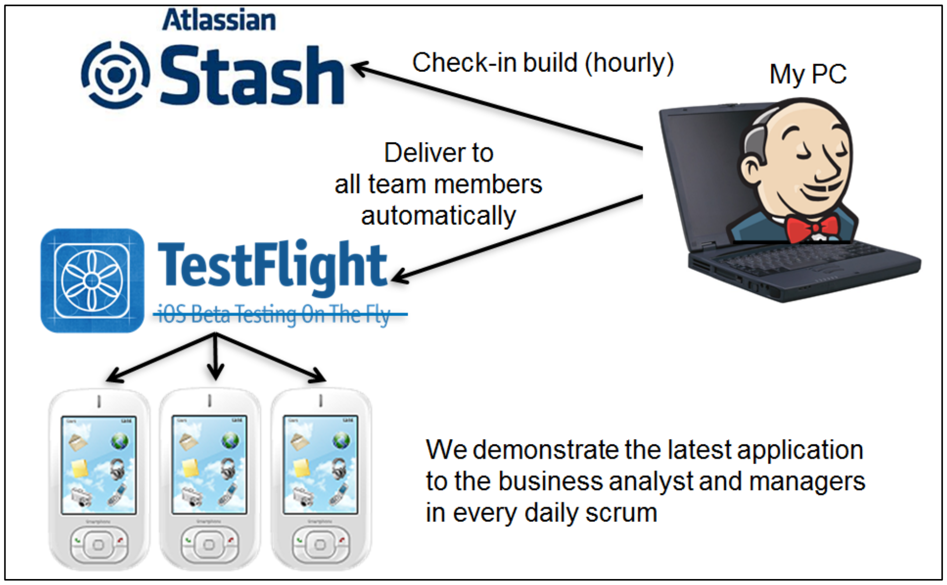


Fig. 1. The mechanism of CI/CD we implemented. We streamlined the work and used the working software as a measure for shared understanding by this mechanism.

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.

Hiroyuki Ito (The Hiro) 14/4/29 17:56
コメント **[12]:** #11
https://github.com/hageyahhoo/agile2014/issues/11
Adding Rebecca-san's advice like this.

3. 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".

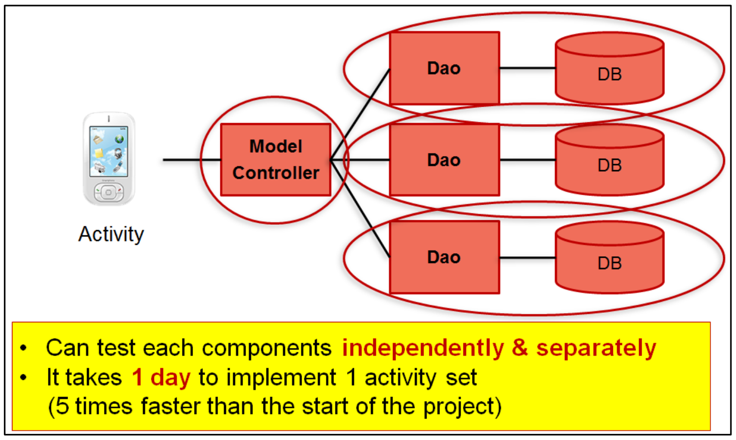

Activity
Model Controller
Dao
DB
Dao
DB
Dao
DB
• Can test each components independently & separately
• It takes 1 day to implement 1 activity set
(5 times faster than the start of the project)

Fig. 2. The result of TDD for Android we implemented. This mechanism also enabled us to learn 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 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.

4. BDD: DISCIPLINE IN A CHAOTIC PROJECT

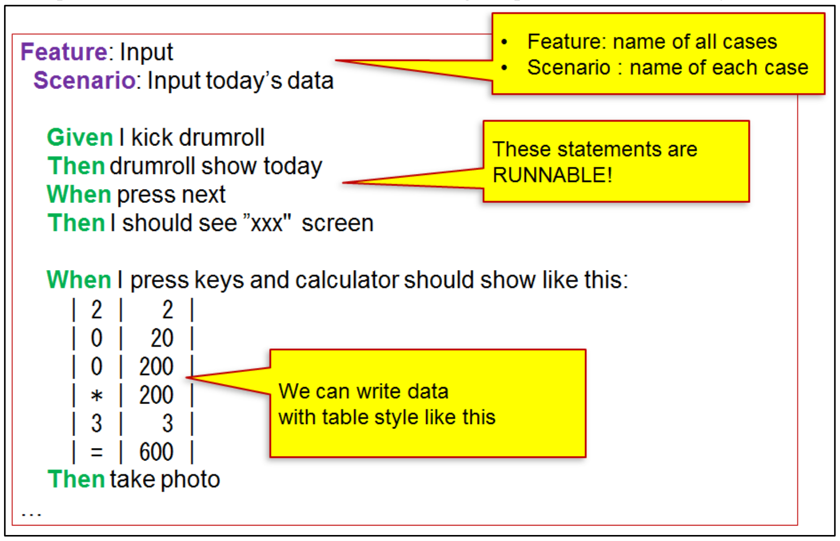
TDD made our work more effective with CI/CD. We got skills to develop required software gradually collaborated with members and stakeholders. We were able to build and release working software faster and faster. By contrast, our project started becoming chaotic. There were 3 challenges in our team.

