

Extreme Programming (XP) Uncovered

by

Chaelynn M. Wolak
wolakcha@scsi.nova.edu

A paper submitted in fulfillment of the requirements
for DISS 725 – Research Paper Four
DISS 725 Spring 2001

Graduate School of Computer and Information Sciences
Nova Southeastern University

July 2001

Introduction

It can be argued that software is the world's most important industry. It is responsible for the increased efficiencies found in most traditional businesses as well as the primary component of many new businesses. In fact, software directly and indirectly affects every facet of life as it connects individuals, entertains, educates, and protects, etc. However, developing quality software is difficult especially now (Booch, 2001).

There are three main reasons why developing software is difficult. They are the following: developing complex software of quality is extremely tough, developing software is not getting any easier especially in Internet time, and there is an increasing shortage of skilled individuals who can do the work (Booch, 2001). In face of these pressures, the amount of software development effort required must be optimized.

Currently, most software development is considered a "chaotic" activity, better known as "code and fix." This means software is written without much of an underlying plan, and the design of the software system is cobbled together. However, there has always been an alternative to this chaotic development by using methodologies (Fowler, 2000).

These methodologies are typically known as "heavy" or "monumental". Examples of them include the waterfall, spiral, etc. They impose a strong emphasis on process especially upfront planning. Even though they have been around for quite some time, they are not noted for being very successful or popular (Fowler, 2000).

The unpopularity of these "heavy" methodologies is a result of the massive effort required throughout the process which can actually slow down development. Unfortunately, that is not a good attribute. Therefore, a new generation of methodologies has evolved (Fowler, 2000).

This modern generation is called "lightweight" methodologies. Most recently it is being referred to as "agile" methodologies in an effort to lose the negative connotations (Constantine, 2001). Nevertheless, it is a move away from software processes that are hierarchical and management driven. The new trend is cooperative styles of development where management dictate is replaced by ethical considerations of community membership (Booch, 2001).

Undoubtly, the best known "agile" methodology is Extreme Programming (XP) (Constantine, 2001). This academic research paper defines XP, details its components, and enabling practices. In addition, it describes when and how to use XP. Lastly, the paper concludes with a summary on XP.

XP Defined

Extreme Programming (XP) was developed by Kent Bent, a Smalltalk code developer, along with colleagues Ward Cunningham and Ron Jeffries (Williams & Kessler, 2000). In the briefest sense, XP is a lightweight discipline to software development adapted from the traditional heavyweight methodologies. Figure 1 in Appendix A shows a picture of the XP Process. The secret to XP is not the modeling techniques themselves but how they are applied to any software development project (Ambler, 2001). Therefore, to understand XP, one must understand the components to it.

Components of XP

XP is an important new methodology for two main reasons. First, it re-examines software development methodologies that have become standard operating procedures. Second, it is one of the lightweight methodologies created to help reduce the cost of software development (Wells, 2001, June 4a). The components to XP are made up of the following: values, principles, and the 12 practices.

XP Values

There are four values to XP. They are communication, simplicity, feedback, and courage (Ambler, 2001). Communication in XP terms focuses on building a person-to-person, mutual understanding of the problem through very minimal documentation. It emphasizes maximum face-to-face interaction with all stakeholders (Cutter Consortium, 2000).

Simplicity is critical for both the actual software program and software process so that it is much easier to explore the idea or improve upon it. This increases understanding (Ambler, 2001). Two of the rallying slogans of XP are “Do the simplest thing that could possibly work” and “You aren’t going to need it (also known as YANGNI).” Both of these slogans are the heart of XP simplicity (Fowler, 2001).

Feedback is another embraced value in XP. Every development methodology advocates feedback. However the difference with XP is that it is required so that XP programmers can feedforward (Cutter Consortium, 2000). According to Kent Beck “Optimism is an occupational hazard of programming, feedback is the treatment” (Ambler, 2001).

