A Study of Extreme Programming in a Large Company

Neil B. Harrison
Avaya Labs
1300 W. 120th Ave.
Westminster, CO 80234
1-303-538-1541
nbharrison@avaya.com

ABSTRACT

Agile software development is an approach to software that focuses on lightweight processes and adaptability to change. The best-known agile methodology is called Extreme Programming. It suggests twelve practices that include iterative development practices, automated unit testing, and pair programming.

Extreme Programming is designed for small projects, but has been picked up through grassroots efforts in some large projects in large companies, including Avaya. We studied six such projects in Avaya. We were interested to learn how projects adapted Extreme Programming to their needs, and which of the twelve practices they used.

Projects adopted iterative development practices, along with dynamic prioritization of work. Although Extreme Programming downplays architecture, every project retained focus on software architecture. They had mixed success moving to automated unit testing, and most used some form of pair programming. These practices appear to be both beneficial and practical for projects of various sizes in large companies, and we recommend them for those wishing to use agile development practices in large companies.

Keywords

Agile Software, Architecture, Extreme Programming, Process

1. INTRODUCTION

Two of the sexiest terms in software development these days are "Agile Software Development" and "Extreme Programming." Both propose a different approach to software development than the traditional waterfall model, and both promise productivity improvements. The methodologies appeal to many software developers, and many have embraced them.

Do these methodologies work in large projects? Extreme Programming explicitly excluded projects larger than twenty people in its earliest description. In spite of this, several projects within Avaya have begun experimenting with agile software development, and Extreme Programming in particular. Their experiences, while preliminary, are providing a framework for further agile development methodology. This paper describes a study of their experiences.

This paper begins with a short summary of key practices in Extreme Programming, and a discussion of large project characteristics that are incompatible with the Extreme Programming practices. It then describes the study we conducted, and its results. It then explores key practices from XP that might be adapted for use in large projects.

2. EXTREME PROGRAMMING

Extreme Programming (XP) burst onto the software scene in 1999 with a book by Kent Beck called "Extreme Programming Explained" [1]. It was based on the experiences of Beck and Ron Jeffries as consultants to a project in the Daimler Chrysler corporation, as well as software process works by Cunningham and Coplien, and Beedle [2, 3, 4]. XP caught on quickly, resulting in user groups, conferences, and other books [5, 6, 7, among others]. More recently XP has been identified with the more general Agile Software Development [8, 9, 10, 11].

Because of its departure from software development practices that are associated with quality software, and with predictability of the software development process, XP has been controversial [12]. Studies of XP projects are beginning to be published (see [13, 14]). However, XP is new enough that there are few empirical studies of the results of using XP, particularly on medium or large projects. Yet its popularity demands that it receive more than an emotional debate, it deserves critical examination.

2.2. Technical Foundations of XP

The technical premise of XP is that the cost of software change over time does not have to increase exponentially over time, but rather can be made to increase much more slowly, eventually reaching an asymptote. This leads to dramatically different behavior, where decisions are deferred until as late as possible, and implementation focuses on doing the simplest thing that could possibly work.

Beck describes twelve practices that are designed to help keep the cost of change low over time, and to take advantage of that low cost. They are [1, see also 15, 16]:

- *The Planning Game* – Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
- *Small Releases* – Put a simple system into production quickly, then release new versions on a very short cycle.
- *Metaphor* – guide all development with a simple shared story of how the whole system works.
- *Simple design* – The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- *Testing* – Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- *Refactoring* – Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.
- *Pair programming* – All production code is written with two programmers at one machine.
- *Collective ownership* – Anyone can change any code anywhere in the system at any time.
- *Continuous integration* – Integrate and build the system many times a day, every time a task in completed.
- *40-hour week* – Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.
- *On-site customer* – Include a real, live user on the team, available full-time to answer questions.
- *Coding standards* – Programmers write all code in accordance with rules emphasizing communication through the code.

2.2. Limitations in Application of XP

Proponents of XP freely admit that XP is not appropriate for all projects. The most obvious constraint on XP is the size of the team. According to Beck, "You probably couldn't run an XP project with a hundred programmers. Nor fifty. Nor twenty, probably. Ten is definitely doable." [1, p 157] Taber and Fowler [17] report a project with 25 developers on a project, divided into two teams. Grady Booch, at a talk to the Denver XP users' group, took an informal poll of the participants about the largest XP project they had ever worked on; the consensus was 30 people. He noted that this was consistent with results from other talks. Beck points out that the biggest bottleneck in scaling XP is the single-threaded integration process.

