The 5 reasons XP can’t scale and what to do about them

Ron Crocker
Motorola, Inc.
1501 W. Shure Drive
Arlington Heights, IL 60004 USA
Ron.Crocker@Motorola.com

ABSTRACT
XP seems to be a good method for small teams to develop high-power software system. There are concerns about the ability of the method to scale to scopes any larger than 12 developers. It only takes a small numb er of changes in the set of practices comprising XP to make it into a method scaleable to quite large scope. Indeed, modifying 5 practices and adding in a few new ones resolves the issues.

Keywords
scaling

1 INTRODUCTION
I have this belief about XP (and perhaps other “agile” development processes) about why they can't scale up, It's similar to why I had problems with heavyweight development process (think CMM) scaling down. In both cases, the approaches focus on a set of problems that exist in the target environment (small-scale development for XP) that don't exist in the other environment and miss important problems that don't exist in the intended environment but do exist in the other environment. In either case, you can't take the process and simply add more people (or in the case of CMM, take people away) and expect it to work.

Hypothesis*: The current set of twelve practices commonly referred to as eXtreme Programming (XP), as defined in Beck [2] and Jeffries et al. [6], does not scale.*

To prove a negative is difficult, often approaching impossible. We are left, therefore, with a proof by contradiction – we assume the hypothesis and show that it can’t be achieved. Such is the case with this hypothesis. In this paper I argue that XP as the four values is inherently scalable, but these twelve particular practices prevent the values from scaling. I then propose some new practices that alleviate the failings of the existing practices in the particular case of large projects. No comments are made relative to these approaches for small projects.

2 THE FIVE REASONS
In particular, the five reasons XP can’t scale are:

- Pair Programming expands quadratically
- Planning Game is too introspective
- Collective Ownership leads to chaos
- Metaphor is too weak
- No means to effectively coordinate multiple teams exists in the other practices

When I use the term “scale” above, I mean for teams and problems of a size larger than XP was intended. I don't know what this number is, but let's say it's somewhere between 10 and 20 team members and some number of function points (or other metric) in a period of time that is beyond the capacity of a given team. There is likely a bias in this discussion toward a model of scaling XP that implies multiple (roughly) co-equal teams. I recommend against a scaling of XP into one large team, as it seems to magnify the issues without providing a means of mitigating them. Indeed, Brooks [3] notes the issues associated with building teams too large.

Pair programming expands quadratically
Pair programming (PP) is too broad of a term to claim it doesn't scale. In fact, there's likely nothing about PP alone that doesn't scale, as long as it only focuses on the programming task itself – the arrangement of two programmers collaborating in real time to solve a problem. The part of PP that doesn't scale is its use in XP beyond the programming task itself. PP in XP is used to provide a medium to communicate design as well as familiarize developers with the code base. The pair-wise communication paths of a team grow as the square of the number of members in the team; double the team size, quadruple the communication paths.

As the team grows in size, it may become distributed to other buildings, other cities, or even other countries. Complexity is added to the pairing process - we have to coordinate across sites, time zones, and languages. Local cliques will creep in; teams will focus on parts of the system and avoid others. A partitioning of system knowledge has occurred, but not a rational one. The system has now become N interacting systems and N interacting teams, yet there is no reasoned approach to this interaction, and there is no mechanism for coordination.

Planning Game is too introspective
The planning game is a means to ensure that the team works on the most important tasks at any time. As the team grows, the number of tasks will grow as well. As the

project grows in scope, the amount of important work will grow correspondingly. As both grow, their effects multiply. Combine this with distributed teams and we have a problem: There is no means to ensure that each team is working on the right things relative to the other teams. You can use the cop out answer that "the customer coordinates this by picking what's important for each iteration," but would you want the customer to be exposed to this degree of your development process? Is it really the customer's responsibility to help you resolve your development difficulties? I don't think so – go back to one of the tenets of the approach: Customers make business decisions; engineers make technical decisions.

It is likely (and occurred several times in my experiences) that a team needs to work on a less-important (from the business perspective) task to ensure that an important system-wide behavior is available. Alas, there is no XP means to do this, as there is no overall coordination role defined in XP.

Collective Ownership leads to chaos
Collective ownership is a good thing in XP. It allows work to proceed without it being dependent on a single particular individual to do the work. That individual may know the area better than others, but the risk of that individual becoming a bottleneck is reduced through the use of collective ownership. Unfortunately, as the scale grows, collective ownership changes from a benefit to a potential source of churn and complexity. The churn comes from two developers (or two pairs) adding conflicting features to the code base. The complexity comes from resolving the interactions of these new features, even when they're not directly conflicting but only cohabiting. Add to this the requirements on tools to support this type of problem resolution, and a huge potential risk has been added to the technical side of the project. Bring on the bonus of no overall coordination and it's game over.