Lastly, there is courage. Courage can be defined as doing what is right, even when pressured to do just the opposite (Cutter Consortium, 2000). Courage is required because it is important to be able to change direction by either discarding or refactoring work that has proven inadequate (Ambler, 2001). “Courage is not just about having the discipline, it is also the resultant value” (Cutter Consortium, 2000).

XP Principles

XP also centers around seven main principles. They are incremental change, assuming simplicity, open and honest communication, local adaptation, traveling light, use

multiple models, and limit contract models (Ambler, 2001). Design alternatives are more effectively understood when they are drawn on a white board with diagrams versus lines of code. Multiple models allow each requirement to be described within its own aspect.

Nevertheless, one of the most crucial XP principles is to limit the number of contract models. Contract models are when an external group controls an information resource that the software system requires. Examples include a database or legacy system. Therefore, it is very important to mutually agree on what to do with that information resource whether to change it or leave it as is. However, by limiting the number of contract models, this helps to conform to traveling light (Ambler, 2001).

XP's 12 Practices

This section briefly describes the 12 main practices of XP. These are often cited as characterizing XP. They are the following: the planning process, small releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour work week, on-site customer, and coding standard (Beck, 1999).

The Planning Process.

The XP planning process mirrors that of most rapid application development (RAD) approaches (Cutter Consortium, 2000). This practice is sometimes called the “Planning Game.” It allows customers to define the business value of desired features, and uses the cost estimates provided by the programmers to choose what needs to be done and what can be deferred (Jeffries, 2001). The desired features from the customer are detailed in stories. Then, the programmers only implement those stories that the customer chooses.

Stories are one of the most valuable features during the planning process. The first decisions about a software project are what it could do and what it should do first. The analysis is put together in terms of stories. Each story must be business-oriented, testable, and estimable. Normally, it takes one month to come up with stories for a ten-year person project (Cutter Consortium, 2000).

Small Releases.

Every release should be as small as possible containing the most valuable business requirement. Small releases tend to provide a sense of accomplishment that is often missing in large projects (Cutter Consortium, 2000). Therefore, the software system is placed into production within a few months even before the whole problem is solved. Then, new releases are made often. Releases can range from daily to monthly (Beck, 1999).

Metaphor.

The shape of the software system is described by a metaphor that is understood between the customer and developers/programmers (Beck, 1999). The metaphor is intended to provide a broad view of the project's goal in an attempt to define an overall coherent theme from which the customers and developers can relate. Then the stories that are developed during the “planning game” are used to describe the individual features (Cutter Consortium, 2000).

Simple Design.

Simple design consists of two parts. First, design for functionality that has been defined. Second, create the best design that can deliver that functionality (Cutter Consortium, 2000). The program should be simple so that it meets only the current requirements. There is not much building for the future here. The program should contain no duplicate code (Jeffries, 2001). This practice can be summarized as “Say everything once and only once” (Beck, 1999).

Testing.

XP uses two types of testing. They are functional and unit (Cutter Consortium, 2000). Functional tests are written by the customer for the stories. The programmers write unit tests which is part of their daily activity. One of the most radical twists to XP is that these tests are done prior to any code writing. It is believed that if programming is about learning, and learning is about getting feedback as quickly as possible, then one can learn much more from tests than coding. One of the primary features of testing is that programmers cannot test their own code (Beck, 1999).

Refactoring.

The design of the software system is evolved over time. This is done through refactoring, or transformation of the existing design (Beck, 1999). Refactoring does not change the observable behavior of the system but enhances its internal structure. It can be thought of as incremental redesign. According to Fowler “Without refactoring, the design of the program will decay. Loss of structure has a cumulative effect” (Cutter Consortium, 2000).

Pair Programming.

