How Did We Adapt Agile Processes to Our Distributed Development?

Cynick Young
Intelliware Development Inc.
cynick.young@intelliware.ca

Hiroki Terashima
Technology Enhanced Learning in Science
hirochan@berkeley.edu

Abstract

Today, many software projects are being developed by collaborating programmers working across multiple locations. Whatever the reason may be, outsourcing, organizational structure, or external collaboration, these projects often suffer from the physical separation of developing across the city, across the country, or around the world. Such distances intensify challenges such as peer communications, shared understanding between teams, and systems integration [1]. This paper describes how three organizations adapted agile processes to overcome barriers such as multiple time zones, mixed cultures, mismatched schedules, and limited travel budget to frequently deliver successful software releases.

1. Introduction

The three teams referred to in this paper represent three different organizations: University of Toronto (Canada), University of California at Berkeley (California), and the Concord Consortium (Massachusetts). All three teams were experienced in developing educational software and all were engaged in designing a platform called the Scalable Architecture for Interactive Learning (SAIL)[1]. Toronto had two members, while Berkeley and Concord each had three members working on the project. Each team had different customers, goals, and delivery schedules.

In September 2006, the Toronto team began a new effort to develop a portal web application for SAIL. This portal was to be written in Java and allowed the creation and management of SAIL artifacts such as users, groups, permissions, and project implementations. The portal also served as a launching point for the existing Java desktop applications that students and teachers used in the classroom to interact with the learning objects. The team at Berkeley joined the portal effort in March of 2007, by which time most of the fundamental architecture of the portal was in place. The Berkeley team was to extend and customize the portal to suit their needs, which were different than those of Toronto. The programmers at Toronto and Berkeley had not met previously; thus, a substantial effort was required to build an effective communication model and trust between the members [2]. A similar effort had to be made when the Concord Consortium joined the team in the summer of 2007. Today, the portal web application has been successfully deployed and is actively used at all three sites. Work is still progressing by the distributed teams to extend its functionality.

The Toronto team was the only one that had full agile development experience prior to the collaboration. Concord had previously used short development cycles (due to their research and prototyping focus) and a continuous integration tool, but was not practicing agile development to its full extent. Toronto, with their experience, had suggested using eXtreme Programming (XP) to collaborate and to deliver quality products. Suggestions of incrementally deliverable software, constant feedback, test driven development, pair programming, and other XP practices were amenable to Berkeley and Concord.

Our iterations were atypical and deviated from the textbook XP practice. Release dates were driven by our customers' requirements that included 1) demonstrations of new features at international conferences and 2) deployments for use in the classroom with various participating schools at each site. Most major releases occurred approximately every two to three months using one or two iterations during that period. Although the iteration lengths were variable from one to the next, the scope of work had to be negotiated in order to meet the hard deadlines.

[1] SAIL is an open-source framework for software, data schemas and web services that facilitate the creation, deployment, and assessment of digital interactive educational content. Numerous educational tools are written in multiple programming languages, such as Java, Ruby on Rails, and PHP, and get deployed and used in classrooms by teachers, students, and researchers across the world. Website: http://www.telscenter.org/confluence/display/SAIL/Home

This paper reports on the numerous challenges the distributed teams faced during the course of their collaboration, and the solutions the teams developed. In particular, we focus on:

- How and why we emphasized effective communication
- How we built trust with remote team members
- How we made architecture and design decisions to maximize productivity

2. Emphasis on Effective Communication

There were many options available for communication. We explored a number of these options to select the one that worked best for each situation. For example, we adopted a policy of being available to each other for quick feedback as much as possible with the use of instant messaging. If one team was blocked from progressing due to technical or design issues, a meeting had to take place immediately. These spontaneous meetings were always started by one party contacting the others on Skype or iChat and asking if they were available for a certain amount of time to discuss the issue. This allowed the person being requested to prepare for the meeting (albeit with a short notice) or to suggest a later time. A tremendous effort was made by all parties to have overlapping work schedules across all the teams, even with time zone differences. We made it an explicit goal to be available synchronously as much as possible, and this allowed the distributed teams to feel confident that we could always be in touch.

