Offshore Agile Maintenance

Naresh Jain
ThoughtWorks
njain@thoughtworks.com

Abstract

Maintaining a business critical application in production having serious performance and scalability issues can be quite a challenging task in itself. Having a new development team sitting on the other side of the globe can easily complicate things further. This experience report describes the challenges faced in this environment, lessons learned, and how we won the client’s trust and delivered business value as quickly and consistently as possible.

1. Introduction

Predominantly the Agile literature covers applying its values and practices to a green field project using a co-located team. However, offshore/distributed development is the reality of the software industry. Sometimes, maintaining and enhancing business critical applications can be the key to an organization’s success.

The existing literature[^1][^2][^3] states the following as challenges with distributed/offshore team:

1. Decrease in communication bandwidth
2. Lack of visibility into project status
3. Configuration management
4. Art of Command and Control structure
5. Cultural difference

Based on my experience working as a lead developer and a part-time Iteration Manager on an offshore agile team maintaining and enhancing a business critical application for a New York based client from Bangalore, I think the above points are by-products of something more important. The root causes according to me are the following:

1. Lack of trust
2. Loss of context, both business and technical
3. Delay in feedback cycle due to increase in distance and time difference
4. Abstraction of business and technical problems and decision

In this experience report I would like to explain how we addressed the above issues and turned the project into a success story. I would also like to present how some standard XP practices helped us maintain this offshore project better.

2. Background

The application was developed by ThoughtWorks US in 2001-2003 for a pay-per-view cable company. This was an EAI project for back office data validation and billing system. It also had components of the J2EE stack for presentation. During the initial development of the project some XP practices were used on the project. After this phase, ThoughtWorks maintained the application for a few months and then the client decided to maintain the application themselves. A year later, the client came back to ThoughtWorks asking us to maintain the application. But they did not have the budget for a two or three member team working from the US. Hence it was decided to offshore the project to India. We had a new team of three developers, a QA and an Iteration Manager in Bangalore. We also had an Account Manager in the US. The initial team spent 3 months in the US for knowledge transfer and then started working from India. From the client side we had three people, one QA, one DBA and a PM, all in New York. This was a one year maintenance contract.

3. Offshore Agile Maintenance project

I would like to clarify certain terms before we jump into the project details:

Client: The product company which provides services to the actual end users. They are mostly technical and business people who maintain and plan the software product. They are the clients to ThoughtWorks.

Stories: Bug reports or new feature requests from the client or internal QA. For the purpose of this paper, I
would consider bugs and features as just stories and they could be interchangeably used.

3.1. Practices used on the project

During the maintenance phase, the team applied most of the XP practices. Following are the list of practices used:

1. Planning game – 2 week iterations, story cards, Iteration Planning Meetings
2. Small releases – 2 to 3 months
3. Refactoring
4. Pair Programming
5. Collective code ownership
6. Continuous integration/Automated Release
7. Test Driven Development [40-50% of the time]
8. 40 hour week / sustainable pace
9. Standup meetings
10. Coding standards

What we did not have or could not do were:

1. Onsite Client
2. Metaphor
3. Simple Design
4. 100% Automated testing

4. My observations on the project

4.1. Lack of trust was the most important challenge. Working with a team, sitting on the other side of the globe, with whom you have never worked before, leads to this challenge. If the client and the development team can trust each other, communication and visibility will not be such a big issue. Also, having a trusted, self-organized team can eliminate the need for command and control structure. Building trust takes time, but should be the highest priority for the team. Improving visibility & communication and constantly delivering working software is the key to win the trust.

4.2. Loss of context is the second most important challenge. People think waterfall is the best model for distributed maintenance teams. This is based on the classical misconception that staging-out the development process will yield better control and visibility. However, staging leads to loss of context. If one does not understand what business problem they are really trying to solve, it can be very difficult to deliver the right solution. Most often in a distributed project, all the decisions are taken onsite and the offshore teams are ordered to implement the solution given to them. Implementing the solution without the technical and business context is a big red flag. This could lead to over engineering or under engineering based on time frames available to the team.

