micro-eXtreme Programming (mXP): Embedding XP Within Standard Projects

Frank A. Adrian
Symantec, Inc.
15220 NW Greenbrier Pkwy., Suite 200
Beaverton, OR 972006
+1 503 614-7904
fadrian@symantec.com

ABSTRACT
In many organizations, XP might not be embraced - XP, when used at all, must be practiced at an individual level. In this paper, we describe a methodology for using XP at an individual level within a standard project framework that we call micro-eXtreme Programming (µXP).

KEYWORDS
Software engineering, extreme programming, process improvement.

1 INTRODUCTION

Using XP is such a good idea, it often seems unbelievable that some organizations would reject it! Oddly enough, though, some do. What happens if you find yourself in this situation? Can you find a way to use XP when no one else wants to?

We find that many of the principles of XP can be re-focused to an individual level to make individual programming tasks more productive. In this paper, we describe a scaled XP process that individuals can use within a traditional project framework, describe why our methodology works and answer objections to the method. We finish with conclusions.

2 THE PROBLEM

There are cases where an individual programmer cannot use XP. A manager may not embrace the XP philosophy – she sees pair programming as a waste of resources and XP coding practices as an excuse for hacking and avoiding formal design. Convincing teammates to use XP may be a problem. As an independent consultant, you might not have a coding partner to pair with. For small or experimental projects, incurring the overhead of arranging pairing to carry out a task may be undesirable.

To address these issues, we scale XP to produce a methodology that uses its practices, but in a form that an individual within a traditional project framework can use. The practice also supports introducing XP into an organization in a non-threatening manner. We call our method µXP (micro-eXtreme Programming).

3 WHAT IS mXP

XP itself is still an amorphous concept - references [2 – 6, 12] differ in their definitions.. In our work we started with core principles taken from [1]. µXP has XP best practices at its core and many XP practices carry over to µXP without change. Other XP practices have undergone subtle shifts in focus. Two XP principles have been omitted from the µXP cannon and one weakened.

The first tenet sacrificed is pair programming. We enjoy pair programming and believe that, if it is possible, it is a great practice to improve code quality. However, it does not fit into the "individual programming" goal that we are exploring and must be removed.

Another principle left behind was collective ownership. Many organizations are too segmented or projects too fragmented to allow this practice. Sometimes, you are working on your own code and the issue is irrelevant. In order to work under these conditions, this practice was elided.

We weaken XP's on-site customer principle. In many organizations, customers cannot or will not be on-site. At our level of focus, product-level concerns are usually already defined, so this is not a major issue.

mXP Principles

The principles of µXP are as follows:

Planning Game – Just as in XP, work is organized via a set of stories. However, the stories used in µXP are so much smaller that we call them paragraphs. Estimates are made and tracked at the paragraph level to determine µXP coding velocity.

Continual Re-prioritization – In µXP, paragraphs are broken and combined often. Between each iteration, one re-prioritizes work to be done. At this level, work is divided for technical and not business reasons, so one does not need a customer standing by to re-prioritize every ten minutes. Continual re-prioritization has the advantage that work remains flexible in the face of design changes.

Small Releases – As in XP, each paragraph is a small, complete fragment that must be integrated into the body of code as a whole. Paragraph-size code fragments are so small that integration is fairly simple, but defects can occur and the unit tests for the next higher level of code are run so that integration errors do not accumulate.

Metaphor – We use the same metaphoric programming practices as XP does. We check our paragraph level code to insure that it fits with the metaphor we are using to design the rest of the system.

Simple Design – µXP believes in simple design. We still include those two key XP sayings "Do the simplest thing that could possibly work" and "You ain't going to need it." In XP, spiking is used architecturally to get code working earlier; in µXP, we use "design spikes" to get working code as soon as possible.

Testing – In XP, the tenet is to test early and test often. So it is in µXP. Unit tests are written for sentences and paragraphs before the code itself is written. Tests are continually run to insure that current changes do not break previously existing code.

Refactoring –µXP retains XP's heavy emphasis on refactoring. The use of design spiking and simple design often produces duplicated code. Refactoring must be performed ruthlessly and continually.

Sustainable pace – XP has the concept of a 40-hour week; µXP has an 8-hour day. It is difficult for people to work more than this length of time without becoming fatigued and introducing error. During the planning game, no more than five ideal hours of work (with allowances for coding velocity) are scheduled during an ideal day.