Working with people in different locations created challenges when intimate code discussions were required. When Berkeley started programming on the portal, they clearly needed to discuss parts of the existing code with Toronto. Copying and pasting portions of code and discussing them through email or text chats quickly became laborious. Audio conferencing used for discussions while each party was looking at their own copy of the code was better; but this too, became cumbersome because not all participants' screens were in synchronization. After several attempts, the teams found that desktop sharing combined with video conferencing proved to be the most useful technique for discussing code. The initiator of the discussion usually became the sharing host, or the Virtual Network Computing (VNC) server, and walked through the code while the others connected to the host's screen as VNC clients. This allowed the host to step through the code using their Integrated Development Environment (IDE) and took advantage of features such as file-indexing, bookmarking, and debugging. Along with the shared desktop, team members communicated using video conferencing. This technique provided the closest simulation of being together physically. During these virtual pair programming sessions, the host would often check to ensure that the other partners were following along and took breaks to organize their thoughts.

Real-time face-to-face meetings through video conferencing allowed us to have a more honest and efficient conversation [3] and permitted the host to gauge participation via body language. One difficult situation with which we dealt was a lack of participation from one or more members during these collaborative meetings. When the meetings were held on text or voice chats, it was unclear why people were being quiet. Were the others thinking, did they have questions, or were they distracted? The teams were left to wonder how long they should wait before asking if the discussion should continue. Video conferences removed most of those doubts. In addition, being able to see each other allowed us to be more sensitive when doing code reviews.

The teams tried to have virtual stand ups as much as possible, schedules permitting. These briefings were a good opportunity to ask and answer questions and possibly set up more formal meetings between individuals. Weekly meetings were held that involved the customers. This gave an opportunity for the customers with different goals to negotiate priority. A wiki space was used to propose weekly meeting agendas in advance. This provided an equal opportunity for everyone to direct the meeting. A separate wiki space was used to take meeting minutes. We had used asynchronous communication tools such as a wiki, a bug management system, newsgroups, and email to document history and preserve discussion trails.

One of our communication success stories came when the Berkeley team was developing a Grading Tool for teachers to use within their version of the portal application. Developing this tool required collaborative work in the portals by Toronto and Berkeley, client application work by Berkeley, and remote data storage work by Concord. In addition, Berkeley's customer had a very aggressive delivery deadline due to the tool's high business value. The biggest challenges for Berkeley were 1) convincing the other customers of the value in this new tool 2) prioritizing the other teams' work to deliver on schedule and 3) ensuring the systems integrated properly. Immediately following the new customer requirement, Berkeley created a public document describing the effort and time line required from each team. Through email and video meetings, the other teams were convinced of the value brought to each team by the new functionalities. The teams began working quickly and met as often as necessary using the techniques discussed above. The teams used a wiki page to merge and prioritize all of the work into one backlog and entered all the tasks into our shared bug ticketing system and tagged them “gradingtool”. All three teams were allowed to edit the backlog and the tickets. The Grading Tool was successfully delivered on time. In retrospect, we think that our distributed teams were successfully agile because we had established effective communications.

3. Building Trust with Remote Team Members

The teams were distanced some three thousand miles and three time zones apart, with no initial opportunity to meet in person. We were asked by our customers to produce a portal web application to suit all three organizations. During the first three months, the teams focused on acquainting with each other and beginning the process of collaborative development. Each team also worked on other projects and that meant time spent on the joint venture was constrained. In addition, each team had an area of specialization and had components for which they were responsible. The outcome of the joint project depended on every team’s ability to deliver successfully. With our distributed teams, this meant mutual trust between each other. Building this trust was not easy.

When the developers first began technical discussions, we found that some members were being too professional and tense. It was hard to discuss anything except work, and every meeting resulted in information overload. Realizing that this was not a healthy way to collaborate, we began incorporating a short greeting period at the beginning of each meeting. For example, at the start of each virtual pairing session, each team tried to provide a physical presence for the other by showing our environment through the webcam. We discussed ordinary subjects such as the weather, each person’s wellness, and anything else that came to mind. This way of starting each meeting allowed everyone to relax and get a feel for how the other members were doing that day, which contributed to a more pleasant meeting.

At the beginning of the collaboration, whenever a meeting was called, the agenda was general because the project was at an infant stage. These meetings usually ended with little accomplished. After several of these initial meetings, the teams agreed that every meeting needed a clear agenda and an expected duration time. For example, when Toronto called for a meeting to take Berkeley and Concord on a virtual coding tour, Toronto announced the goal of the meeting, the IP address for connecting to the virtual desktop, and how long they expected the meeting to last. Knowing this information beforehand helped Berkeley and Concord to anticipate how the meeting would unfold and to better prepare for it. As a rule, if the teams anticipated the session to last longer than one hour, we agreed to take a break on the hour-mark, and generally kept that schedule.

