# Success through Failure: How we got Agile to work on a Distributed International Team

**Erin Chapman**

erin.chapman@gmail.com

## Abstract

Multinational teams are becoming the standard as companies compete globally. We hear stories of US-based teams who prefer Agile development practices but who encounter growing pains, including cultural and time zone differences, when they add overseas contributors.

Engineering teams want to collaborate and work well together regardless of how they may be distributed. This case study delves into the problems we experienced on one team spread across two continents (in United States and India). To bridge the differences and solve these problems, we had to take the time to really communicate with one another about how each site interpreted Agile concepts, what they saw as the main obstacles to development, and how we could best solve problems without sacrificing our priorities.

The solution was not for the team members at one site to issue a process to the entire team; instead we collaboratively reached a solution that ensured communication between sites, decreased turnaround time, improved quality, and received buy-in from the whole engineering team. Some of the practices we have put into place could assist other teams in being successful working across development sites.

In the end, we found that Agile development can work internationally if the process has enough structure, team knowledge is made more explicit, and everyone's voice is heard.

## Biography

*Erin is a software design engineer at Tektronix where she is involved in software development, scrum process management and software testing. She has an M.S. in Computer Science from Portland State University and B.S. degrees from Rensselaer Polytechnic Institute in Mathematics and Computer Science.*

# Introduction

You can find many companies switching to Agile methods these days from the traditional waterfall process. Tektronix is one of these companies, having made the switch several years ago to Scrum. While we've generally had success with our collocated teams, regardless of their location, it's been a mixed bag when it comes to distributed teams, even within the same time zone. This mirrors the experiences that we hear from other companies and developers.

## 1.1 Distributed Scrum Teams

When Scrum was first proposed, it was recommended against using it for distributed teams. One can find many industry experts, such as John Puopolo arguing that Scrum should only be used with collocated teams [4]. Over the years, it has become more of the norm for a company to have multiple teams across multiple sites working on the same project. With this has come a variety of models for managing a distributed team. Teams can choose whether they have a separate scrum team at each site or not, how many Scrum Masters and at what points the teams coordinate. You will most likely see one of three versions of scrum: One Scrum run across all sites, one Scrum at each site that are coordinated via a **Scrum of Scrums**, or one Scrum at each site that are not coordinated and run independently of each other. Each method has its strengths and weaknesses, and which method you choose depends on the specific circumstances of the project [1].

## 1.2 International Teams

Adding the international element additionally complicates the distributed Scrum team arrangement [5]. Each site may have a different management structure, as is the case at Tektronix. Along with this, the sites may have different work flows that do not mesh with each other or complicate the coordination between teams. At Tektronix, it is not uncommon for management at one site to have a different definition of what progress looks like and push their team to meet their definition, while another site has a completely different view. When working in a single code base, this can cause the team to disagree with each other on all parts of the development process.

Additionally, **communication** can more easily fall by the wayside with international teams. When working across time zones where teams do not have a common day, the **increased turnaround time** and **mental distance** will decrease the amount of time teams spend talking to each other. Tools such as instant messaging, e-mail, and teleconferences can help overcome these issues, but only if the team is cohesive at the start. [3].

## 1.3 Historical Trends at Tektronix

Tektronix has been slowly making the switch from a traditional waterfall style process to Scrum over the past several years. The transition has been somewhat piecemeal, and how Scrum is run depends heavily on the department. Within the department that the XYZ team worked, all of the scrums were run independently and with varying levels of coordination. The Scrum teams at various sites often did not communicate until close to a customer release date, when major issues would often appear. Scrum teams at the same site sometimes coordinate their work, but often only because the Scrum Master was the same person for both teams.

During a sprint, code could be delivered to the development branch in the source control repository at any time, without any verification testing. All testing was done in a specific hardening sprint as the product release occurred. Frequently, incomplete features would have to be removed near release time due to too many last minute issues discovered in the code.

# Our Story: The XYZ Team

When the XYZ team was formed to work on a new product, they were given the opportunity to change not only the tools that they used to do their job, but also develop a new work flow for development. The XYZ team consisted of two groups at two different sites: the Beaverton team who would be working on the core infrastructure for the new system and the Bangalore team who would be writing algorithms to do data analysis. The original plan, as laid out by management, would be that the Beaverton team would get a head start on creating the infrastructure and once a sufficient base was in place, the Bangalore team would be spun off fairly independently on writing the algorithms.