High-availability Customer – In XP, the customer must be available on-site. In most µXP projects, this is not feasible. In its place, we require high-availability of the customer. In general, the customer must be available to answer paragraph-level issues and this implies that answers to a question should be available within a couple of hours. In most cases, issues are already answered at a story level.

Coding Standards – As with metaphor and refactoring, this low-level XP practice is carried over unchanged, but with additional emphasis due to µXP's closeness to the code.

4 CONTINUAL RE-PRIORITIZATION

If there is one thing unique about µXP, it is the principle of continual re-prioritization. Using µXP within the scope of a traditionally structured project, there are more external dependencies to manage than in a standard XP project. Waiting on project dependencies would cause interminable delays. To smooth project flow, we use continual re-prioritization.

In XP, between story-level iterations, the practitioners must re-prioritize the set of tasks to be done. In µXP, the tasks are re-prioritized within the iteration each time a sentence or paragraph is completed. We like to think of this step as continual refactoring of the schedule. Being an opportunistic practice, this also increases efficiency. An example will help illustrate…

We start with a requirements document and a set of tasks from a traditional project schedule to implement a specific feature. The first convert these tasks into stories. Then, on a combined basis of prerequisite availability and design advantage, we select a story for implementation:

Journal Processing

The items in the journal are an ordered set of commands. The commands are of the form:

- Create an Order.
- Add a Line to an Order.
- Modify a Line within an Order.
- Delete a Line within an Order.

After each command from the journal is processed, the Order will be checked for consistency.

We write our functional level tests for the story and then divide the story into a set of paragraphs:

> Create an Order.
>
> Create a new Line.
>
> Add a Line to the Order.
>
> Modify a Line within an Order.
> Remove a Line from an Order.
>
> Check the Order.

After writing tests at the paragraph level, we divide the paragraphs into sentences, prioritizing them as to design advantage:

> Create the Order object.
>
> Add a Line collection to the Order.
>
> Create a new Line.
>
> Check the Line for correctness.
>
> Add the Line to the Order.
> Modify a Line within an Order.
>
> Remove a Line from an Order.
>
> Check the non-Line portion of the Order.
>
> Check all Lines in the Order.

Combine Order checks.

As we start to implement, we construct tests and build code. Suppose we get to the item "Add the Line to the Order." and a test fails – we need to figure out what to do if a line with a duplicate ID is added to the order. We take this opportunity to re-prioritize our work, deciding that replacing the given line is the simplest way to handle this. We see that we can implement modifying a line by replacing the line, and that replacing a line is the same as removing the original line and adding a (unduplicated) line. We get the following as a result of our reprioritization:

~~Create the Order object.~~

~~Add a Line collection to the Order.~~

~~Create a new Line.~~

~~Check the Line for correctness.~~

~~Add the (unduplicated) Line to the Order.~~

Remove a Line from an Order.

Replace a (or add a duplicated) Line within an Order.

Check the non-Line portion of the Order.

Check all Lines in the Order.

Combine Order check.

The work proceeds until the sentences have been completed.

In addition to working on a single story, one can combine paragraphs from multiple stories and sentences from multiple paragraphs. This allows greater flexibility in the face of external dependencies. E.g., if we are waiting for a file format to be designed, we cannot write the code to read the file, but we can proceed with the design and test of the internal objects into which the data is to be placed. This method also allows a sort of "pre-factoring," where the "You ain't gonna need it" principle doesn't apply – if it's on a sentence or paragraph list, it's needed to implement some part of your chosen stories, isn't it?

Structured handling of paragraph and sentence level of the design leads to a more efficient coding regimen. Writing tests at the sentence level insures that the primitives are error-free. Many paragraphs turn into single sentences and the tests can be shared between the two levels.

5 WHY DOES µXP WORK?

µXP works because XP works. It uses best practices of XP and scales them down to the sub-story level. The structure that XP adds to the design process is echoed in µXP's coding practices, as are the advantages. The process structures coding tasks without turning into a straightjacket. The structuring reduces defects and increases ones confidence in the code.

Within the scope of a traditional project, µXP provides a more structured and efficient way of doing coding tasks. Building your schedule by breaking tasks down to the paragraph level and measuring coding velocity at this level will help improve estimates. Using XP methods within sections of the project will help you be more responsive when other parts of the design change (and they will, won't they?).

