XP On A Large Project – A Developer's View

Amr Elssamadisy
ThoughtWorks, Inc.
651 West Washington Blvd.
Suite 600
Chicago, IL 60661 USA
+1 312 373 8523
Amr.Elssamadisy@thoughtworks.com

ABSTRACT
At ThoughtWorks we have adopted a modified version of XP that has been tailored through our experiences to suit a large project of over 35 developers and 15 analysts. This project, a leasing application which we internally called ATLAS, originally started 3 years ago with the standard analysis and design front-loading of traditional development projects. This paper is written from a developer's point of view, the experiences and the techniques that were tried and either became habit because they were useful, or never quite caught on. We will take the different practices that are encouraged by XP in Kent Beck's *Extreme Programming Explained* [1], and give our feedback on each practice. Then, in summarizing, we will give a recommendation of what changes must be made to the XP process to be able to utilize this methodology and still produce quality code at a fast pace for a large project.

Keywords
Large project XP

1 INTRODUCTION

At ThoughtWorks we have been working with XP on a large project for over 15 months. This project originally started out 3 years ago with a huge requirements document and several functional independent sub-teams. Starting in Jan. of 2000 we decided to drastically change direction with XP knowing that XP is NOT for large projects. We needed to start delivering functionality to our client and give them (and us) confidence by delivering a working subset of the project and not just a prototype (we have over 25 developers on the team and about 15 analysts). This is not to say that we did not have functionality ready, but each team had its own scaffolded piece of code that it could run, but there was no complete application that could be run. To make the long story short - the client needed to see something and we had yet to show them some real functionality. Iteration 1 was a great success because we delivered to the client a working subset of the application that made sense in the real world. We built on this initial success - with developers signing up for different tasks every time so that our Asset team and AR team and GUI team started to meld together. There were definitely growing pains - some egos clashing between analysts and developers but we were able to get over these hurdles.

Everything was rosy – we were delivering functionality at blazing speeds - but as time wore on and the project kept going - there were some things that weren't working as expected. This is what this paper will be about - what we REALLY learned after the honeymoon. How we ACTUALLY maintained a fifty-man project over a year and a half through development and successful (and sometimes not so successful) iteration releases. The pains we went through, the work we did and are continuing to do, and finally what we have learned and what we will do differently on our next projects.

We will first present our initial starting setup – how we applied XP fifteen months ago, and then present our current application of XP. That said, the rest of the paper will discuss the reasons we changed our technique changed. That is, we will analyze the natural selection process that killed off many of our bad ideas or molded them into more effective ideas that help in running a large project. We will then summarize and give a discreet recommendation of how to proceed with XP in a large project.

2 ELEMENTS OF XP WITH A LARGE PROJECT

Ok, now we get to the good stuff. The team of developers and analysts consists of approximately 35 developers, 15 business analysts, and about 10 QA. The developers rely on the analysts to be the customers of the project. There is a real customer however, and the analysts work with them to collectively make customer decisions. The table below shows the elements of XP as discussed in [1] and gives a brief description of how that aspect was being used at different stages in the project. We will use this table to analyze our team's natural selection of practices and how textbook XP evolved on a large project to support a fifty-person team that has taken a project to completion.

	Planning	Release Cycle	Metaphor	Design Simplicity	Testing	Refactoring
1/2000	Large iteration planning meetings. Full team of developers and analysts met for an entire day to discuss new story cards and estimate. Most developers sign up for new functionality.	1 month	None	Transitioning with an existing code base. Existing code still complex, but new code as simple as possible. This phase included throwing out and rewriting existing functionality that 'someday might be used'.	Unit tests started with some of the new code. Push being made to have a large test base. QA does all functional tests and has the authority to pass/fail a story card.	Refactoring done as needed when touching old code.
7/2000	Same as 1/2000 – but definitely feels very inefficient. Most attendees not concentrating or participating. Long drawn-out discussions. 50-man meeting is overwhelming.	1 month	None	Designs continuing to be as simple as possible. Refactorings of existing designs being done. Code reviews being done to collectively discuss new designs so that the whole team is familiar with coding/design trends.	Better unit test coverage but still not full coverage. Coded Functional Tests (CFTs) to help test coverage.	Most developers are heads-down in delivering functionality, very little refactoring done. Code base getting worse at end-of-iteration crunch to deliver cards to QA
1/2001	Try making iteration planning meetings faster – by doing more prep work with small groups of developers before the actual meetings.	2 weeks	None	Most of the designs built on existing designs – standards kept, code reviews tapered off since new types of designs/refactorings not being done.	Attempted to get rid oif CFTs and replace with a screen-scraper and failed - went back to CFTs. New unit tests being added but coverage still lacking. QA starts to automate functional tests with screen scraper.	Refactorings being done much more often as code starts to spaghetti in some parts of the app. Major clean-ups being done because implemented simply and then iteration deadlines caused to code to grow without the important refactorings being done.
6/2001	Several meetings for cards/sets of related cards between developers interested in or knowledgeable about the functionality and analysts who are responsible for the story card.	2 weeks	None	Large part of the team and all of QA working on delivering version 1.0 to the customer. Code base split and new functionality being added without QA. More reliance on unit tests.	Test coverage seems to have stabled at a level that is lacking. QA has been out of the loop with new functionality because focused entirely on version 1.0 delivery.	Refactorings on the release are minimal, while on the continuing product branch developers are refactoring more conscientiously after being forced to do larger more painful refactorings earlier in the year.

