Effective Distributed Pair Programming

Mark Rajpal, M.Sc.
Agile Global Results
Canada
mark.rajpal@agileglobalresults.com

ABSTRACT

Pair Programming[1] has quickly become a widespread technique for teams adopting Agile. The interest has gained even more popularity for those implementing Extreme Programming (XP) which is a well-known Agile methodology. There is a general agreement that pair programming works well with a pair of developers working side by side. However, that is not always possible. Distributed teams and distributed team members are becoming the norm. With that said, is distributed pair programming possible? And if so, can it be done effectively? This paper examines a project where distributed pair programming was extremely effective. Contrary to agile dogma, some individuals may perform better in a distributed manner over collocated.

CCS CONCEPTS

• **Computing methodologies → Software development**;

KEYWORDS

Agile, Scrum, pair programming, extreme programming

1 INTRODUCTION

Distributed pair programming is a technique where two individuals are performing pair programming with the exception that they are not physically located next to one another. For example, pairs could be separated by temporal or geographic barriers but that does not necessarily have to be the case. There could be a situation where distributed pair programming is accomplished by two individuals working in the same office tower but located on different floors.

While distributed pair programming is not a new concept, there seems to be very little research in this area. Pair programming on the other hand seems to be well represented in the form of experience reports, case studies, literature review, conference presentations, blog posts, books, etc.

The next section provides an overview of the project. Section 3 provides a synopsis of the project team. Section 4 provides a summary of the approach. Section 5 provides a summary of the various tools that were used. The Results are summarized in Section 6 and the Analysis is provided in Section 7. Lessons Learned is covered in Section 8 and Conclusions are presented in Section 9.

2 PROJECT XYZ

In 2016, a software development project (Project XYZ) was undertaken for a client in the United States. The aim of the project was to rescue a web based application that had undergone much neglect from a previous vendor. The Java application was riddled with defects and the business had endured many changes over a long period of time where the application fell behind. The application was completely new to the team and the project was approximately 10 months in duration. It was decided that 2 week sprints would provide the client with frequent progress and the ability to provide constant feedback which is something they had not been used to.

A single Scrum team was assigned to the project and was distributed throughout Canada (i.e. Edmonton/Calgary/Vancouver/Toronto) where some team members were collocated. Pair programming and distributed pair programming was a daily ritual for the developers.

3 TEAM COMPOSITION

An attempt was made for the entire team to be collocated in Calgary. However, after many interviews and LinkedIn searches it was evident that this would not be possible due to a lack of available local talent. Once the team was fully staffed the result was; 2 developers and the Scrum Master in Calgary, 1 developer in Vancouver, 1 developer in Toronto, and 1 tester/requirements engineer in Edmonton.

Agile and Scrum were fairly new to most of the team but everyone agreed that would be a good starting point. Additionally, some XP practices were also incorporated including; continuous integration, small releases, simple design, and the whole team approach. Prior to this project, I had experience in other Agile initiatives both as a Scrum Master and Team Member (but not both roles at the same time). As a result, I facilitated the role of Scrum Master, developer, coach, facilitator, teacher, and mentor.

ICGSE '18, May 27–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5717-3/18/05.
https://doi.org/10.1145/3196369.3196388

4 SETTING THE STAGE

Prior to the start of Project XYZ, the team was assembled to complete an existing project for the same client that allowed them to sever ties with their previous software vendor. This project was run in a Waterfall fashion. The team started with myself and was later joined by collocated developers. As an onboarding activity, each new developer paired up with an existing developer. The assumption was that this would continue until the new developer was familiar enough with the application that they could code on their own. However, the newly formed pair decided to continue working together. Not only had they established a good working relationship, but they were also highly productive.

With the success of collocated pair programming, the team came up with an approach to parlay their technique for the distributed team members that would be joining the team in the near future. Essentially, as new distributed team members joined the team they would pair up with me and take on real project work for approximately one to two weeks. During this pairing, something unexpected happened. The collocated pair could overhear the conversation I was having with the new distributed team member and the two sets of pairs would share information in the moment as opposed to relying on email or other non-real-time forms of communication. Instead of being a distraction, the overheard conversations allowed the collocated pair to be even more productive because they always knew what the other pair was working on and at the same time allowed the new distributed team member to get up to speed much faster. Once the distributed team member was familiar enough with the application, they paired with another distributed team member for the remainder of the project. The onboarding and hiring process took approximately four months and resulted with one collocated pair and one distributed pair that did not change throughout the project. The requirements engineer was not involved in pairing.

Once Project XYZ kicked off, I occasionally paired with each pair when I sensed they were getting stuck. However, each pair participated in pair programming over 90% of the time, every day. In rare circumstances where defects were reported, one team member would address the defect while the other half of their pair would continue working on the task at hand.