XP has its origins in projects that are custom software for a single customer. The Chrysler project was a payroll system for the company. This tends to make *On-site customer* easier to accomplish, and having the customer write acceptance tests a viable quality assurance strategy. It also tends to make flexible prioritization of features, which is the hallmark of *The planning game*, workable. However, software written in Avaya is for mass-produced products. We were interested to see whether these practices had been adopted, adapted, or eschewed in the XP projects we studied.

Other factors may make XP an inappropriate methodology. Some are technological, such as a system with very long build times, or where testing is impractical or impossible (e.g. space shuttle software).

There are non-technical factors that make XP difficult to adopt in a large company. Commitment to ISO standards tends to foster company-wide methodologies, involving numerous quality checkpoints and records. For XP to be used, it must somehow fit within such a framework. Other non-technical factors include geographic distribution of development teams.

Within Avaya, many projects are large, geographically distributed, have large legacy code bases, and have very stringent quality requirements. Each of these is in some way incompatible with XP, yet some organizations found benefit from parts of XP. We were interested to learn how they adapted XP practices to their projects. Ideally, we hoped to find practices that could be broadly useful across the company.

2.3. Other Agile Methodologies

XP is not the only so-called agile development methodology. There are a variety of methodologies that subscribe to the "Manifesto for Agile Software Development", which states:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more [8].

Agile methodologies include Crystal methodologies [9], Adaptive Software [11], and SCRUM [2]. There are certainly also many individual approaches to agile development that do not subscribe to a published methodology.

The reason this study focused on XP rather than agile practices in general is because the projects identified themselves with XP specifically. It turned out that it is probably more appropriate to call their practices agile rather than the more specific XP, however.

3. THE STUDY OF XP PROJECTS

Within Avaya, some individuals or small groups have begun to study XP. Independently of each other, and largely without knowledge of each other, several projects have begun to follow some XP practices. The projects tend to be small, or small sub-projects within a large project. All the projects operate under the ISO certified corporate software development process guidelines. Some of the projects have written documents that describe how their use of XP fits within the corporate guidelines.

We studied six projects. Within those projects, we interviewed fifteen people. The projects ranged up to forty people, with two to twenty developers. Two were parts of larger projects; the size of the larger projects is not included in the numbers above. All the projects except one are still in the early to mid phases of development. The one project to finish was canceled for business reasons about the time it completed development, so there is no comprehensive quality or productivity data. However, the practices of the projects were very instructive. They are described later.

3.1 Approach

The primary method of gathering information was through personal interviews. The author followed a script of questions, and recorded the answers. Most interviews were conducted individually, although a few were conducted with two or three individuals of the project simultaneously. Multiple person interviews may have produced more information, but not substantially. One project was located in Australia, so an email exchange replaced the personal interview. In most cases, developers were interviewed, although in one large project, the author was able to also interview a project manager, a product manager, and an architect. In another project, the project manager was interviewed together with a developer.

Because the original goal of the study was to determine which practices (XP or non-XP) would be good candidates for an agile development process in Avaya, the study did not do a rigorous enumeration of the XP practices. In fact, the study of the first project asked about compliance with each of the twelve XP practices. However, the responses were somewhat sterile, and did not effectively highlight those practices that were considered to be particularly effective, as well as a departure from the status quo. In addition, most projects were not far enough along in their development cycle to assess the effectiveness of, say, collective ownership. In short, it was more relevant for us to find out what the projects did, whether it was "official" XP or not, than to assess some level of compliance to the XP practices.

Therefore, we simplified the questions. We asked about project characteristics, and asked them to describe how they used XP. We asked what were the most important aspects of XP for them, and how well it worked. These questions elicited a fairly complete description of projects' use of XP, and focused on what worked best. Where people didn't mention a particular XP practice, we asked about it in most cases.

4. FINDINGS OF THE STUDY

There was fairly strong agreement among the projects on which practices they followed, and which they felt were important. The following is a list of the findings from the study.