As is often the case, the best of intentions often go awry. The Beaverton team imposed strict code quality standards on not only themselves, but the Bangalore team. The Bangalore team was brought into the project before the base infrastructure was stable. Both teams had very different development processes in mind and very different ideas of what is “agile”. The result was a **jarring discord** between the two teams that threatened to derail the project. This is a look back at how we got to that point and how we moved forward to become one of the best distributed, multinational teams at Tektronix.

## 1.4 Founding the Beaverton Team

The Beaverton team was started with four members to work on the core infrastructure of the new project, O, R, V and N. They were given the opportunity to choose the toolset everyone on this project would be working with. Given that freedom, they decided to aim for their dream build environment that would help the team reach high quality goals.

Historically, the development environment had been somewhat of a **free for all**. Each developer had a local copy of the code that they could work in, but everyone committed their code to the same branch on the server, without any gatekeepers as to what would and would not go in. One build would run at 6 PM PST with whatever was checked in and the results would be e-mailed to the team (except only one team member reliably read it).

Frequently, one person would check in their code and move on to the next thing they needed to do. The next morning the build e-mail would be deleted by everyone and no one would notice the broken code until the complaints built up. And that was if the build actually failed. Sometimes errors would only be discovered during final testing before a release and an entire feature would need to be pulled at the last minute.

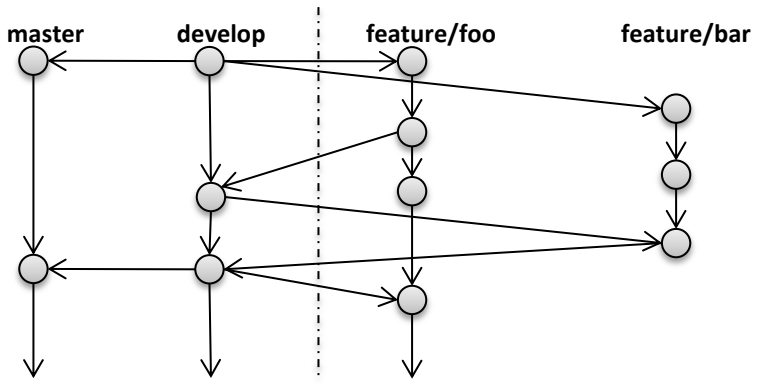


Figure 1: The branching model used to keep work in progress (feature branches right of the dashed line) separate from completed work (**develop**) and released work (**master**)

The Beaverton team wanted to get away from this mode and move to a model with continuous integration, testing and gatekeepers so they could produce a high quality product. They started by choosing Git for source control, with Stash from Atlassian to manage the repository. To preserve code stability, they used the **Git flow** [2] branching model to keep all work in progress in isolation. All work is

done in a **feature branch** by the developers. Each individual feature has its own branch so it can remain in isolation until complete. When the feature is done, fully tested and meets all requirements, it is reviewed and merged into the develop branch. The develop branch contains all completed features that have not yet been released to a customer. Finally, develop is periodically merged into a master branch which is then released to customers. All code in develop and master was expected to be stable at all times, even during a sprint.

To prevent code from being released before it was complete, the Beaverton team adapted a **strict rule set** enforced by their tools. First, they restricted who had permission to push code to develop and master to a small set of people. This requires all merges to develop to go through a **pull request** in Stash (essentially a peer review). In Stash, no one is allowed to merge their own code. This allowed the team to insist that all code go through peer reviews, one of the best ways to catch defects early [6]. In addition to requiring a peer review, they also configured **Stash** to tie into the **Jenkins** automated build system. Jenkins built their code across three platforms (Windows, Linux and Mac OSX) and ran unit and functional tests across all platforms as well. If the build or any of the tests failed on any platform, Jenkins would report an error to Stash and block the merge until it had a successful build. The team also collectively decided that all comments in a **pull request** had to be addressed before the code was merged.