4.3. Abstraction of business and technical problems and decision, compounded with loss of context gave a very misleading picture on both sides.

4.4. Not considering offshore teams as equal stakeholders in the project is a recipe for disaster. It is difficult to trust outsiders. Let it be consultants, contractors or offshore teams. There is a feeling that outsiders do not take the same ownership and responsibility as the onsite team. This leads to one of the biggest misconception that offshore team members only care about their pay checks. Based on my experience I can tell, they care, if not more, at least as much as the onsite team about successful delivery of a quality software solution.

4.5. Traditionally, people try to address the communication and visibility problems by having one point of contact and building a reporting structure or a tower of hierarchy. Communication and visibility are not ends in themselves. They are mainly required for getting timely feedback. With experience on agile teams, it is clear that you need a team of motivated professionals who all communicate, all the time and give the right feedback to make appropriate adjustments.

4.6. Change is inevitable. This project suffered from frequent changes on client team. We had to deal with new team members on the client side. So the challenge was not just building trust, but was to constantly build trust with different people and handle political situations which arise due to job insecurities.

5. Things that made a difference

5.1. General practices:

5.1.1. Empowered small teams: A small teams of motivated individuals playing different roles made the team very effective and helped us achieve significant tasks. Having a self-organized team talking to the client directly gave all the team members a better perspective of the project. Adding more people to the project complicated the whole team dynamics and lead to a lot of communication problems. With a small team it was easy to have a shared understanding and eliminate a lot of waste.
5.1.2. Freedom to try new things: This helps the team innovate better practices and techniques. Hard problems on hand and lack of time are the situation on most of the projects. But lack of freedom to try new ideas can impact innovation and motivation.

5.1.3. Lots of outings, parties, ice-creams, food, etc. Life is worth enjoying. This helped us address the stressful nature of fluctuating workload.

5.1.4. Acceptance testing with FIT gave us an unambiguous automated tool to communicate with the client. This helped us understand the requirements better and to get better feedback on task completion.

5.2. To address lack of trust:

5.2.1. At times the team went out of their ways to deliver business value to the client. This helped them realize that the offshore team cares and respects their business. Once we convinced the client about this, it was easy to win their trust and get the equal stakeholder status. Everyone on the team was passionate about the project, because they felt they had a stake/say on the project. We also gave a lot of freebies initially to build a healthy relationship with the client.

5.2.2. Building a personal rapport with the client helped us understand them better. It helped us to build mutual respect for very team member, which is an important ingredient of any successful team. Also with the fluctuations on the client side, it became very important for the offsite team to relate to the new team members faster.

5.3. To address delayed feedback cycles:

5.3.1. Retrospectives: It created a great platform for the team to express themselves and to think constructively to improve the project and to adapt. Involving the client in retrospectives gave them greater visibility. We used retrospectives to control the direction of our project. It helped us to identify issues and brainstorm about them. Retrospective was one of the most influential practices which helped the team succeed.

5.3.2. Seamless collaboration between Developers and QA: Dev-QA pairing on acceptance tests and other talks helped in driving our development through tests. QA smoke tests on developer’s machine helped in faster feedback.

5.3.3. 100% Automation: Anything that had to be done more than once in a day was automated. This gave real agility to the team to move at the pace the business wanted us to.

5.3.4. Refactoring fests to mercilessly refactor the legacy code base. Not improving the design of the system constantly can seriously impact the ability to deliver business value.

5.3.5. Realistic measures of progress: Constantly monitoring the unit, functional and regression test count gave the team unbiased and unambiguous visibility into the progress of the team.

5.4. To address loss of context and abstraction of business/technical problems and decisions:

5.4.1. Pairing and SCube sessions: Pairing and SCube [Seek, Speak and Share] session helped us to spread domain knowledge much faster. We had the SCube sessions during lunch hours where the team members discussed either technical or business aspect of the project. These sessions also helped kick start new team members.