	Pair programming	Collective Ownership	Continuous Integration	40 hour week	Onsite Customer	Coding Standards
1/2000	Since we had made a decision with XP, the entire team read [1] and were encouraged to pair-program. Everyone tried it and it caught on with most of the team.	In the initial phase we were moving from function-oriented groups, so we did not feel collective owner-ship, but also did not feel protective of our code.	Online from iteration 1. See [2].	This was a proof of concept work – and we wanted the client on-board. So a lot of extra hours were put in to meet the deadline	Business analysts are the on-site customer. 15 analysts on the team. The actual client was offsite and the analysts communicated with them.	None other than normal Java syntax.
7/2000	Pair programming still strong, developers are pair programming for new functionality, but bugs and maintenance have developers working solo. Some programmers stop to pair program all together.	Ownership of code is completely diluted as more developers touch different pieces of the app. Very good communication takes place via informal chats, code-reviews, and short Stand-up meetings.	CFTs also added to the build process	Cyclic hours hitting 50 and 60 hour weeks at the end of each iteration to try to pass the story card.	Same as above	Bi-weekly code reviews give developers a chance to discuss ways that different subsystems are being implemented where we agree on informal ways of coding similar parts of the system.
1/2001	Pair programming definitely less as more of the coding is straight-forward or because a developer is refactoring a piece of code.	Stand up meetings are dropped as being inefficient, but code owner-ship remains diluted. Developers start specializing in parts of the system again.	Stable – same as above.	With 2 week iterations, developer estimates become more accurate and closer to 40 hour week..	Same as above	Code reviews tapered off, saturation of design/coding standards.
6/2001	Definite trend with almost all new functionality being pair-programmed and almost all debugging/maintenance done by a single developer.	With specialization, there is a trend with a small set of developers knowing more about different parts of the code – so we tend to have them be more active in the continuing designs, but still communal ownership of the code.	Stable – same as above.	Same as above.	Same as above.	Same as above.

After examining these tables we notice that almost all the practices have evolved over 18 months. Some have stabilized – since we see many 'same as above' in columns like continuous integration and testing.

3 PAIR PROGRAMMING

First of all, lets discuss our flavor of pair programming. We mostly have 2 developers working on the same story card for an entire iteration (or sometimes several iterations on related cards). With a large project, developers need more focus – since start-up time on a new area of the code is not negligible. Good communication between developers and iteration planning meetings keep everyone 'in the big picture' about who is doing what. This allows the classic textbook pair programming where developer A goes to developer B and ask him to pair to solve a problem he is working on.

Pair programming is good - but not realistic all the time. The most common reason a developer does not pair program is if he is working on bugs or maintenance. In this case we have found there is little added value having a set of eyes debug code. Also, there are many tasks which are 'just like the task we did last iteration.' In that case, since the solution has already been found (usually by a pair of developers), there is also no added value to pairing up.

Finally, developers have different personalities - some people just need a break from pair programming. Some are really just more talented than others and are slowed down by it - and it becomes obvious that it becomes a burden for these people.

4 UNIT TESTS AND INTEGRATED BUILDS

Unit tests and integrated builds - are ABSOLUTLY MANDITORY - we would be stopped in our tracks and not able to deliver one piece of code if we could not rely on tests. As the application gets larger and larger it becomes almost impossible to add new code or refactor existing code without going through tests. We currently have an integrated build [2] where a new build and tests are run when new code is checked into our repository. The details of every build, tests broken, and people responsible are immediately available on an internal web page that developers can access to see the current state of the build and business analysts and QA can access to retrieve the latest build to test the functionality they are working with.

5 GROUP OWNERSHIP AND INFORMATION SHARING

Dissemination of information through communication and rotating through different parts of code is important to keep such a large project from fragmenting into several independent pieces that make inaccurate assumptions about the system as a whole[1]. Communication is a must - but there is no way you can really force a quiet person to talk - so we tried in our bi-weekly stand up meetings for everyone to say something so that the quiet people say what they need. In the end we dropped the meetings because most developers felt it was a waste of time and were talking informally about all issues. This is one of this team's strongest point – it almost as if we are one large communal developer – there is a definite synergy akin to that in pair programming when an entire team of developers communicate well.

As for the rotation and doing a little of everything - we started out that way - but in the end, when you have deadlines - we find ourselves signing up for things we already know. The inherent start up time to come up to speed on a complex piece of code is too much of a time sink. In the end moderation is best - during crunch time you do what you know or are familiar with, at other times - when you are doing bugs - you can explore - or sign up for one task you know and one that you don't. So what is now the norm is that a developer will sign up consistently (for a few iterations) for related cards and then move on to another part of the system. Signing up for cards in several parts of the system in one iteration is definitely out of fashion these days.