As expected, no project followed XP exactly. A major reason for this was that most of the projects studied were small parts of larger projects. Therefore, they could not implement all the practices of XP; they did what they could in their own situation. It was not clear from the study which other XP practices they would have implemented if they had had the chance. However, since a large part what we wanted to learn was which practices work in our culture, we were interested in the empirical rather than the hypothetical, anyway.

Because of the position in the schedule, as well as the fact that they were in larger projects, it was impossible to collect reliable quality and productivity metrics. However, it appeared that their productivity compared favorably to standard development methodologies in the company.

It was significant to note that every person interviewed was enthusiastic about the methodology. All intended to continue their practices, and expand them where possible.

The following list describes the teams followed each of the twelve XP practices:

- *The Planning Game* – No project implemented the Planning Game fully. While planning was flexible, it was required to be within the boundaries of the larger product plan. This had two major ramifications. First, an overall project plan was established. This was necessary not only for the project as a whole, but also for other teams, in particular documentation, sales, marketing, and product support. Second, others' needs often constrained prioritization somewhat.

- *Small Releases* – All projects used small releases in some form. The size of the development intervals was from two to four weeks. Where the XP development was part of a larger project, the development intervals culminated in deliveries into the official code base of the project.

  One team punctuated each interval with a demo to themselves and others on the project. They reported that this was a strong morale booster on the team.

  One project's data showed that they had not made substantial improvement in the accuracy of their development estimates over several iterations. They admitted, however, that they had not re-estimated after each iteration, but plan to do so in the future.

- *Metaphor* – No project had a metaphor. This is consistent with reports from Kent Beck , who stated that people tell him that they do XP, "..except metaphor, of course."[18]. Others have also noted that people have difficulty understanding Metaphor [19]. This is certainly one major reason that no project used metaphor.

  The other major reason is that XP's *Metaphor* intends to fill some of the purpose of traditional software architecture, namely creating a shared vision of the system to be built. Every project in the study did produce an architecture. This was no doubt partly due to the influence of existing practices, but no project even considered creating a metaphor rather than an architecture. Nobody was interested in metaphor.

- *Simple design* – Projects did not highlight simple design as an important part of their XP process. This should not be construed to imply that they did not have simple designs, but rather that it was not an important difference from their traditional processes.

- *Testing* – All projects intended to require that tests be submitted along with code, creating a body of automated regression tests. One project followed this rigorously. Other projects followed it partially. The major reason for this was schedule pressure with focus on delivered functionality. A related reason was the time and effort were not available to set up an automated regression testing system, particularly where the XP project was part of a larger project.

  All projects were for software to be sold to multiple customers, so it was not realistic to have a customer write and execute acceptance tests. However, every project did have an extensive system verification program. In this model, the system testers functioned as "surrogate customers", writing and executing acceptance tests.

- *Refactoring* – Refactoring did not figure prominently in the projects. The only project to refactor frequently was a forward looking work project with three people on the team. The other projects indicated that they were not opposed to refactoring, but there really hadn't been a need to do so. This may be a reflection of more up-front design than is typically done in an XP project.

- *Pair programming* – All projects did some form of pair programming, but each did it a little differently. No project required it for every line of code; in fact in every project, pair programming was voluntary. In one project, developers began by doing all programming as pairs, but found it was too inefficient. So they programmed the simple code individually, but paired up on the difficult code. Another project had two developers, located 2000 miles apart. They tended to write code individually, but debugged their code together, using a shared desktop. One developer pointed out that this was actually more convenient, because they weren't crowded against each other.

  Code inspections are standard practice in Avaya. In one instance, pair programming was allowed to replace code inspections. The project did not have data to indicate whether one was more effective than the other in finding errors.

- *Collective ownership* – Code ownership practices varied from project to project, due in part to constraints of the surrounding projects. Where collective ownership was practiced, there was a practice of de facto code ownership: people gained natural expertise in certain parts of the system, and made the bulk of changes there. One project codified this practice into a "lightweight ownership" policy: one could change any of the code, but needed to check with the owner of the code for advice.