5.4.2. We involved the client into the development process on a daily basis through daily status mails, project wiki in client environment, instant messenger conversations, iteration plan meeting and lots of informal communication. Team members used take ownership of these tasks in turns. This helped everyone on the team feel special. Client-driven demos on the developer’s machine every day was a great way to win client’s trust and get feedback.

6. Our Approach

6.1. Structure and execution of the project

Initially we started with 2 week iterations and bimonthly releases. After our release the client had a QA team test the software for a couple of weeks on their QA environment. If no major issues were found, the build was promoted to their on-boarding environment. This environment was very similar to the production environment. Most of the performance and stress testing was done in this environment. If everything went smooth, the build was promoted into production. The total turn around time for production move was roughly three months.

6.1.1. Planning

A release would begin with a release planning meeting. The meeting included: the client side project manager, account manager from ThoughtWorks and from offshore - the Iteration manager and a lead
developer. Here they decided what needed to be delivered in the next release. Since our releases comprised of bug fixes and few feature enhancements, the client prioritized based on the impact on business.

Biweekly iteration began with an iteration planning meeting. This meeting involved the whole team. The developers and QA estimated and the client prioritized or shuffled the stories. The meeting was usually done on a Friday evening offshore time and Friday morning US time. This helped the team to start working on the new iteration on Monday morning.

6.1.2. During the iteration
The local QA started testing and reproducing the bugs slotted for the next release. The developers picked a story from the story board. If it was a bug, the developers would pair with the QA to see them reproduce the bug on the local QA environment. If it was a new feature or an enhancement, a developers and QA worked with the client side person to come up with an acceptance test. Once this was in place, the actual development or bug fix began. After the developers felt they are done, they paired with the QA to do some basic smoke test on the developer station. This really helped to get some initial feedback. If everything looked fine, a new build was deployed on the QA environment for regression and exploratory testing.

Based on the complexity of the story, the client had a conference call in the evening with the team. They used VNC to connect to the developer or QA environment and drove the demos. During this conference call other issues or questions were also answered. Following the conference call a summary email or a status mail was sent out to the entire team.

6.1.3. End of iteration
The development would freeze on the second Wednesday of the iteration, leaving Thursday for regression testing. Friday morning the developers created the actual release build, which was then installed on a clean QA environment and tested. Towards the end of Friday the actual release is FTPed to the client server. After lunch on Friday the team had their retrospective followed by the Iteration Planning Meeting.

6.1.4. Release process
The client side QA would install the new release in their QA environment. The creation and installation of the release was fully automated. The client side QA tested the software and raised any bug, if found, in the common bug tracking system. The local QA and the developers then looked at those issues. If they were minor bug fixes, they were fixed immediately, else pushed to the next iteration.

6.1.5. Big win!
Towards the latter part of the project we also tried releases which were not time boxed. A set of bugs and/or features were selected. The offshore team would fix or develop them, test them locally and then release it to the client. The client would then test it in their different environments and deploy it to production. The turn around time could be anything between 2 days to a month. This is perhaps where the real agility in the development process comes into play. We were able to reduce the time spent on planning and estimation. Since we had won the client’s trust, they were in agreement with this process.

6.2. Life cycle of a bug fix or a change
On this project we did not have a change control board or any thing on those lines. We believe the client is the best person to judge the importance and priority of the bugs and new features for their business. We, as the development team could provide the technical knowledge to help them plan. We always strived towards customer collaboration over contract negotiation and responding to change over following a plan. Appreciating the importance of the last two points helped us to embrace change for the client’s competitive advantage.

Following are the stages a bug went through before it is fixed:

6.2.1. Bug report: Any member of the team, developer or QA could file a bug in the common bug tracking system.

6.2.2. Bug validation: The Client side QA and local QA validated the bug in the light of the business domain. As a part of their validation, they tried to reproduce the bug and analyzed if there were other scenarios which could cause similar bugs. At this stage the bug or feature was associated with priority and severity from the QA point of view.

6.2.3. Bug prioritization: Once the bug was reported the client side project manager prioritized it based on the business needs. The priority would be discussed in the daily status calls.