XP has popularized the practice of pair-programming. Two programmers work side-by-side at one computer collaborating on the design, algorithm, code or test (Fraser, Beck, Cunningham, Crocker, Fowler, Rising, & Williams, 2000). This method has demonstrated to not only improve productivity but also quality (Williams & Kessler, 2000). In a 1999 experiment at the University of Utah on pair-programming, it showed that pair-programming produces code with statistically significant higher quality and that the cost increase for “two doing the work of one” is not statistically significant (Fraser et al., 2000).

In pair programming, one programmer is responsible for the typing/writing of the code while the other is responsible for the continuous review of the work. It is required that each programmer take turns doing the typing/writing while the other is reviewing. Both of the programmers are equal participants in the process and own everything (Williams & Kessler, 2000).

Collective Ownership.

Collective ownership is the practice that anyone on the team can change any of the code at any time. This lets the team perform at optimum speed because when something needs changing, it can be changed without delay (Jeffries, 2001).

This collective ownership provides another level of collaboration, which is also seen in pair programming. It encourages the entire team to work more closely together. For many programmers and managers this is a radical change from one person performing all code work (Cutter Consortium, 2000).

Continuous Integration.

XP teams integrate and build the software system multiple times per day (Jeffries, 2001). New code is integrated within the system no more than a few hours (Beck, 1999). Continuous integration allows rapid progress while keeping all programmers on the same path (Jeffries, 2001).

40-hour Work Week.

One of the fundamental rules of XP is no one can work a second consecutive week of overtime. XP tries to maintain everyone on a 40-hour work week (Beck, 1999). Overtime is defined as time in the office when one does not want to be there. If an individual works more than one week of overtime, then there is something wrong with the development process (Cutter Consortium, 2000). Nevertheless, tired programmers make more mistakes (Jeffries, 2001).

On-site Customer.

On-site customer is similar to user involvement in the traditional terms. This is one of the most talked about practices in software development – getting the customers involved (Cutter Consortium, 2000).

In XP, the customer is a dedicated individual who sits with the team throughout the whole development process. He/She is empowered to determine requirements, set priorities, and answer questions. By having the customer on-site, it improves communication throughout the whole development cycle (Jeffries, 2001).

Coding Standard.

Briefly, coding standard is where all programmers write code the same way. This is a requirement especially for pair programmers to work effectively. The way to writing code has to be standardized (Jeffries, 2001).

When & How to Use XP

XP is often characterized by the 12 practices as described above. Therefore, in order to do XP, does one have to do all 12 practices? Not necessarily. The practices are merely the

suggested route, but one needs to implement all four values of XP to be doing it properly (Hendrickson, 2000).

Out of the four values – communication, feedback, simplicity, and courage – courage is the trademark to XP. Without it, teams cannot respond to unexpected changes or to optimize performance. The four values need to be practiced. It just happens that those 12 practices help achieve the four values. However, over time new practices and new combinations of practices can make more sense later. Even though the practices may change, the XP values do not (Hendrickson, 2000). Thus, when is it best to use XP?

XP is designed for small to medium-sized teams, which is typically no more than 12 people (2000, September). The programmers do not have to be PhD level. The appropriate time to implement XP is when projects have dynamic requirements or when projects are considered high risk (Wells, 2001, June 4d).

There are at least three benefits to using XP. They are easier to learn and use, reduced overhead, and greater productivity. Since XP is a light-weight methodology, it is easier to explain and learn the process than the traditional heavier methodologies. It does not mean XP is easy, but easier to master (Constantine, 2001).

XP also reduces overhead. Design is done on-the-fly and only on as-needed basis. Typically index cards and whiteboards sketches are the only documents. In contrast, traditional methodologies emphasize many deliverables and numerous artifacts (Constantine, 2001).

Another benefit to XP is greater productivity. XP projects almost always report greater programmer productivity when compared to other projects within the same corporate environment (Wells, 2001, June 4d). In addition, with short release cycles that produce a fully functional system, results are seen very early (Constantine, 2001). Therefore, why does not every company use XP for software development?