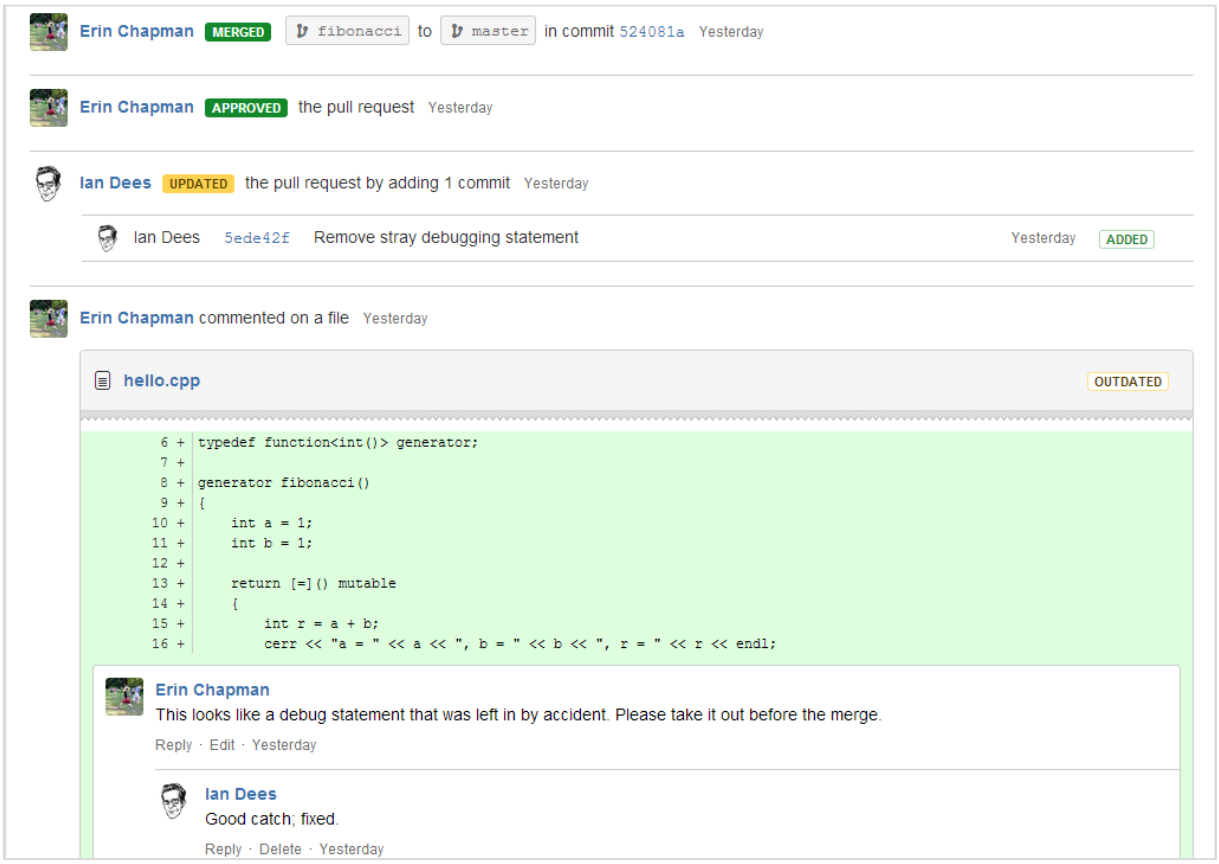

Figure 2: A **Pull Request** in Stash showing a typical interaction

During this initial project startup phase, the Beaverton team experienced some growing pains. Initially the team was not collocated together in the building. O and R, having worked together on a previous project, were in cubicles somewhat close to each other, while V and N were on another floor. While the team was seated apart from each other, they found it difficult to frequently work together as a team or in pairs. This set up some initial habits of working as individuals on tasks that created barriers to communication later. After several months on the project, the team was able to **collocate** in a section of the building together. Once they started sitting together, there continued to be communication issues for a short period. Not all

of the team members would attend the **Daily Stand Up** and there was a lack of awareness as to what each person was working on.

After a month of rough communication, the team reaffirmed the importance of the Daily Stand Up, which was attended by all team members going forward, and also started a group work time in the afternoon where the team would leave their cubicles to work around a table together.

## 1.5 Bringing in the Bangalore Team

After the Beaverton team had been working on the infrastructure for roughly eight months, there was **pressure from management** to bring the Bangalore team (five members: Z1, F, C, E and Z2) on line to start work on the algorithms. Because the core of the project was still relatively unstable, the Bangalore team started working in their Git repository with the base system pulled in as a component. To get them started with this work, the Beaverton team provided a demonstration of the APIs and wrote several how-to documents on the architecture, requirements and data types.