6.2.4. Bug reproduction phase: Once the bug is planned for a particular iteration, the QA tried to reproduce the bug on the local environment following the exact steps mentioned in the report before the planning meeting. Pairing up developers with the QA towards the end of this phase helped the team in tasking and estimating the bugs better.

6.2.5. Bug planning and estimation: During the planning meeting, the development team
broke down the bug into tasks and estimated it. Based on the estimates and severity the bug/feature was prioritized. It was not easy to estimate a bug upfront. We did some investigation before planning and used our past experience in dealing with similar bugs.

6.2.6. Acceptance tests: The QA and the developers paired to write acceptance tests in Fit. These tests were sent to the client and were discussed during the next conference call.

6.2.7. Fix or development phase: The developers tried to reproduce and analyze the bug on their machines. Pair programming really helped to quickly identify the bug. Throwing more eye balls at a problem made the problem look shallow. Once the developers had analyzed the root cause of the bug, they wrote a failing test. It could be a unit test or it could be a functional test in case of work flow related issues. In either case bug or new feature, once the developers had a failing test, they made code changes till all the tests passed.

6.2.8. Smoke testing on developer’s machine: Once the developers claimed to have fixed the bug or developed the new feature, the QA quickly validated the same on developer’s machine. This helped in quicker feedback cycles. The smoke test was usually a 5 to 10 minute task.

6.2.9. New QA build: After successful smoke testing, the developers checked-in their changes. On successful CruiseControl build, the QA installed the latest build in their environment and start regression testing.

6.2.10. Complete regression test: Towards the end of the iteration, a complete set of manual regression tests were executed to make sure everything was fine. All the automated tests would run with every CruiseControl build cycle. Though we strived to automate all the tests as much as possible, there was still considerable manual testing left.

6.2.11. Release test: Once the release build was created the QA smoke tested the build to make sure all the bugs and features were addressed. Anything which was not resolved was documented as known issues and sent to the client.

6.2.12. Client side testing: The client side QA team installed the new release in their environment and ran their tests to certify the release. Any issues found during this phase went through the complete bug fix cycle. The client side QA team had their own set of tests and test data.

6.3. How did we manage with lack of tests and documentation?

Most often when we talk about maintenance projects, the first thing that comes to people’s mind is documentation. On Agile projects though, the first thing that comes to our minds is tests. On this project we did not have either. The way we addressed this problem was:

6.3.1. We wrote unit and functional tests every time we modified any portion of the existing system. Tests gave us a great way to understand what the code was doing and be sure about it. At times we could understand what the code is doing, but since we did not have the bigger picture, we were not too sure whether it was doing the right thing. This was when we discussed it with the clients about the functionality using the functional tests.

6.3.2. While the developers build their suite of automated unit tests, the QA started with manual tests because it had quick turn around time. Slowly over a period of time, the QA started building some automated tests for regression. It took us a long time to convince the client. Automation was very challenging and time consuming.

6.3.3. We always start changing code with a failing test. That gave us the confidence that the work was done correctly. Also having the QA constantly regression test the system gave us a lot of confidence. This helped us build our test harness.

6.3.4. We used the artifacts and tools that the development team used during the initial development. For example, at times we looked at the logs and tried to understand the business logic and its flow. This helped us in debugging and understanding the system better. Since this project was an EAI project, there were lots of asynchronous parts to the system, which made it difficult to hook a debugger and step through the code.

6.3.5. Constant refactoring of the system. Having built a small test harness around the code, we had more courage to change the code to fix issues. These tests also gave us the ability to make the code more understandable from our view point. We had inherited a lot of JavaScript code which was difficult to understand and test. We slowly moved all the code to Java and wrote extensive unit tests around it. This gave us the ability to simplify our design to a great extent. Refactoring to patterns
helped us understand and communicate the existing design better.

6.4. Automation

Automation is a big part of any project, especially a maintenance project. We spent a lot of time automating things on this project. Following is a list of things that we automated:

6.4.1. Installation process. Initially it used to take 3 days to manually install everything and get started with the project. We automated it and brought it down to 30 minutes. It helped us have the same configuration on all the project installations.