The distributed pair was separated by a 3-hour time difference. In spite of this, they were able to align their working hours quite closely because the team member in Vancouver preferred to start their work day a little later and the team member in Toronto preferred to start their work day a little earlier.

There were opportunities for the entire team to meet face-to-face. This typically occurred once every four weeks. Essentially, the distributed team members (including the requirements engineer) would travel to Calgary. The team would spend the last of the day of sprint showcasing their work to the client (sprint review) and looking at ways to improve (sprint retrospective). The next day was dedicated to sprint planning. The first half of that day is where the team would review and clarify the user stories with the client. The second half of that day is where the team would re-estimate the user stories and apply tasks to each user story.

5 TOOLSET

As expected, distributed pair programming requires the incorporation of various tools. Table I shows a listing of these tools.

TABLE I. TOOL LISTING

Tool	*Vendor*	*Purpose*
IntelliJ	IDEA	Integrated Development Environment
Jira	Atlassian	Issue Tracking
Confluence	Atlassian	Wiki
Fisheye	Atlassian	Version Control Viewer
Crucible	Atlassian	Code Review
Bamboo	Atlassian	Continuous Integration
Bitbucket	Atlassian	Git Version Control
HipChat	Atlassian	Instant Messaging
Nexus	Sonatype	Repository Manager
Skype	Microsoft	Telecommunications
GoToMeeting	LogMeIn	Online Meeting
TeamViewer	TeamViewer	Remote Desktop Sharing

None of theses tools were used to support the software application prior to the project which required the client to purchase new licenses and also required setup time from the team. All of these tools lived on after the project. Many tools from Atlassian were selected because of they easily integrate with another. These tools also helped to reinforce other XP practices in addition to pair programming.

TeamViewer worked well not just because it provided a remote desktop viewer but it also allowed distributed pairs to take control over their counterpart's computer.

Phone calls via cell phone were typical but Skype was also used. However, there were times where Skype was problematic. This could be because the distributed pairs were working from home and their residential internet connections provided low bandwidth.

GoToMeeting was used for the entire team to connect. This was often used with the client as well. The team also invested in a Bluetooth speaker phone which integrated nicely with GoToMeeting. The quality was exceptional as long as the devices running GoToMeeting had a strong WiFi connection.

6 OUTCOMES

Our newly formed team had no historical data that would allow us to determine our velocity. Essentially, we had to guess at our velocity which started out at 50 story points. In the first few sprints, the team was able to complete slightly more than 50 story points. In later sprints, as the team got more comfortable with distributed pair programming they easily accomplished their goal and their most productive sprint resulted in almost 500 story points. This was largely due to the team's commitment to continuous improvement which allowed them to steadily progress from sprint to sprint. This increased velocity allowed the team to

complete the project three months ahead of schedule which is quite remarkable considering the entire project was only scheduled for ten months.

Another contributing factor to the success was the inclusion of team (aka improvement) stories. A representative from the client (located in the U.S.) performed the role of the Product Owner and allowed the team to take on a team story every sprint that did not exceed 3 story points. This was included in the original predicted velocity of 50 story points which meant that the team expected to deliver 47 story points of business value each sprint. The team was given the freedom to choose whatever they wanted. In some cases, the team incorporated various Java frameworks (i.e. Spring Data/JSF PrimeFaces) that would allow them to increase their velocity in later sprints.

Including the right tools (see Section 5) and using them in the right way definitely helped to increase velocity. These tools helped the team to focus on writing code instead of performing manual tasks. A continuous delivery pipeline was established that allowed the team to deploy to production (or any other environment) by the push of button.

The use of Scrum seemed to work quite well. While the team was skeptical at first, they quickly adopted the various Scrum ceremonies. They enjoyed the 2-week pace and even at one point considered moving to a 1-week sprint. Ultimately, the team decided that they needed to be extremely good at a 2-week sprint before moving to a 1-week sprint and did not feel they were quite there. The prescriptive nature of Scrum provided the team with the right amount of direction. Since the team was new to Agile, I do not believe a less prescriptive Agile methodology would have produced the same results.

The support from the client cannot be understated. They handled the Product Owner role extremely well with very little direction and minimal training. Furthermore, they provided feedback whenever required and went so far as to provide a single tester at all times.

7 ANALYSIS

Distributed pair programming was a huge success. But what is about distributed pair programming that allowed the team to achieve such a high velocity? What aspects of distributed pair programming is highly important? At the completion of the project, the distributed pair took part in an informal survey (prepared by myself) to enlist their opinions on this technique. The findings are summarized in the following sub-section.

A. Why did it work so well?

Effective: There was an overall agreement that this skill was an effective way of performing software development. This was directly related to the velocity the team achieved.

Enjoyment: Team members preferred to perform their work using distributed pair programming as opposed to working in silos.