The Bangalore team began their sprints in sync with the Beaverton team to work independently on the algorithms. During this time there was minimal communication between the two teams. N would attend a regular phone call with the Bangalore team but little of that information made it to the rest of the Beaverton team. The remainder of the Beaverton team had little or no contact with the Bangalore team.

## 1.6 Disagreement and Discord

After several months, the Bangalore team sent a copy of the algorithms in progress to the Beaverton team for a quick review. Upon receiving the code, the Beaverton team had **concerns about specific requirements** being forgotten or misunderstood, as well as **conformance to the team coding standard**. The Beaverton team e-mailed their concerns to the Bangalore team for revision.

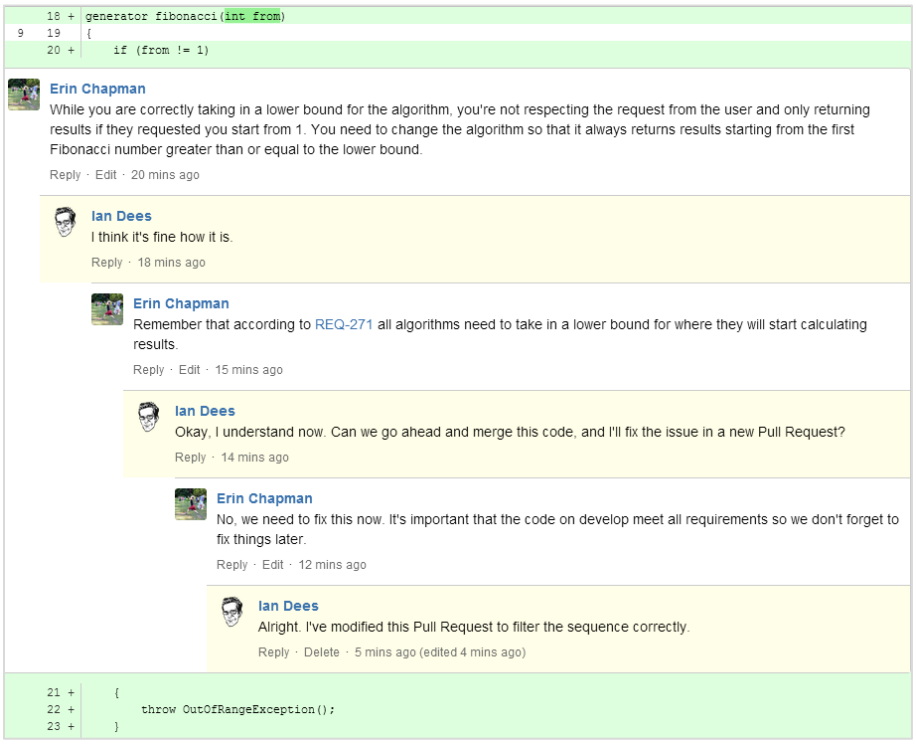


Figure 3: An example of early interactions between the Beaverton and Bangalore teams

Around this same time, it was decided to pull all of the algorithms into the same Git repository as the core infrastructure. At this time, the Beaverton team started reviewing all of the pull requests created by the

Bangalore team. Because of their various quality concerns, the Beaverton team insisted on **reviewing all code** and did not give anyone on the Bangalore team merge permissions. When they felt the issues were not being addressed, the Beaverton team required any concerns to be addressed in the current pull request, rather than merging the code as-is and having the author make the necessary updates in a future pull request. The Bangalore team, meanwhile, wanted to push on bringing in an initial version of all of the algorithms and address all of the requirements and review comments after the first pass was complete. The Beaverton team, in part due to historical issues with incomplete features and having to remove work in progress from the code base immediately before a release plus a tight project schedule with little time to rework issues later, had very strong feelings against this particular work flow and stopped merging code. Because no one was willing to budge on their process, all work on the teams stopped and angry e-mails started flying back and forth between the two groups and their managers.

## 1.7 Time Out to Talk

As the e-mails escalated and no code was being merged, the team hit a hard stop for roughly a month where little work was completed. It was soon evident that this was not going to be solved via e-mail, so the Beaverton team started scheduling conference calls with the Bangalore team to discuss what was going on.