6.4.2. Release process: The creation and installation of a new build was completely automated. It helped us significantly reduce packing related issues.

6.4.3. Configuration management: Configuration differences seemed to be a big issue on this project. A lot of bugs were due to wrong configuration of different components on the projects. We used ANT and properties file to automate this.

6.4.4. Database change management: This project suffered from poor performance. It was identified that the database was a huge bottleneck. So we automated changes to the database, which gave us control on what was being changed. Preventing ad-hoc changes to the database and going through the standard scripts made the database more stable. Once we had enough scripts for our database, it was easy to look at all the scripts and identify proper indexes for tables and further fine tune the database.

6.4.5. Functional tests: We started using Fit as our acceptance test framework. We also used it for regression testing. Automating tests helped us in quicker feedback and saved a lot of time and effort in identifying issues due to code or configuration changes. We ran our functional tests with every CruiseControl build.

7. Results

7.1. Our team resolved the visibility and communication problems by constantly communicating with the client and giving them timely feedback. Correct feedback helped the offsite team to build the trust in the client. Good technical skills and interpersonal relation were also very important.

7.2. We were able to build a self-organized team of highly motivated individuals communicating constantly with the client to break the distance, time and culture barriers.

7.3. The team adapted very well to fluctuations on the client side. We were able to make our way through the political noise to actually deliver business value by making the system better everyday.

7.4. Good version control system, continuous integration servers, build scripts, automation skills really helped us resolve configuration issues. There are plenty of tools available to make this task easy, but it was important to build the right mind set and attitude in people to do it.

7.5. The team had not just maintained a business critical application which delivered great business value, but had also built a relationship which the client respected and valued. The client was ready to renew the contract and pay significantly higher rates to retain the team.

8. Improvements on my future projects

8.1. Heavy focus on automated tests and refactoring. This helps in keeping the code base live.

8.2. Exchange programs where off shore team members can rotate with client side team members. This helps to build trust and improve visibility. It’s a great way to solve communication issues. Being onsite helps one to understand the business needs better.

8.3. Good roll-off plans for the team members. Since most maintenance projects take significant time for a team member to be really productive, there is a tendency to retain them on the project for a long period of time. Having clear and realistic roll-off plans help set clear expectations for all parties, and helps the team members stay motivated.

8.4. Scratch your personal itch day. Every team member needs a breather once a month to work on things which could improve productivity or help the project in some way. Every team member seems to be itched differently by different things on the project. Having a day to resolve some of those issues can help the project and can also be a great motivation for team members.

9. Conclusion
After maintaining the application for 15 months, I think all the agile values and principles can be very helpful on any offshore project. By nature, offshore/distributed development is very difficult. The need for tighter feedback cycle is even more important on offshore projects.

The emphasis on automation, testing and refactoring is a key success factor for any maintenance project. For any maintenance project it is important to make the system better everyday. Agile gives the right mind set to people to do this. In my experience most of the Agile practices, XP in particular, can be very useful for an offshore maintenance project.

Agile cannot solve all those problems, but it can certainly make it much evident and easier to fix them. The bottom-line is Agile can help, but it’s the people who make the difference. Agile gives a lot more power into the hands of the team, if they have the right attitude, it can do wonders.

10. Acknowledgements

I thank my team for their cooperation and support on the project and various people in ThoughtWorks for their advice on this paper. I’m also thankful to Monica Yap for providing valuable guidance and encouragement in shepherding this paper.

11. References

[1] Distributed Agile Development and the Death of Distance

[2] Case Study: Distributed Agile Development
http://www.pivolis.com/pdf/Distributed_Agile_V1.0.pdf

[3] Distributed Agile
http://www.agilealliance.com/articles/steindlchristophdistr/file

[4] Using an Agile Software Process with Offshore Development
http://www.martinfowler.com/articles/agileOffshore.html

[6] Follow the Sun: Distributed Extreme Programming Development
http://doi.ieeecomputersociety.org/10.1109/ADC.2005.26