Courage: Everyone acknowledged that their new-found proficiency encouraged them to do things that they would not do if they were working individually. For instance, some team members hesitated to refactor code because they were scared of breaking something. However, the fear subsided when they performed refactoring as pairs.

Quality: There was definitely a sense of built-in quality as few defects were reported by the customer throughout the project. Additionally, throughout the sprint there were few instances where user stories had to be reworked. Unit, integration, and user interface (UI) tests were all maintained while still accomplishing the work agreed upon in sprint planning.

Confidence: Not surprising, as the velocity increased so did the team's confidence. This is largely due to the fact that were able to over deliver every sprint.

Agile Interest: The distributed pairs felt that pair programming increased their interest in Agile. They sensed that Waterfall projects would likely not see the same benefits from pair programming. As a result, they were open to trying other XP techniques as mentioned in Section 3.

Trustworthiness: Trusting someone that you have had very little or no face-to-face communication is difficult to do. In a distributed manner you will likely not have the luxury of reading your partner's body language (low quality webcams do not provide that) and may have to rely on listening to the tone of their voice or stopping what you are doing to read their instant message (IM).

Social Etiquette: It may not be obvious but social etiquette is an important part of distributed pair programming even though the pairs are not face-to-face. In our case, the pairs were separated by geographical, temporal, and cultural barriers. Accommodating each other's schedule and respecting differences in beliefs was a necessity.

Expressing Thoughts: The distributed pairs felt that it was easier to express their thoughts compared to working in a collocated environment with no pair programming.

Letting go of Obsessions: The importance of listening to your partner even when you feel you are correct is an essential part of distributed pair programming.

Enough Sleep: Since it is common for distributed pairs to live in different time zones, they may be required to start their work day earlier of end their work day later than normally expected.

B. How we could have improved?

Travel: A travel budget was established and it did include travel so that distributed team members could experience collocation with each other and the rest of the team. I do feel that if this had happened more frequently, the distributed pair would have felt like they were part of the organization as opposed to a sub-team.

Tools: The tools were selected prior to establishing pair programming and as such did not have distributed pair programming in mind. That is not to say they did not work well for distributed pair programming. However, we could have investigated tools that had more support for this technique. Even though the team generally enjoyed coding in the IntelliJ IDE, open source options like Eclipse offer plugins like Sangam [1] and RIPPLE [8] to assist with distributed pair programming.

Training: While the team had a resource with Agile experience to rely on, it is my opinion they could have benefitted from formal

upfront training. This training could have provided the right setting to build the Agile mindset from the beginning as opposed to trying to manufacture it throughout the project.

Cloud Offerings: Whether it's IAAS/PAAS/SAAS/etc., cloud offerings make a lot of sense for so many reasons. The maintenance of a hosted environment is a burden that development teams do not need. One of the major benefits is that localized outages are not affected by cloud offerings. Unfortunately, this is not always possible. Strict regulatory compliance may require that all tools and environments be hosted.

C. Why would I recommend Distributed Pair Programming?

Knowledge Transfer: When left to fend for themselves, new team members can spend a lot of time reading endless documentation when they could be contributing to the team. Sometimes the travel budget is not adequate to have a distributed team member spend weeks with the collocated team. In these situations, pairing up with another team member allows them to learn faster and start contributing sooner. This also acts as a form of risk mitigation in the event that team members leave the team.

Encourages "Jelling": Based on my earlier experiences, distributed team members working in silos are often treated as second class citizens. It is extremely difficult for them to feel like they are part of the team. When pair programming occurs, these individuals start to feel like they are part of something and productivity is sure to follow.

Improved Quality: Distributed pairs end up doing things they would normally not do and that includes holding each other accountable. The act of joint coding forces the pair to encourage each other to perform test driven development (TDD), think about the implications of their design decisions, refactor whenever possible, etc.

D. What are the problems we encountered?

Lack of Professionalism: There are times when distributed team members need to be visible by the rest of the team and sometimes by the client. In these situations, the distributed team members simply turned on their webcams just as they would when performing pair programming (webcams were normally used all the time). This can present many problems. We witnessed situations where the distributed team members were dressed inappropriately, their kids/pets were making a lot of noise, and their homely background setting was insulting to those that made it to work every day. That is not to say the collocated team members wanted to work from home, just that they put in the effort to present themselves in a professional manner and expected the same from their distributed counterparts.

Success can also present Challenges: At one point in the project, one of our distributed pairs relocated to the same city as the collocated team members. Instead of joining the team in the office, the individual decided to continue working from home and pair program with his distributed counterpart. At this point, collocated pair programming was completely ignored due to the success of distributed pair programming that the project had already received. The individual decided that distributed pair programming was working adequately and did not see the need to travel to the office. This situation was allowed as the project manager gave consent prior to the move. The collocated team members were mystified because they felt the team would benefit by having another collocated team member.