Code is definitely worse than we started. But is this because the project is larger? Or is it because many people touch the code are first timers[2]? A little of both - but at the same time we don't get islands of code that do not have anything to do with the rest of the application. There needs to be a constant cleaning up of code. Which brings us to the next point of refactoring.

6 REFACTORING

Refactoring is a definite need on large projects using XP to make up for the inconsistent code. Even with people who are familiar with the most of the app – or large parts of it – some refactorings just take too long and are always getting pushed back. Time out needs to be allotted for refactoring – this is something the project managers have to realize – and in our case they did. We were able to take the time needed to refactor major parts of our code..

7 SHORT ITERATIONS

Iterations and deadlines are mandatory – but the length has always been an issue. We originally had longer iterations – one-months iterations– and this caused an end of the month squeeze and bad code being checked in because we had such large cards – and inevitably we had problems estimating. We had to learn to accept cards not making it

---

[1] It is accepted in developer circles that it is a GOOD thing for modules to be completely independent and only assume the interface of other modules. This is true of PROGRAMMING, but not true of developers. Developers need to be interdependent – to know what is going on in the rest of the app, so that the independent modules they write have a consistent business foundation.

[2] Not first-time developers, but new comer's to the area of functionality.

(although it is still hard on developers – this one included – to miss a deadline). We are now doing two-week iterations that have made our estimations closer to target and we are also allowing cards to fall through. On the other hand, the two weeks seem to be over really fast – it is harder to take on a large refactoring at the beginning of an iteration so a new piece of functionality can be added. So which should it be 1-month or 2-week iterations or something completely different? This is agile development right? Then we should be agile with these rules – allow some issues to cross iterations. Lately we have had some issues that we knew could not be effectively split into iteration deliverables and allowed up to 5 two-week iterations for completion. To offset this large period we continued to regularly revisit the progress at the end of every iteration.

8 SUMMARY

So – after 18 months on a 50 man team what are our recommendations and lessons learned? Let's list them:

1. Have an iteration planning meeting at the beginning of each iteration where the customer and developers split up in groups all day to discuss the latest story cards and estimate them. At the end of the day regroup and present your estimations and findings and then have developer signup. This will keep the whole team in-the-know about what is happening without burdening everyone with an extremely tedious and long meeting.
2. Keep releases as small as possible – 2 weeks works for us, but at the same time be flexible when larger pieces need to be done over several iterations. Allow signup for a multiple iteration card but always review progress every iteration.
3. Write as many unit tests as you can – that is self-evident. You should also have an automated suite of functional tests to keep the test coverage at an acceptable rate. A QA team cannot be replaced – no matter how many tests developers write – we are flawed in our biased understanding of how the system works or should work.
4. Simple designs have helped us release a working product to the customer consistently. Frequent design meetings (lunch is the best way to gather the development team) are very helpful during stages of intense new functionality being added. This will keep from having parallel and maybe incompatible solutions to be implemented in different parts of the application.
5. Refactoring is the only way to be able to have simple designs as stated in (5). Refactoring of designs is just as important as refactoring of the code. It will always be tempting not to refactor and to just patch a solution, but if it is patched too much the team will be forced to make major refactorings later on.
6. Pair programming should be religiously followed when new functionality is added, and should be skipped when fixing bugs or doing repetitive tasks that have already been 'solved' before by a pair of developers.
7. Collective ownership goes hand in hand with communication. The team must figure out a way to communicate effectively. It may be no more than just informal discussions which worked best for our team, otherwise regular stand-up meetings that last for 10-15 minutes are a good way to disseminate information.
8. With a large project a group of individuals are needed to be the customer – to generate enough work for the large number of developers. This is strictly dependant on the where the business knowledge is.
9. Coding standards have been very informal and this has not been detrimental to our progress. What is more important is communication of ongoing work through presentations. Code is not enough of documentation, developers need to see the big picture also – and that cannot be relayed through code.

We also found that the following things didn't work for us:

1. Bi-weekly stand-up meetings were not efficient. Opted for informal communication with once-a-month iteration team meetings.
2. Full team meetings during the iteration planning meeting did not work. A day of small group meetings with 30-45 minute review at the end of the day of the cards worked better.
3. 1 month long iterations were too long and were detrimental to code quality. Moved to 2-week iterations which is easier to track and makes estimations more accurate.
4. (3) does not work all the time for larger chunks of the code – especially if refactoring a large part of the system. So exceptions are made where one card may span several iterations.
5. Metaphors are unrealistic with large projects. They are just too complex. Period.
6. A 40 hour week has never been an issue for us. 40 hours is the minimum and we have not been adversely affected by working more than 40 hours, but then again, we are not pair programming 100% of the time.

That's it. XP, or our evolved version of it, has done wonders for us as a team. We are in the process of delivering a very large and complex application on time and have built some very serious experience as developers that will enable us to tackle just about any project that comes along.

REFERENCES

1. Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999; ISBN 201-61641-6
2. Fowler, M. and Foemmel, M.; Continuous Integration; http://www.martinfowler.com/articles/continuousIntegration.html