As the teams grew more comfortable, the number of virtual pair programming sessions increased. During these sessions, the driver was often going too fast for the others to keep up. Sometimes, this was due to the network connection being slow. At other times, this was due to the challenge of communicating difficult concepts remotely. Thus, we made conscious efforts to ensure that everyone kept up by frequently pausing for questions and clearing up any misconceptions before continuing. In essence, the teams strove to maintain real-time two-way communication and to simulate the experience of working together. Such efforts allowed these virtual pair programming sessions to be successful and gave each participant the feeling of shared code ownership. We emerged with an equal pride in the outcome and responsibility for the code.

We found that bringing all participants into team discussions was important for building trust and working efficiently, but sometimes email and instant messengers were the wrong tools for the wrong audience. Email and instant messages were often composed quickly without too much regard for the readers. Messages were sometimes misunderstood and subsequent discussions became heated, eroding team trust and spirit. Often, a short video conference resolved the miscommunication.

When a new team member joined the group, it created uncertainties for everyone. How was this person going to catch up with the rest of the team and contribute to the project? How did this affect the team chemistry? These questions needed quick resolution before incurring any negative effects on the teams' morale. New members were welcomed during an introductory video conference which included orienting the person to our coding standards, programming practices, documentation, and testing expectations [4]. Newcomers would describe their personal skills and the rest of the team could adjust their expectations accordingly.

In general, everyone was mindful of etiquette as the members were from different cultures. Holidays were different from location to location and we learned to accept and respect that those teams were not to be disturbed during that time. We saw the collaboration as a chance to learn more about each other's culture and expand our horizons. With time differences, we did our best to arrange meetings that were reasonable for most teams. We did this by gathering everyone's availability for the following week and agreed on the next meeting time during our weekly meeting. Over time, the team members learned when other people were generally available. This went a long way to making other team members comfortable. Having comfortable team members allowed team trust to build naturally and led to a more enjoyable collaboration experience.

4. Software Architecture, Tools, and Design Approaches

One of the many challenges we faced was maximizing productivity while working separately yet collaboratively together. The Concord team was proficient with the Ruby on Rails (RoR) framework. The Toronto team was skilled in Java and J2EE. The Berkeley team, while proficient in Java, did not have the web development experience on the Java platform. Naturally, both Ruby and Java developers wanted to keep their preferred language and framework. This made sense from the customers' perspective as well, because it avoided costly training and proficiency ramp up time. We saw very little business value in compromising with just one programming language, at least in the short to medium term.

The Concord team had already begun some of their work on a remote data storage system which was re-purposed to serve all three portals. Concord had created several eXtensible Markup Language (XML) based Representational State Transfer (REST) [5] web services prior to Berkeley and Toronto starting their portals. Since REST using XML was a more natural style for RoR and functional requirements for that component were already being satisfied for all three teams, there was no need to go down a different path.

A wiki page was maintained by Concord to document the web services Application Programming Interface (API). As Toronto and Berkeley progressed with development, a comprehensive set of functional tests were created to verify and enforce the actual API against the written specification. These functional tests were part of our continuous integration test suite that provided quick feedback [6]. Whenever changes to the API were required by any team, a video conference with all the teams was called to discuss these changes and subsequently update the API and its tests. This worked extremely well for us as a lightweight and agile process to maintain a binding contract between the evolving systems.

From the outset, Toronto and Berkeley team members were faced with questions about how to develop the core code that would be shared by both projects, but could also be customized to suit each project's requirements. There were many long and heated debates about the co-development of a common library with separate codebases for each individual product versus a common codebase that could derive customized products. In the end, it was a single codebase that won the battle. In retrospect, it was absolutely the right choice. However, making the decision was the easy part. How would we implement this strategy?

The Toronto and Berkeley portals differed in many respects. These differences were more than just cosmetic branding. The Berkeley portal was targeted for researchers who were studying high school students, in particular science learning activities. The Toronto portal was aimed at a much broader audience of researchers. The core functionality of basic user management and supporting the launch of interactive curriculum was the same, but some of the tools and business logic were quite different.

The key to our success in achieving a single codebase was programming to interfaces and not to implementations [7]. Interfaces defined the interactions between components. Toronto and Berkeley had very different implementations of these interfaces which allowed them to have differing behavior. Even data models required interfaces because these models had different requirements. For example, a Student data model for Toronto only required first name, last name, user name, and password. Whereas, the Berkeley implementation required other additional attributes such as birthday.