There are some limitations to XP. First, XP does not scale. Due to the integrative approach, it is hard for some to understand exactly where the project stands. In a typical environment, upper management wants to know when each phase is completed such as design, code, or test. Thus, due to the various iteration steps, it can be hard to understand if the project is on track (Fraser et al., 2000).

Second, XP does not handle large teams well. The approach only works for small to medium-sized teams (Fowler, 2000). Lastly, XP does require highly-skilled and highly-motivated individuals which may not always be existent. XP places a premium on having premium people (Constantine, 2001). Despite these limitations, XP is an innovative and viable alternative to the traditional software development methodologies.

Summary

This academic research paper defines XP, details its components, and enabling practices. In addition, it describes when and how to use XP. In the briefest sense, XP is a lightweight discipline to software development adapted from the traditional heavyweight methodologies. The secret to XP is not the modeling techniques themselves but how they are applied to any software development project (Ambler, 2001)

The components to XP are made up of the following: values, principles, and the 12 practices. There are four values to XP. They are communication, simplicity, feedback, and courage (Ambler, 2001). XP also centers around seven main principles. They are incremental change, assuming simplicity, open and honest communication, local adaptation, traveling light, use multiple models, and limit contract models (Ambler, 2001). The 12 practices to XP are the following: the planning process, small releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour work week, on-site customer, and coding standard (Beck, 1999).

XP should be used with small to medium-sized teams, which is typically no more than 12 people (2000, September). The appropriate time to implement XP is when projects have dynamic requirements or when projects are considered high risk (Wells, 2001, June 4d). There are at least three benefits to using XP. They are easier to learn, reduced overhead, and greater productivity. Some of the disadvantages to XP are the limitations to large size teams, inability to scale, and requiring highly-skilled and highly-motivated individuals.

Nevertheless, XP is a mish-mash of tricks and techniques, plus methods, models and management tools along with practices. But the hallmark to XP is that it is more people-oriented than process-oriented. Success with most agile methodologies, such as XP, require close, hands-on management aimed at smoothing the process and removing impediments (Constantine, 2001).

XP is very innovative and different. It is similar to a jig saw puzzle. There are many small pieces where individually they make no sense, but when combined together provide a complete picture (Wells, 2001, June 4c). This is a significant change from most traditional methods. Therefore, it appears XP may succeed where other methodologies have not.

References

- (2000, September). *UML World: Modelers Unite in Learning*. Retrieved May 12, 2001, from the World Wide Web:
<http://www.sdmagazine.com/articles/2000/0009/0009b/0009b.htm>.
- Ambler, S. W. (2001, April). A Closer Look at Extreme Modeling. *Software Development*. Retrieved July 2, 2001, from the World Wide Web:
<http://www.sdmagazine.com/articles/2001/0104/0104m/0104m.htm>.
- Beck, K. (1999, October). Embracing Change with Extreme Programming. *IEEE Computer*. Retrieved July 14, 2001, from the World Wide Web:
<http://dlib.computer.org/co/books/co1999/pdf/rx070.pdf>.
- Booch, G. (2001, March). Developing the Future. *Communications of the ACM*. Retrieved July 4, 2001, from the World Wide Web: <http://portal.acm.org>.
- Constantine, L. (2001, June). Methodological Agility. *Software Development*. Retrieved July 2, 2001, from the World Wide Web:
<http://www.sdmagazine.com/articles/2001/0106/0106f/0106f.htm>.
- Cutter Consortium (2000, February). *Extreme Programming*. Retrieved July 14, 2001, from the World Wide Web: <http://www.cutter.com/ead/ead0002.html>.
- Fowler, M. (2000, December). Put Your Process on a Diet. *Software Development*. Retrieved July 2, 2001, from the World Wide Web:
<http://www.sdmagazine.com/articles/2000/0012/0012a/0012a.htm>.
- Fowler, M. (2001, April). Is Design Dead? *Software Development*. Retrieved June 17, 2001, from the World Wide Web:
<http://www.sdmagazine.com/articles/2001/0104/0104a/0104a.htm>.
- Fraser, S., Beck, K., Cunningham, W., Crocker, R., Fowler, M., Rising, L., & Williams, L. (2000). *Hacker or Hero? Extreme Programming Today*. Paper presented at the Conference on Object-Oriented Programming, Systems, Languages, and Applications on Addendum to the 2000 proceedings, Minneapolis, Minnesota.
- Hendrickson, C. (2000, December 5). When is it not XP? *XProgramming.com*. Retrieved June 17, 2001, from the World Wide Web:
<http://www.xprogramming.com/xpmag/NotXP.htm>.
- Jeffries, R. (2001, June). What is eXtreme Programming. *XProgramming.org*. Retrieved June 17, 2001, from the World Wide Web: <http://www.xprogramming.org>.