- *Continuous integration* – In most cases, projects worked within the larger project methodology of integration. This was generally weekly integrations into the main software base, although the XP sub-projects were able to integrate more often. In one standalone case, the team members integrated continuously.
- *40-hour week* – One or two projects made this a policy. However, no project appeared to have suffered extended periods of long hours.
- *On-site customer* – No project had an on-site customer. As stated above, this model is not practical where one has many customers or potential customers. In addition, it is usually not desirable; the project gets only a single view among many customers. Projects continued to use a surrogate customer model, where an aggregate view of customer needs is created.
- *Coding standards* – Avaya has had a tradition of coding standards. The XP projects followed their pre-existing coding standards.

In brief summary, projects followed XP most closely in the coding activities. As expected, where the projects touched other organizations in the company, XP practices were weak, or not followed at all.

5. APPLYING XP IN LARGE COMPANIES

The experiences of these projects indicate that some of the XP practices can be very useful in large companies. Of course, they must be adapted to fit into a company's processes and organization. The following practices are recommended.

- Projects may dynamically prioritize requirements in a manner similar to The Planning Game or other ways, such as Work Queue [4] or Adaptive Cycles [11]. It is important to note, though, that these work subservient to the needs of customer-facing organizations including marketing, documentation, and strategic planning. Therefore, projects will commit to some set of functionality, and implement additional features as possible. This becomes a workable compromise, allowing a team to commit to a schedule several months away, but having some flexibility about the content.
- Continue to create software architecture. Dikel [20], Garland [21], and others note the importance of architecture in large systems, and guidelines can be found in many sources, such as [22, 23]. An architecture will help create a shared vision of the system, partition the system into comprehensible parts, help provide natural units of work for implementers, and provide a first order check whether the system design can meet the requirements, particularly non-functional requirements such as reliability and performance. (See, for example, [24] and [25].) Note that agile development recommends parsimony of documentation, see [26, 27].
- Short iterations will be an important part of a large agile project. They can be used even where only a part of a project adopts agile processes. Note that large projects tend to have many internal dependencies. This will affect the content of iterations, but need not affect the standard duration of the iterations. Also, some work items, including infrastructure work, may be too long to complete in a single iteration. Developers should break down such work into pieces that can be completed in an iteration, if possible.
- Extensive unit testing, strongly advocated in XP, is naturally appropriate in large projects. Automated testing is a must. In large projects, the body of regression tests may grow enough that running all of them daily may become cumbersome. In that case, tests may have to be written strategically.

  XP says nothing about system-level testing, but it will be needed in virtually all large projects. The short iterations described above can give system testers opportunities to receive a working system much earlier than they have in the past; this can be a significant advantage for testing.
- Large projects can benefit from a relaxed code ownership policy rather than complete collective ownership. It is impossible for every developer to become proficient in all parts of the system, but code owners can serve as mentors. They should be aware of changes to their areas, but need not be the only one making the change.
- It is typically impossible to have a customer on-site for large projects, or any project with a large number of customers. But someone must represent customer interests, and be in touch with (probably several) customers. Use cases would be appropriate for large agile projects.
- Other XP practices may be adopted as they fit with the project's processes and culture. Pair programming may fit with the culture, and is recommended (see [28]). The system can probably not be integrated continuously, but can be done frequently.
- Projects may wish to consider other agile methodologies rather than XP specifically. In addition, they should focus on issues of organizational structure and communication, as outlined in [3, 29, and 30].

6. CONCLUSIONS

Much of XP is proving to be useful to projects within Avaya. No project follows all of XP. There were two major reasons for deviance from all the XP practices. The first was that certain XP practices were not appropriate for large software projects. The second was that some practices did not fit with other practices in the existing projects. For the most part, the projects benefited from the practices they did adopt.

Large projects in other companies may benefit from adopting and adapting the practices that the Avaya projects found useful. Those projects may wish to explore agile methodologies in general, rather than focus on XP, which is most appropriate for small projects.

7. REFERENCES

[1] K. Beck, *Extreme Programming Explained*, Reading, MA: Addison-Wesley, 2000.

[2] M. Beedle, et al., "SCRUM: A Pattern Language for Hyperproductive Software Development," in *Pattern Languages of Program Design 4*, N. B. Harrison, B. Foote, and H. Rohnert, eds., Reading, MA: Addison-Wesley, 2000, pp.637-652.

[3] J. O. Coplien, "A Generative Development-Process Pattern Language," in *Pattern Languages of Program Design*, J. O. Coplien and D. Schmidt eds., Reading MA: Addison-Wesley, 1995, pp. 183-238.