Metaphor is too weak
Metaphor is the least well defined of the XP practices. My reading of Metaphor indicates it uses as a general compass for understanding the solution space. This use of metaphor is sufficient because PP reinforces the metaphor continually.

Metaphor is insufficient as design, though, as its relationship to the solution is often rather loose. Indeed, as the team size grows, the ability for the metaphor to be sufficient decreases. Combine this with the decreasing ability of PP to meet the communication needs of the team leads to a team that can quickly diverge instead of converge.

No means to effectively coordinate multiple teams in the other practices
As was noted above, there is no notion of coordination across teams in XP because XP is oriented at a single team. This is one of those items left out of XP because it was not an issue in the target environment of XP.

However, this leaves a dilemma for scaling XP: Scale XP by making a single large team and risk collapse under the communication burden; scale XP by growing several collaborating teams, and risk collapse due to lack of coordination.

3 SOLUTIONS

I've heard it stated that "[t]o point out issues without providing solutions is whining," and since I don't want to whine, here are some recommendations for new practices to add to XP (replacing existing practices) that allows XP to scale successfully.

Based on my experiences, the following new practices are required to support a multi-team XP environment:

- Loosely coupled teams
- Team coordination layer

Loosely coupled teams
It seems clear to me that to effectively scale XP requires choosing the latter of the options presented above, specifically to create a collection of loosely coupled collaborating teams. This is both a practice and a philosophical position. The practice part comes from breaking the project into these teams; the philosophical part is that you have to keep these teams focused and functioning. It could be argued that this practice replaces Pair Programming, Collective Ownership and Coding Standards in the highest-level of the project, and imposes requirements for a new practice.

My experiences in multiple-team development lead me to a strong recommendation: the "prime" team should not dictate or impose a process on the "subcontracting" teams.[1] However the "prime" team should indeed impose the feature roll-out plan (the equivalent of the XP "Release Plan") on the teams. Our approach imposed the order of deliverables and used a consensus of the development teams to decide the dates, and adjusted the dates as necessary to meet the business objectives.

This position leads to a collection of subcontractors, each with their own development process but with shared dates and deliverables. As the teams have their own development processes, there's no real way to coordinate any of the rules of the game – specifically rules that could be used to share development across the teams, such as the coding or CM rules. As there's no real way to ensure the same rules across the teams, it is unlikely that any coding can occur across teams, obviating the need for Pair Programming or Collective Ownership at the "across the teams" level.

Team coordination layer
To resolve the issue with coordination, a coordination

[1] This recommendation is counter to that proposed by Carmel[4], but this type of dictation is almost a guarantee for failure; see [1] for further explanation.

layer is added (arguably outside the scope of XP) to the project to support the team interactions. This can be viewed as a layering the project structure, where at the highest level in the hierarchy there is a collection of teams being coordinated. At the bottom are the individual teams that are doing the work. This type of structure can go on as necessary, allowing the projects to scale to enormous scope.

The Team Coordination layer replaces the Metaphor, Pair Programming, Collective Code Ownership, and On-site Customer practices with "Architecture Lite," adds a role ("Liaison") and augments Planning Game to work with multiple teams, keeping the project needs in full focus at all times.

Up-front Architecture Lite

The division of tasks among the various teams must be rational – a team should work a problem for a reason. That reason can be availability of workers, but that should be the last reason on the list. Indeed, the tasks should be allocated to teams in such a way as to minimize the required day-to-day communication between the teams to get their job done. This, in turn, requires that some structure be provided to the solution space to direct this allocation. Traditionally, this would be the role of the "Architect" and architecture in the project.

To meet the needs of the project, though, often a full "architecture" is not required. Rather, an "Architecture Lite"[5],[7] is more appropriate. An Architecture Lite fully describes the system architecture but at a high level of abstraction. The description includes both structural and behavioral aspects of the system. In our use of this approach, we used the architecture lite to define and use a few strongly held key principles. Abstractly, these are:

- Low coupling among the elements
- Well defined interfaces
- Concise semantics for the element behaviors.

This Architecture Lite replaces Metaphor and Pair Programming, the latter from the perspective of its use to share system design information. One of the keys in my experiences was to provide a stronger metaphor than Metaphor typically provides. Indeed, we defined a partitioning of functionality and behavior into "boxes" that formed the architecture. This allowed us to then carve up the work and assign it to the teams in a way that leveraged their particular strengths (which include not only their capabilities but any conveniences caused by their location) maximally.