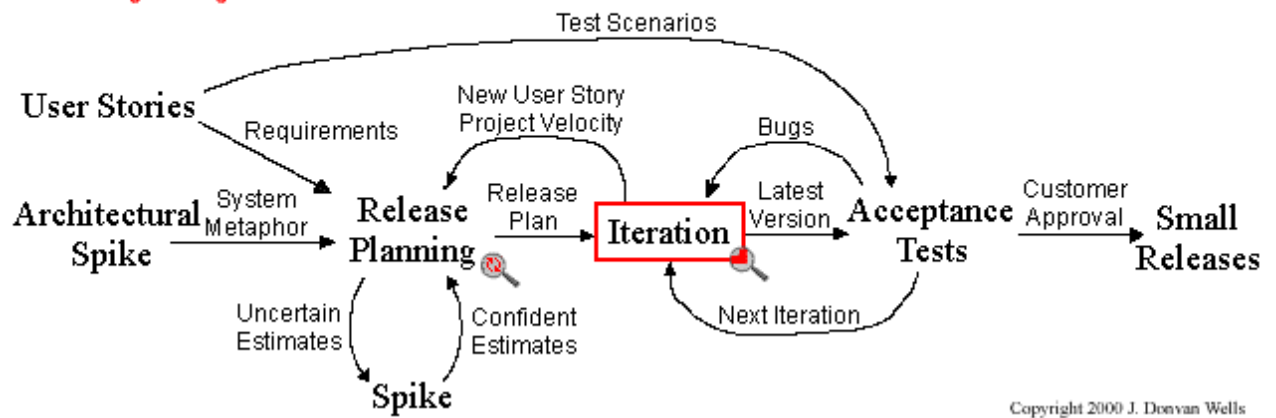
- Wells, J. D. (2001, June 4a). Do We Need Yet Another Methodology? *ExtremeProgramming.org*. Retrieved June 17, 2001, from the World Wide Web: <http://www.extremeprogramming.org/another.html>.
- Wells, J. D. (2001, June 4b). Extreme Programming Project. *ExtremeProgramming.org*. Retrieved June 17, 2001, from the World Wide Web: <http://www.extremeprogramming.org/project.html>.
- Wells, J. D. (2001, June 4c). What is Extreme Programming? *ExtremeProgramming.org*. Retrieved June 17, 2001, from the World Wide Web: <http://www.extremeprogramming.org/what.html>.
- Wells, J. D. (2001, June 4d). When Should Extreme Programming Be Used? *ExtremeProgramming.org*. Retrieved June 17, 2001, from the World Wide Web: <http://www.extremeprogramming.org/when.html>.
- Williams, L., & Kessler, R. R. (2000, May). All I Really Need to Know About Pair Programming I Learned in Kindergarten. *Communications of the ACM*. Retrieved July 4, 2001, from the World Wide Web: <http://portal.acm.org>.

Appendix A

XP Process



Extreme Programming Project



Copyright 2000 J. Donovan Wells

Figure 1. XP Process (Wells, 2001, June 4b)