The first meeting between the teams was a review of the requirements, which it seemed like everyone understood but there was **disagreement on how the requirements were being implemented** by the Beaverton team. The Beaverton team, on the other hand, felt like those decisions were behind them and understood how that design had been reached, so they were eager to end discussion of the requirements and move on to the next agenda item. This particular issue would crop up several times, until the Beaverton team explained the various designs they had considered and why the particular design they had chosen was the only one that met all requirements. Once that understanding was reached, this particular issue disappeared.

The rest of the meetings all related to the **development process**. What emerged was that each Scrum team had a very different idea of what the development process was. The Beaverton team was looking for each piece of functionality (infrastructure or algorithm) to be **complete** before being merged into **develop**. To them, complete meant that it implemented all of the requirements, had unit and functional tests, used current C++11 best practices and passed a peer review with no one else on the team having concerns (admittedly, the Beaverton team set a high bar for both the Bangalore team and themselves). The short summation of the Beaverton team's idea of agile: only customer releasable code is in **develop**.

On the other hand, the Bangalore team wanted to create a **basic version** of each algorithm and then go back and **add additional features** one at a time to meet all of the requirements. Testing the algorithms would be done after all of the requirements were implemented. The Bangalore team also preferred automated testing to peer review. The team leads there felt that automated code review was better at catching issues than a review by other developers. There were also feelings that it was unfair that no one in Bangalore had permission to merge the code and only Beaverton team members did. The Bangalore team felt that only **master**, not **develop**, needed to contain releasable code and that being agile meant frequent, small pushes to **develop**, regardless of completeness of code. When release time came around, the completed code could be separated from the incomplete code and be merged into **master**.

The Beaverton team had strong feelings against this type of development process as we had historically run into problems come release time at Tektronix. After further discussion, we determined that the teams had different definitions of when an algorithm was done. This realization gave them a place to start in an attempt to find middle ground.

## 1.8 Process Changes

The Beaverton team started with two major changes. First, R took responsibility for the relationship with the Bangalore team instead of trying to pass information via N. Since R was the one reviewing all of the code, it made sense for her to be the one interacting the most with the Bangalore team, especially when it

came to planning and communicating requirements. Because of the streamlined communication flow, the correct information made it back and forth between folks and R could confirm with the Bangalore team whether they understood what was being asked for.

As part of this, R made herself available to the Bangalore team for support. It was made clear that anyone could e-mail R with questions or concerns, but if it was felt that the issue could not be resolved via e-mail or code review, an on line meeting could be scheduled. At first, no one on the Bangalore team took R up on the offer, but after R started scheduling one on one meetings with folks on the Bangalore team when issues came up during code reviews, the Bangalore team got to the point where they feel comfortable putting meetings on R's calendar to discuss requirements, work in progress or issues that came up during a code review.

R and O also started phoning into the Bangalore team's **daily stand up**. After a while, it became clear that R and O were doing a scrum of scrum style meeting with Z1, so the team switched to doing a scrum of scrum meeting twice a week for just the three of them attending.

To help clear up the confusion about what was being looked for during code reviews, the Beaverton team created two **checklists**. The first checklist was a list of items they consider when reviewing the actual code, made as concrete as possible. This included items like using C++11 types, such as smart pointers, nullptr keyword and enum classes; using class APIs instead of writing your own function, correct usage of data types and writing Doxygen headers for classes. The second checklist included a list of unit and functional tests that would be used to verify the code meets requirements. After releasing these checklists, the Bangalore team reviewed all existing code, making huge changes and removing many of the concerns the Beaverton team had about code quality.

During this process, the Beaverton team also realized that they were not **recognizing the good work** done by the Bangalore team. While the Beaverton team is used to being critical of each other's work, they realized that just being critical of the code written by the Bangalore team may not go over the same as the relationship is not the same.

The relationship between Beaverton teams and Bangalore teams has always been a bit strained. The Beaverton engineers are typically recognized as being senior to the Bangalore engineers. Additionally, the Beaverton engineers tend to make the decisions and inform the Bangalore engineers what they will be doing. This leads to the Bangalore engineers feeling like they have little say in what happens. Coming in to a situation where you were working independently before, into one where not only are other people making the decisions, but they’re pointing out you are doing wrong, can be hard to take, regardless of where you are.