By enforcing strict coding standards that required programming components with interfaces, we were able to leverage an Inversion of Control (IoC) container [8] such as the Spring Framework that helped us loosely couple our systems and easily configure the implementation that was appropriate to build each portal. The Toronto team had built the underlying foundation, which was the default behavior of the components, and ultimately the base product. Then Berkeley took advantage of polymorphism and had the option to extend or re-implement that particular component depending on the requirement differences.

We built the Spring configuration loading such that the Toronto defaults would always be available for all products. The Berkeley portal would then be able to override the defaults or add to the list their specific components. We also combined our environment customizations such that all developers were able to easily build and run any other portal by changing some personal parameters. Achieving this objective took considerable time. It took several iterations and many support meetings between development teams to resolve a variety of problems. In the end, this proved to be a very helpful process in which each team supported the other and all teams progressed simultaneously. Whenever the customers and developers negotiated cross-product requirements, we had the ability to combine our efforts and work on tasks together for one another's project. This enhanced our ability to address high priority items and changing requirements quickly, and to deliver our products independently and concurrently.

Another technical challenge we attempted to solve was the ability to satisfy individual developer's personal preferences. The Berkeley team was developing on Mac OS X while the Toronto team began development on Windows, and we had planned to deploy the system on Linux in the future. Also, one of the goals of the SAIL project was to develop an open source community and allow others to adopt and contribute to the portals. So while supporting all possible development environments was beyond our scope, we did have to support a sufficiently large number of environments that were in use by team members.

We had chosen Maven 2 as the development management tool and Ant as an auxiliary utility for the Java platform. Both Toronto and Berkeley were using Eclipse as their IDE and some individuals had plug-ins for Maven and Ant while others used the command-line exclusively. Ant became the "Swiss Army Knife" that provided us with the ability to run virtually any command on the three operating systems. We had a policy that no commands were ever executed manually. Everything had to be repeatable and reproducible through Ant or Maven, even one-off tasks. By enforcing this policy, we were able to support both command-line users and Eclipse plug-in users. Ultimately, we were able to set up development environments for newcomers very quickly with this high degree of automation. This may be one of the reasons why our portal had successful early adopters from an outside community[2].

Concord's RoR data service continues to serve all three sites. This service may eventually employ JRuby (an embedded Ruby interpreter that runs in the Java Virtual Machine (JVM) written in Java) in hopes of bundling a self contained deployment of the portal and the data storage system. Those will be stories that hold the highest value for Toronto's customer and it will be up to that team to convince the other collaborators to help with that work.

[2] The SAIL Portal was adapted successfully by a European group known then as CIEL/SCY for a project called Co-Lab in May of 2007.

5. Conclusion

Having distributed collaborative teams presented many challenges including the lack of physical presence which slowed down or hindered processes. One of the benefits of agile development is to respond quickly to changes and feedback. Our group which consisted of development teams from three separate organizations adapted agile principles to various degrees to achieve this goal. Everyone made conscious efforts to find ways to improve communication. We were sympathetic and responded to questions and requests for help in spite of our own busy schedules. We built strong working relationships prior to most of us meeting in person. Lastly, we dedicated time, effort, and resources from the beginning of the collaboration to ensure that the software architecture was suitable for all teams and we were able to support each other along the way. Distributed agile development was effective for our group. We delivered timely software releases incrementally in response to our customers' needs.

6. References

[1] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, Rebecca E. Grinter, "An Empirical Study of Global Software Development: Distance and Speed", *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, July 2001, pp. 81-90.

[2] John Child, "Trust - The Fundamental Bond in Global Collaboration", *Organizational Dynamics*, Volume 29, Number 4, Elsevier, 2001, pp. 274-288.

[3] Julia Kotlarsky, Ilan Oshri, "Social Ties, Knowledge Sharing and Successful Collaboration in Globally Distributed System Development Projects", *European Journal of Information Systems*, 2005, pp. 37-48.

[4] Ilan Oshri, Julia Kotlarsky, Leslie P. Willcocks, "Global Software Development: Exploring Socialization and Face-to-Face Meetings in Distributed Strategic Projects", *Journal of Strategic Information Systems*, Elsevier, 2007, pp. 25-49.

[5] Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures", *Dissertation, Doctor of Philosophy in Information and Computer Science*, University of California, Irvine, 2000.

[6] Kent Beck, Cynthia Andres, *Extreme Programming Explained: Embrace Change, 2nd Edition*, Addison-Wesley Professional, 2005.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[8] Rod Johnson, Juergen Hoeller, *Expert One-on-One J2EE Development Without EJB*, Wrox Press, 2004.