(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. Developers set deadline 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%.



```
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

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.

5. 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 by automation techniques 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 with automation techniques.

I have introduced a series of technical improvements valuable to most of members and stakeholders. Automation and techniques made our work more effective. Moreover, these improvements also have learning and collaboration mechanisms that are underpinned by automation and techniques. I can tell that this mechanism can grow a voluntary and continuously improving culture. It is the essence of the “Technology-Driven Development”.

6. PROBLEMS

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

The first problem is that I faced with the organizational and/or cultural traditions which could not be solved by only technical excellence and working software. For example, changing scope was very difficult inherent in the organizations of our members. In the middle of the project, I had found that the business analyst had not accepted any scope change. On inquiry, I realized that the team members except me belonged to both our company and customer’s one. In customer’s company, they must have to achieve all they 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 it was not true “in the customer’s company”. It was exactly a headache. It was impossible to solve this challenge only by collaboration with team members. Managers could not support us, too. Finally, I told our company’s executive the whole story and ordered to arrange it. After that, the team was able to change scope as necessary.

The second problem is that some members and stakeholders went against a series of improvement. The iPhone team members were not cooperative. There were a lot of miscommunications and distrusts between iPhone them and Android team by tradition. There was a project manager, however, he did not manage the project. He had held plural projects and another project had been in crisis then. Therefore, I had decided to bring results by technical improvements more to overcome the absence of the project manager and to “bow” iPhone developer. Though it didn’t work well. The iPhone developers had more resisted emotionally. The project manager had sent stakeholders the apocryphal progress reports regardless of the real achievements. I should have communicated more sincerely and honestly with them to get support, rather than forcing the results.

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

伊藤 宏幸 14/5/4 21:10
コメント **[13]:** 唐突すぎる印象あり。

7. POSSIBILITIES AND FUTURE

On another front, I found the possibility and future of the “Technology-Driven Development” through the challenges I mentioned above.

Firstly, numerical measurement makes the “Technology-Driven Development” more effective. Numerical measurement helps us to find the problems the team wants to solve. Moreover, numerical measurement is able to support the decision-making by the team. It also means it can help managers and stakeholders.

Secondly, we would be better off using the “Technology-Driven Development” as a measure of total optimization. To optimize totally, it is necessary to see the whole project. The leader of the improvement (or the Agile Coach) ought to behave as a coordinator, rather than a mere workforce to do that. Additionally, it is very useful to get support from managers for keeping the whole picture. Numerical measurement is a key factor to collaborate with managers.

Finally, it is judicious to use the "Technology-Driven Development" as a measure to achieve results by the team, not by the leader. The leader often feels the temptation to achieve personally. However, this "false" achievement-oriented attitude takes the team away the ability to solve problems by themselves. The more the team achieves, the more the growth is sustainable.

The automation and techniques are powerful but difficult because they can achieve short-term results easily. However, short-term effects are not sustainable. To make effects long-term, it is necessary to grow the team continuously. That is to say, we ought to grow an agile culture. The "Technology-Driven Development" has the possibility to grow an agile culture. On the other hands, it is judicious to improve the mechanism continuously by itself. It is the key factor to make the mechanism sustainable.

8. CONCLUSIONS

I have described the "Technology-Driven Development", a mechanism to drive streamlining, learning and collaboration in the software product development team using automation and 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 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. "Technology-Driven Development" will be a good backbone to support and enhance agile processes and mindsets.

On the other hand, there is room to improve the mechanism. There are challenges which cannot be solved by only technical excellence and working software. We should not ignore the organizational and/or cultural traditions. We need to get support from all as many as possible of members and stakeholders, rather than forcing the achievements. Leader of the improvement should act as a coordinator, not a mere workforce. Numerical measurement will support it.

Finally our young team members released the Android and iPhone applications successfully. After that, some other teams started adopting the "Technology-Driven Development" mechanism. Currently it is usual to use this idea as a way of growing the agile culture in our teams and organizations.

I firmly believe that I intend to strengthen the "Technology-Driven Development" thoroughly and continuously to strengthen our team members, organizations and company. Furthermore, I would improve "Technology-Driven Development" by itself iteratively and incrementally to make the mechanism sustainable.

Hiroyuki Ito (The Hiro) 14/5/2 21:08
コメント **[14]:** #8
https://github.com/hageyahhoo/agile2014/issues/8
Removed the word "agile apprentices".

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/.

伊藤 宏幸 14/5/6 21:38
コメント **[15]:** #19
https://github.com/hageyahhoo/agile2014/issues/19