We rely on these principles to guide us when a behavior emerges. Since the elements behavior and interfaces are well defined, they could be developed in a context that was shared among the teams. As the network elements had low coupling, each team could run independently of both any other team as well as the architect. The net result of managing complexity at this level is that we are free to distribute the development of any element to any team.

Note also that one of the roles of the Pair Programming practice was to ensure that design knowledge permeated the team. The Architecture Lite replaces this role at the highest level in the project, explicitly acknowledging the loss in fidelity of the information.

Liaison

The role of liaison was one we stumbled on, but is one that is critical to success in a multi-team environment[1]. Liaisons are members of the various teams that join forces to develop the Architecture Lite. By collaborating on the foundation of the system, each of the liaisons has intimate knowledge of the rules of the system – the principles of the system. Since each team has a liaison, this person can act as the conscious of the Architecture Lite, to ensure that the principles are kept. The Liaisons allow the teams to work independently by filtering the required communication among the teams, as they have intimate understanding of the roles of the other teams. As the project evolves and emergent behaviors cause the Architecture Lite to become invalid, the Liaisons initiate changes to the Architecture Lite to continue to be a useful tool.

The Liaisons also fulfill an important role abdicated by the Architecture Lite – they are the keepers of the high-resolution knowledge of the code that implements the Architecture Lite. In sub-teams that are XP, any member should be able to fill this role, reducing overall project risk (by reducing the team's "Bus Number").

Team-wise planning game

The Architecture Lite allows for a rational distribution of work across the teams and the Liaisons allow each team to proceed without much interaction, but neither of these resolves the issue of coordinating when such work is completed. Planning across teams is a difficult problem; each team can have its own set of issues. A Team-wise Planning Game (TPG) is similar in many ways to the existing Planning Game practice, so it is more correctly viewed as an enhancement to that practice rather than a new practice.

The key aspect of this approach is to agree to participate in the approach throughout the project; the TPG coordinates the development of those features across the teams. Each individual team is represented in the TPG – much as the individual developer is represented in the PG. However, instead of representing an individual, they represent the team. They sign up for the work associated with their team, based on the structure provided under the "Architecture Lite." If the new work breaks the architecture, the architecture is reworked for the next iteration (as the Liaison role notes).

In the project discussed in [1], we had a team of drivers of this coordination. In a certain sense, this team interacted with the customer to determine what the important business feature was, and then interacted with the multiple development teams (as proxy customer?) to coordinate their activities. In the role of proxy customer, the coordination team could ensure the teams worked on what was important from the global perspective.

We had weekly teleconferences among the teams to ensure information about progress was being shared, and to understand the impacts of any late team on any other teams progress. Often the tasks were inter-dependent at the system level, in the sense that some portion of the system required both teams to complete their work to support some system-wide behavior. We used the weekly meetings to remind the teams of these dependencies.

This approach works best where the sub-teams are XP, as they are best able to respond to changes in requirements from iteration to iteration.

4 CONCLUSION

Presented above are the five reasons that project of sufficiently large scope, larger than that of a single team, will not be successful with XP. Also presented are the ways to improve their chances of being successful. These ways include the addition of some up-front coordination work and active management of multi-team issues, both of which are beyond the scope of the currently defined XP. Therefore, it is my conclusion that for XP to be successfully scaled, these practices must be included in the method. It is possible (even likely) that other changes are required. In that sense, these changes are only necessary but not sufficient to guarantee success.

5 REFERENCES

[1] Battin, R., R. Crocker, J. Kreidler, K. Subramanian, "Leveraging Resources in Global Software Development," *IEEE Software*, March 2001.

[2] Beck, K. *Extreme Programming Explained.* Addison-Wesley, 1999.

[3] Brooks, F. *The Mythical Man-Month, Anniversary Edition*. Addison-Wesley, 1995.

[4] Carmel, Erran. *Global Software Development Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall PTR, 1999.

[5] Coleman, Derek, "Architecture for Planning Software Product Platforms," Tutorial presented at *The First Software Product Line Conference*, August 30 – September 1, 2000, Denver, CO.

[6] Jeffries, R., A. Anderson, C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[7] Toft, P., D. Coleman, J. Ohta, "A Cooperative Model for Cross-Divisional Product Development for a Software Product Line," in Proc., First Software Product Lines Conference (SPLC1), August 31-Sept. 1, 2000, Denver, CO.

[8] Williams, Laurie, Robert R. Kessler, Ward Cunningham, and Ron Jeffries,"Strengthening the Case for Pair Programming," *IEEE Software*, July 2000.