After recognizing how difficult this could be on the receiving end, the Beaverton team decided to make a concentrated effort to not just point out issues in **pull requests**, but also to point out things that were done well and thank developers for making changes or responding to comments.

## 1.9 Current Process

At this point, the Beaverton and Bangalore teams have a relatively smooth process in place that has been working well for over half a year. About a week before a new sprint, R, O, Z1 and A (the product manager in Bangalore) get together and discuss the upcoming goals for the sprint. O and R check if the Bangalore team is going to require anything in the next two to three months from the Beaverton team and let Z1 and A know if they're going to need anything from them. The two teams then do their respective sprint planning and send each other the lists of tasks pulled into the sprint afterward to confirm that both teams are still aligned. During the sprint, these four meet on the phone twice a week to discuss what each team is doing and make sure there are no issues or blocking items that need to be addressed.

Once the sprint starts, **early communication** is heavily emphasized. If anyone has questions about an algorithm or an architecture piece, they know the expectation is to ask questions before starting work. At this point, the Beaverton team gets questions from developers on the Bangalore team on a weekly basis (most importantly, before code is written or as soon as the question comes up, instead of waiting).

Both teams have discovered that **code reviews** in Stash are a great communication tool as part of their early communication efforts. If a Bangalore team member has a question on work in progress, or a Beaverton team member is filling out some architecture needed by the other team, developers have started opening **pull requests** before code is ready to be merged so that the teams can have a discussion in the actual code about what is being done. These have been some of the best conversations between the two teams, sometimes going back and forth for a week or more on design.

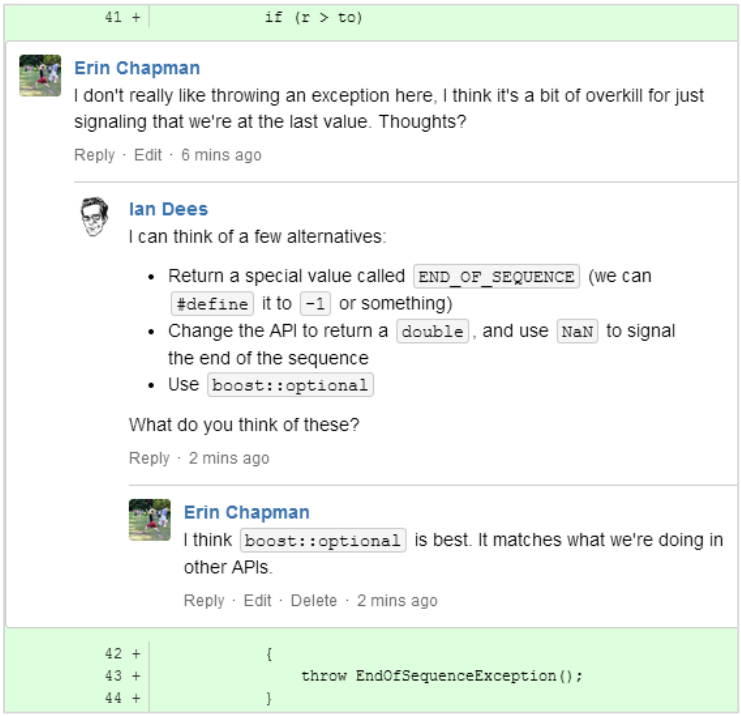


Figure 4: **Pull requests** for design discussion between Beaverton and Bangalore team members are now common

Along these same lines, when a **pull request** is opened for a final review and merge, comments are no longer viewed as a negative thing. The team has had some **pull requests** remain open for days or weeks while debating some point or discussing requirements. If a complicated merge needs to happen or an API has changed, everyone trusts that others will help out. The attitude is no longer merge now and figure it out later, but one of wanting to reach a conclusion about what's best here. Questions or concerns about code are not viewed as a personal attack on the developer, but wanting the team as a whole to produce high quality work.

The biggest gain seen between the two sites is probably with how well the teams talk with each other. Every developer at each site is willing to talk with anyone else. E-mails with questions about requirements and code go back and forth on a daily basis. If a solution can't be reached via e-mail quickly or someone suspects that there is miscommunication, a conference call is rapidly scheduled. Surprisingly, it's not just the Beaverton team scheduling meetings with the Bangalore team anymore. The developers on the Bangalore team have reached a point where they feel comfortable setting up meetings of their own to discuss code.