[4] W. Cunningham, "EPISODES: A Pattern Language of Competitive Development," in *Pattern Languages of Program Design 2*, J. M. Vlissides, J. O. Coplien, and N. L. Kerth, eds., Reading, MA: Addison-Wesley, Reading MA 1996, pp. 371-388.

[5] K. Beck and M. Fowler, *Planning Extreme Programming*, Reading, MA: Addison-Wesley, 2000.

[6] R. Jeffries, *Extreme Programming Installed*, Reading, MA: Addison-Wesley, 2000.

[7] J. W. Newkirk and R. C. Martin, *Extreme Programming in Practice*, Reading, MA: Addison-Wesley, 2001.

[8] K. Beck et al, "The Agile Alliance Manifesto," available http://www.agilemanifesto.org/principles.html.

[9] A. Cockburn, *Agile Software Development*, Reading, MA: Addison-Wesley, 2002.

[10] M. Fowler and J. Highsmith, "The Agile Manifesto," *Software Development*, vol.9, no. 8, pp. 28-32, Aug. 2001.

[11] J. Highsmith, *Adaptive Software Development*, New York, Dorset House, 2000.

[12] B. Boehm, "Get Ready for Agile Methods, with Care," *IEEE Computer*, pp. 64-69, Jan. 2002.

[13] R. Gittins, S. Hope, and I. Williams, "Qualitative Studies of XP in a Medium Sized Business," University of Wales Bangor, Bangor, Wales, 2002.

[14] B. Rumpe and A. Schröder, "Quantitative Untersuchung des Extreme Programming Prozesses," Munich University of Technology, Munich, Germany, Technical Report TUM-I01, 2001.

[15] M. Fowler, *Refactoring*, Reading, MA: Addison-Wesley, 1999.

[16] L. Williams, "The Collaborative Software Process," University of Utah, Salt Lake City, UT, PhD. Dissertation, 2000.

[17] C. Taber and M. Fowler, "An Iteration in the Life of an XP Project," *Cutter IT Journal*, vol. 13, no. 11, pp. 13-20, Nov. 2000.

[18] K. Beck, "The Metaphor Metaphor," Invited Talk, OOPSLA 2002, Advance Program avaliable at http://oopsla.acm.org/ap/files/spe-metaphor.html.

[19] M. Fowler, "Is Design Dead?" available at http://martinfowler.com/articles/designD.html.

[20] D. M. Dikel, D. Kane, and J. R. Wilson, *Software Architecture: Organizational Principles and Patterns*, Upper Saddle River, N.J.: Prentice Hall PTR, 2001.

[21] J. Garland and R. Anthony, *Large-Scale Software Architecture*, Chichester, England: John Wiley & Sons, 2002.

[22] F. Buschmann, et al., *Pattern-Oriented Software Architecture: A System of Patterns*, Chichester, England: John Wiley & Sons, 1996.

[23] P. Clements, (ed.) et al. *Documenting Software Architectures*, Reading, MA: Addison-Wesley, 2002.

[24] R. Katzman and L. Bass, “Toward Deriving Software Architectures From Quality Attributes,” Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, SEI Technical Report CMU/SEI-94-TR-10, 1994.

[25] M. R. Barbacci et al., “Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis,” Software Engineering Institute, Carnegie-Mellon University, Pittsburgh PA, SEI Technical Report CMU/SEI-97-TR-029, 1998.

[26] S. W. Ambler, “Lessons in Agility from Internet-Based Development,” *IEEE Software*, pp. 66-73, March/April 2002.

[27] M. Fowler, “The Almighty Thud,” *Distributed Computing*, Nov/Dec 1997, available http://www.martinfowler.com/distributedComputing/thud.html.

[28] L. Willaims, et al, “Strengthening the Case for Pair Programming,” *IEEE Software*, pp. 19-25, July/August 2000.

[29] N. B. Harrison, “Organizational Patterns for Teams,” in *Pattern Languages of Program Design 2*, J. M. Vlissides, J. O. Coplien, and N. L. Kerth, eds., Reading, MA: Addison-Wesley, Reading MA 1996, pp. 345-352.

[30] N. B. Harrison and J. O. Coplien, “Patterns of Productive Software Organizations,” in *Bell Labs Technical Journal*, vol. 1, no. 1, pp. 138-145, Summer 1996.