We also find that following µXP practices makes one a better programmer (just as following XP practices makes a team better). We are exploring whether µXP principles can be used as the basis for personal software improvement, much as CMM was used as the basis for the PSP [7 – 9].

6 OBJECTIONS

With any new methodology, people question whether or not it really works. Here are a few of the concerns raised about µXP:

Doesn't the removal of some of the core XP principles (particularly pair programming) destroy XP? We find it does not. If you can use XP, use it – it works. If you can't, we still believe a little XP is better than no XP at all. We have seen increase in design and coding errors when all XP practices are not followed, but we also see productivity gains over the monolithic design-code-test methods.

Doesn't design spiking and simple design lead to fragmented and incoherent designs? In general, due to rigorous use of refactoring, metaphor, and coding standards, our designs have become more integrated and easier to understand.

Isn't this just the same as iterative design? Yes and no. Iterative design is usually focused on breaking projects into large-scale chunks and not subdividing the smaller tasks. µXP brings structure to this lower level.

Isn't this the same thing most XP'ers do? Again, yes and no. XP does not prescribe structuring or testing at the sub-task level the way µXP does. The best XP practitioners do use these methods, but not stringently. By adding structure and tests at low levels, µXP makes coding more efficient.

Doesn't the structuring of micro-tasks add too much overhead? In practice, we find not. And the use of testing at the sub-task integration level reduces errors and builds code confidence.

My system has no validation tests. How do I know my code doesn't break the system? You don't. You should probably build some tests! But in this respect you're no worse off than you were with standard design and coding methods.

My system takes ten hours to build. How can I use mXP? With great difficulty? Seriously, you can still use µXP. You can build components in isolation, using test stubs. This allows you to code these components using µXP, integrating into the larger system less frequently. You should still build good integration tests at the system level, though.

7 CONCLUSIONS

µXP is not a panacea. At best, is a strategy used when one cannot use real XP. However, we have found up to 20% decreases in overall task time when using µXP [10]. These results are preliminary and the experiments have too small a scale for any scientifically valid conclusions to be drawn. Even so, they show µXP is able to improve the productivity and scheduling accuracy of the individual practitioner even within the scope of traditional projects. Put simply, we believe that µXP works! This is consistent with our own use of µXP. More research is needed to extend and verify these findings.

µXP can still be refined and improved. It is in the tradition of XP practitioners to vary XP to encompass whatever works. We hope that µXP inherits these pragmatic roots, as well.

Giving up XP tenets like pair programming is not necessarily a tragedy. We still believe that following XP strictly is a more effective way to pursue multi-person projects. But we are also convinced that many of the XP practices and methods can be applied to individual work.

Brooks [11] may be correct that there is no silver bullet to slay the werewolf of programming. But in the end, better programming comes down to discipline and commitment to quality. We believe that µXP is a simple way to instill both.

8 ACKNOWLEDGEMENTS

I would like to thank the reviewers and Ron Jeffries, in particular, for their suggestions on improving this paper.

9 REFERENCES

1. Beck, K. eXtreme Programming eXplained: Embrace Change. Addison Wesley. 2000.
2. "Extreme Programming Roadmap." Online document available at URL http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap. 2000.
3. Wells, J.D. "The Rules and Practices of Extreme Programming." Online document available at URL http://www.extermeprogramming.org/rules.html. 1999.
4. Jeffries, R., ed. "What is eXtreme Programming?" Online document available at URL http://www.xprogramming.com/what_is_xp.htm, 2000.
5. "Extreme Programming." Online document available at URL http://ootips.org/xp.html. 1999.
6. Beck, K., Fowler, M. eXtreme Programming Planned. Addison Wesley. 2000.
7. Humphrey, W. Introduction to the Personal Software Process. Addison Wesley Longman. 1996.
8. Bemberger, J. "The Essence of the Capability Maturity Model." IEEE Computer. June 1997. Pp.112-114.
9. ıXP – Using XP Principles for Improving Personal Software Productivity. On-line document available at URL http://www.ancar.org/xp/iXP.htm.
10. µXP – Initial Research and Findings. Online document available at URL http://www.ancar.org/xp/exper1.htm.
11. Brooks, F. "No Silver Bullet." IEEE Computer. April, 1987.
12. Jeffries, R., et.al. eXtreme Programming Installed. Addison Wesley. 2001.