Since the communication has become smoother between the two teams and the perceived code quality has improved, the Beaverton team has even reached the comfort level where they feel confident in assigning work to the Bangalore team. This is a widely divergent place from where the team was a year ago, and was mostly resolved by fixing various communication problems between the two teams.

# Lessons Learned

The failure to do use Scrum successfully and produce high quality code really came down to a number of **communication** problems, both between Beaverton and Bangalore and within the Beaverton team.

- **There Can Never Be Too Much Communication:** Scrum emphasizes communication as a major part of enabling the development of quality software. With the Beaverton and Bangalore teams, we learned that erring on the side of over communication, especially early in a project, can reduce the speed bumps hit later on. The Beaverton team adapted a policy of e-mailing the Bangalore team before making any changes that may affect them (since changes like that were typically one way on this project) and again when the changes were merged. Any information that the other team might want to know was e-mailed out immediately. Sometimes this produces a high rate of traffic in between the two teams, but there have been few surprises since then.

- **Learn How the Other Team Wants to Communicate:** Part of communicating with the other team, involves communicating with the other team effectively. This is a learning curve and involves adapting what one typically does. With this project, the Beaverton team learned that phone conversations get complex information over to the Bangalore team better than e-mails. They also learned that some of the more junior team members in Bangalore aren't comfortable asking questions during group conference calls, so they stay on line for a while after each call in case someone wants to instant message them with questions privately.

- **Get Everyone Comfortable With Talking:** The Beaverton team noticed that while they would say that they could be contacted directly by anyone with questions it rarely happened. Instead, they had to make it clear that no one would get in trouble for contacting them and they'd prefer to set up phone calls to talk with the Bangalore team in the mornings or evenings. A concentrated effort had to be made initially to communicate one on one with the developers on the Bangalore team, even if it was just a quick check in e-mail, until everyone was comfortable initiating a conversation

  Along these same lines, noticing who is comfortable talking to who plays a large roll. Two of the developers are more comfortable talking with R than with O, so O keeps one on one contact short and positive, while R handles more complex issues or things that may be perceived as being critical..

- **Personal Relationships Matter:** Initially there were no relationships between the Beaverton and Bangalore teams; no one had any idea of who was on the other end of the phone or keyboard. This made some of the conversations between the two teams very confrontational. Once the two teams made an effort to get to know each other, the conversations softened. On conference calls now the teams ask about families, talk about vacations or other activities in their colleague's lives. Knowing who is on the receiving end directly effects how you interact with them.

- **Defined Structure for the Process:** While Scrum typically prefers a loose, less defined process, when working across sites, a more defined process may help. Explicitly listing out the scrum of scrums process, plus the checklists for code reviews, and putting subtasks on tickets for “talk to developer X on the Beaverton team before starting work” has helped to set the expectation for what is being looked for by the Beaverton team.

# References

[1] Anderson, Rick D., 2013. “Scrum in a Land Far, Far Away… (or Using Scrum Across Dispersed Sites).” *31st Annual Pacific Northwest Software Quality Conference*, pages 79-91. http://www.uploads.pnsqc.org/proceedings/PNSQC_2013_Proceedings.pdf

[2] Driessen, Vincent, 2010. “A Successful Git Branching Model.” http://nvie.com/posts/a-successful-git-branching-model/ (accessed August 3, 2013).

[3] Kiesler, Sarah and Jonathon N. Cummings, 2002. “What Do We Know about Proximity and Distance in Work Groups? A Legacy of Research.” http://www.ece.ubc.ca/~leei/519/2002-ProxmityDistanceWorkGroup.pdf (accessed May 5, 2014).

[4] Puopolo, John P., 2007. “Be There or Be Square.” Scrum Alliance. http://www.scrumalliance.org/community/articles/2007/august/be-there-or-be-square (accessed May 16, 2014).

[5] Simons, Matt, 2002. “Internationally Agile.” InformIT. http://www.informit.com/articles/article.aspx?p=25929 (accessed June 2, 2014).

[6] Sommerville, Ian, 2011. *Software Engineering*. Boston: Addison-Wesley.