Growing and Expanding Transformational Leaders Team with Experiments

Hiroyuki Ito, SET taskforce, LINE, Japan

Since I joined LINE Corporation as the first member of SET (Software Engineer in Test) in 2017, we have been solving a variety of software and organizational problems. Through these achievements, we have been adjusting our responsibilities from software quality to software delivery, profitability, and organizational processes.

This report is about why and how we are becoming a team of [Transformational Leaders](https://en.wikipedia.org/wiki/Transformational_leadership) that is responsible for software delivery performance and organizational culture (\*) based on experiments.

# Introduction

LINE is a free chatting and telecommunication service for smartphones that has released since 2011. Our company name is derived from this service.

After the first release, LINE Corporation has been increasing users and messages transferred rapidly and globally. Especially, high sound quality with free, and the “sticker” feature that we can send a variety of rich emoticons as a message attracted a lot of users.

For adapting to the rapid growth of LINE, we have been improving LINE's architectures and code base iteratively. We chose [Microservice Architecture](https://martinfowler.com/articles/microservices.html) to earn scaling out, independent development, and fast delivery capabilities.

However, outages of LINE have also been increasing. Features, especially fintech ones like payments and banking, have been increasing dramatically. Troubles at Integration Points (\*) among each Microservices have also been increasing. They mean increases of negative monetary impacts to LINE users.

For reducing outages and improving product development processes, LINE Corporation decided to open new positions for a Scrum Master, a DevOps engineer, and an SET.

I joined LINE Corporation as the first member of SET in 2017.

# CLARIFY DEMANDS AND RESPONSIBILITIES TO START

## CHALLENGES

When I joined LINE Corporation, there were lots of problems which relate to not only software quality but a variety of software and organizational problems.

The biggest challenge was the confusions and disagreements about SET among stakeholders. There were no clear objective, missions, and responsibilities of SET. Additionally, there were no shared understanding about SET. Therefore, I needed to clarify them at first.

Other big challenge was that I was a newbie of LINE Corporation and I didn't have enough knowledge of our services, architectures, technologies used, and so on.

Moreover, there were few leaders to solve problems which affect more than one team and/or service. In other words, few leaders could act beyond silos. We have been widely adopting to Microservice Architecture. It was critical to overcome this problem for solving outages quickly and properly with less impacts to users.

## ACTIONS

For the smooth start of SET activities, I did the following actions.

### GATHER INFORMATION WITH PRODUCT DISCOVERY

To clarify objective, missions, and responsibilities of SET, I utilized the idea of [Product Discovery](http://productdiscoverycanvas.com/tag/david-hussman/) taught by [David Hussman](https://twitter.com/davidhussman) for gathering necessary information.

At first, I analyzed our services and products. I utilized static code analysis tool named [SonarQube](https://www.sonarqube.org/) to know the code coverage and technical debts for each service. I also added simple unit and integration test scripts to know behaviors of the products. Test scripts are good for understanding software under test (\*).

Next, I focused on analyzing "outage reports". "Outage reports" mean both postmortem meetings and published reports. They are a treasure-trove of information we need to solve. I was able to know causes of outages, impact on sales and profits, and problematic products through those reports. I understood that public APIs provided for external users was the most problematic product. Additionally, I found that reducing MTTR (Mean Time to Repair) would be an impactful solution as the first step.

Moreover, I talked stakeholders like developers, QA persons, product managers, senior managers, and executives to hear their concerns and troubles directly and beyond silos. Stakeholders' worries are also a treasure-trove of information to improve. Through those conversations, I understood that they had lots of non-verbalized problems. I also learned that verbalizing problems through direct and honest conversations

is critical for discovering real needs, shared understanding, and collaborations beyond silos.

### TODO

### TODO

# 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”.

Figure 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 Figure 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. That’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 testing challenges with testing on the other platform. 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. As I mentioned earlier, the iOS team members always thought of themselves as being correct and of the Android team members as being wrong without any proof. The iOS team was hesitant to adopt TDD because we the Android team proposed them to do it. Moreover, the iOS team members did not have enough technical skills to adapt TDD. We tried to teach them TDD, however, we could not procure Mac PCs to provide the test harness of the iOS and to support the iOS team. Therefore, it was too difficult for us to teach the iOS team TDD remotely. These challenges led to a number of bugs and troubles in the iOS application later on.

# 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 use-cases.

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 Figure 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%.



Figure 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.**

# 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 associated 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 the other location. (It took about six months to develop the iOS application and it had twice as many 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 efficiency improved, the more the slack time became available 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 practice can grow a voluntary and continuously improving culture. This is the essence of “Technology-Driven Development”.

# 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 the 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 were 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 the 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 the stakeholders. Through the process, the team members and the stakeholders acknowledged my skills to develop software and to lead the project. They requested that I work more on developing the software than leading the project due to a severe lack in scope implementation and delivery, and a shortage of skillful engineers. I thought their request was valid at that time and I accepted it. However, this brought the project to a standstill. There was no person who can lead the project. Additionally, feature creep happened. The more I developed, the more they requested. At times I became so busy that I lost focus of the bigger picture in order to see the whole project. I couldn’t also have given the team appropriate advice. 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 just a workforce.

# 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 prevents the team from solving the problems on their own. The more the team achieves, the more the growth they sustain.

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 practice continuously by itself. This is the key factor to make this practice sustainable.

# Conclusions

I have described “Technology-Driven Development”, a practice 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 practice 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 practice. 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 just a workforce. Numerical measurement will support improving the “Technology-Driven Development” practice.

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

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 practice more sustainable.

# AcknowledgementS

Many people helped me write this paper. Many thanks to Kotaro Ogino, Ryotaro Ikeda, and Takeshi Arai for reviewing most of my writing. Joel Tucci, Yasunobu Kawaguchi, and Xuebin Ma also supported the reviewing process. Dai Fujihara and Takao Oyobe always give me a lot of advice as an Agile Coach. I would like to say thanks to Masaya Mori, Yoshihiko Saeki, Kenji Kosaka and Rakuten. They allowed me to proceed the approaches I mentioned on this paper.

At last, special thanks to Jutta Eckstein, my shepherd, and Rebecca Wirfs-Brock. I could not write this paper without their insightful and positive suggestions.

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/.