Stuck!: Getting stuck happens to everyone. Typically, the correct course of action is to ask for help. Unfortunately, this can be difficult for distributed pairs to take advantage of. They may feel uncomfortable with this approach because they are somewhat removed from the rest of the team. This can result in endless hours or possibly days where they deliver minimal value.

8 LESSONS LEARNED

The pair programming concept involves two roles; the Driver, and the Navigator [2]. This has worked effectively in collocated pair programming for a long time. But would the same model work with distributed pair programming? We learned that both roles are still necessary. In fact, the roles do not really change. However, it is essential that the distributed pair check-in with the rest of the team on a regular basis so that they are aware of design and architecture changes. We found that a check-in worked well when the distributed pair started working on a new user story. Essentially, all developers would join a GoToMeeting and quickly discuss.

The accomplishments of this project cannot be understated. The client was extremely happy, the new vendor was happy that their new client was extremely happy, and the team was delighted. Unfortunately, it is rare for software projects to realize those three situations. While we experienced tremendous success with distributed pair programming we must admit that it is not well suited for all projects. Some projects that may benefit include:

- Mix of collocated and distributed pairs
- Ensuring there is adequate budget for travel and team building events
- Implementing and maintaining the right tools

The same can be said for human resources. Not everyone is well suited for distributed pair programming. Individuals that may be a good fit include:

- Willingness to try Agile and Agile related views even if it seems like a foreign concept
- Commitment to continuous improvement
- Acceptance that working in silos will not produce great results

9 CONCLUSIONS

While collocated pair programming is quite popular, not all individuals are a good fit. The face-to-face nature presents many nuances that some people are not comfortable with. In some cases, these individuals are sometimes marginalized and exiled from the team. Another option is to have them participate in distributed pair programming where the social aspects are quite different. Some individuals may exhibit health conditions or lack of social etiquette (e.g. heavy breathing, body odour, etc.) that make it very difficult to pair program side-by-side. Others are simply uncomfortable working in close proximity with others. For these reasons, distributed pair programming can produce better results

because some individuals cannot partake in collocated pair programming effectively. In fact, their only options may be to perform distributed pair programming or work alone.

When the team started down the path of distributed pair programming there was a lot of uncertainty. Not just because the team was unfamiliar with pair programming but also because there is not a lot of research on distributed pair programming. More research (and literature) is needed in this area. Many projects have incorporated this technique but they have not documented and analyzed it.

There are many things this project did not experience. And with lack of research it is difficult to determine the outcome of certain situations. For example, some individuals may request to work part time. Does distributed pair programming require full time participation? This is an important question to answer for the purposes of recruiting. Also, what are the effects of turnover? What happens to the remaining team member when this occurs? Do they work in a silo? Do they act as a third member to an existing pair?

The results of this project may not always be repeatable. Pair programming and distributed pair programming is not necessarily a good fit in all situations as illustrated in Section 8. In fact, there is research that suggests pair programming does not provide any advantage as indicated by Nawrocki and Wojciechowski [3], Vanhanen and Lassenius [4], Arisholm et al. [5], Rostaher and Hericko [6], and Hulkko and Abrahamson [7]. Additional research to explain the discord may also be necessary.

ACKNOWLEDGMENTS

A special thanks to my shepherd, Maria Paasivaara, I could not have done it without you!

REFERENCES

[1] C. Ho S. Raha E. Gehringer L. Williams "Sangam—A distributed pair programming plug-in for eclipse" in Proc. OOPSLA Workshop on Eclipse Technology Exchange, pp. 73-77 2004.

[2] K. Beck and C. Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley, Boston, 2004.

[3] Nawrocki, J. and Wojciechowski, A., 2001. Experimental Evaluation of pair programming. In: Proceedings of the European Software Control and Metrics Conference (ESCOM 2001). ESCOM Press, 2001, pp. 269-276.

[4] Jari Vanhanen and Casper Lassenius, Effects of Pair Programming at the Development Team Level: An Experiment, 2005 IEEE

[5] Erik Arisholm, Hans Gallis, Tore Dyba, and Dag I.K. Sjoberg, Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise, IEEE Transactions on Software Engineering, Vol. 33, No. 2, Feb 2007.

[6] Matevz Rostaher and Marjan Hericko, Tracking Test First Programming – An Experiment, XP/Agile Universe 2002, LNCS 2418, pp. 174-184, 2002

[7] Hanna Hulkko and Pekka Abrahamsson, A Multiple Case Study on the Impact of Pair Programming on Product Quality, ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA.

[8] K. E. Boyer A. A. Dwight R. T. Fondren M. A. Vouk J. C. Lester "A development environment for distributed synchronous collaborative programming" ACM SIGCSE Bull., vol. 40 no. 3 pp. 158 2008.