Extreme Programming

Akhil Rasheed
CSE 'C'

1. Introduction

Extreme Programming is a software development model which primarily focuses on the software quality improvements and responsiveness to the changing customer requirements. It is a type of agile software development model. It advocates frequent releases in short development cycles. These releases are intended to improve the productivity and enhance the quality of the software by introducing certain measures that focus on initiating certain checkpoints at which the customer requirements can be
adopted and met.

Extreme Programming focuses on light-weight processes. The main phases involved in the cycle of XP are- Planning, Design, Coding, Testing. As it is an iterative model, the system is developed by dividing the overall project into small functions. The cycle of development from the design to the test phase is performed for one function. After executing for one function and debugging it properly, the developers then shift to the next.

XP is based on rapid release cycles and continuous communication between the developers and the stakeholders; that is, the customers. It strongly relies on oral communication, frequent testing, code review and designing. Communication is regarded as an important criteria in XP. The communication should frequently happen among the concerned parties like the developers, customers, and managers

2. History and similarities.

XP was created by Kent Beck during his work on Chrysler Comprehensive Compensation system (C3) payroll project. Kent Beck became C3 project leader in March 1996 and refined the development methodology used in the project and wrote a book on the Extreme Programming methodology
in 1999. XP is regarded to be better than the traditional waterfall model. It is increasingly adopted by companies worldwide for software development.

Since in the waterfall model, the development process is completed on a single project and the requirements have to be known beforehand, there is no scope of changing the requirements once the project development starts. So the waterfall model is not flexible to the changing requirement needs. On the other hand, XP focuses on iterations and allows for changing requirements even after the initial planning has been completed. Also, the waterfall model does not require the participation of customers whereas XP focuses on customer participation and satisfaction and regards it as a primary goal during software
development process. So, XP is a preferred model in the software development these days.

XP is a framework of the agile software development model, and it primarily aims at producing higher quality software. Agile software development advocates adaptive planning, evolutionary development, continuous improvement and encourages rapid and flexible response to changes. It focuses on customer satisfaction, simplicity and continuous attention from the developers as well as the customers. XP is the framework of the agile software development and hence all these practices are implemented in XP.

3. Xtreme Programming Practices

The core of XP is the interconnected set of software development practices listed below. While it is possible to do these practices in isolation, many teams have found some practices reinforce the others and should be done in conjunction to fully eliminate the risks you often face in software development.

The XP Practices have changed a bit since they were initially introduced.The original twelve practices are listed below.

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-hour week
- On-site Customer
- Coding Standard

Below are the descriptions of the practices as described in the second edition of Extreme Programming Explained Embrace Change. These descriptions include refinements based on experiences of many who practice extreme programming and reflect a more practical set of practices.

Sit Together

Since communication is one of the five values of XP, and most people agree that face to face conversation is the best form of communication, have your team sit together in the same space without barriers to communication, such as cubicle walls.

Whole Team

A cross functional group of people with the necessary roles for a product form a single team. This means people with a need as well as all the people who play some part in satisfying that need all work together on a daily basis to accomplish a specific outcome.

Informative Workspace

Set up your team space to facilitate face to face communication, allow people to have some privacy when they need it, and make the work of the team transparent to each other and to interested parties outside the team. Utilize Information Radiators to actively communicate up-to-date information.

Energized Work

You are most effective at software development and all knowledge work when you are focused and free from distractions.

Energized work means taking steps to make sure you are able physically and mentally to get into a focused state. This means do not overwork yourself (or let others overwork you). It also means stay healthy, and show respect to your teammates to keep them healthy.

Pair Programming

Pair Programming means all production software is developed by two people sitting at the same machine. The idea behind this practice is that two brains and four eyes are better than one brain and two eyes. You effectively get a continuous code review and quicker response to nagging problems that may stop one person dead in their tracks.

Teams that have used pair programming have found that it improves quality and does not actually take twice as long because they are able to work through problems quicker and they stay more focused on the task at hand, thereby creating less code to accomplish the same thing.

Stories

Describe what the product should do in terms meaningful to customers and users. These stories are intended to be short descriptions of things users want to be able to do with the product that can be used for planning and serve as reminders for more detailed conversations when the team gets around to realizing that particular story.

Weekly Cycle

The Weekly Cycle is synonymous to an iteration. In the case of XP, the team meets on the first day of the week to reflect on progress to date, the customer picks the stories they would like delivered in that week, and the team determines how they will approach those stories. The goal by the end of the week is to have running tested features that realize the selected stories.

The intent behind the time boxed delivery period is to produce something to show to the customer for feedback.

Quarterly Cycle

The Quarterly Cycle is synonymous to a release. The purpose is to keep the detailed work of each weekly cycle in context of the overall project. The customer lays out the overall plan for the team in terms of features desired within a particular quarter, which provides the team with a view of the forest while they are in the trees, and it also helps the customer to work with other stakeholders who may need some idea of when features will be available.

Remember when planning a quarterly cycle the information about any particular story is at a relatively high level, the order of story delivery within a Quarterly Cycle can change and the stories included in the Quarterly Cycle may change. If you are able to revisit the plan on a weekly basis following each weekly cycle, you can keep everyone informed as soon as those changes become apparent to keep surprises to a minimum.

Slack

The idea behind slack in XP terms is to add some low priority tasks or stories in your weekly and quarterly cycles that can be dropped if the team gets behind on more important tasks or stories. Put another way, account for the inherent variability in estimates to make sure you leave yourself a good chance of meeting your forecasts.

Continuous Integration

Continuous Integration is a practice where code changes are immediately tested when they are added to a larger code base. The benefit of this practice is you can catch and fix integration issues sooner.

Most teams dread the code integration step because of the inherent discovery of conflicts and issues that result. Most teams take the approach “If it hurts, avoid it as long as possible”.

Practitioners of XP suggest “if it hurts, do it more often”.

The reasoning behind that approach is that if you experience problems every time you integrate code, and it takes a while to find where the problems are, perhaps you should integrate more often so that if there are problems, they are much easier to find because there are fewer changes incorporated into the build.

This practice requires some extra discipline and is highly dependent on Ten Minute Build and Test First Development.

Test-First Programming

Instead of following the normal path of:

develop code -> write tests -> run tests

The practice of Test-First Programming follows the path of:

Write failing automated test -> Run failing test -> develop code to make test pass -> run test -> repeat

As with Continuous Integration, Test-First Programming reduces the feedback cycle for developers to identify and resolve issues, thereby decreasing the number of bugs that get introduced into production.

Case Study A:

A team of four developers was acquired to implement a system (code-named for eXpert) for managing the research data obtained over years at a Finnish research institute. A metaphor that better describes the intended purpose of the system is a large “virtual file cabinet”, which holds a

large number of organized rich (i.e., annotated) links to physical or web-based resources. The system is an web-based client-server solution.

Item	Description
Language	Java (JRE 1.4.1), JSP (2.0)
Database	MySQL (Core 4.0.9 NT, Java connector 2.0.14)
Development Environment	Eclipse (2.1)
SCM	CVS (1.11.2); integrated to Eclipse.
Documents	MS Office XP
Web Server	Apache Tomcat (4.1)

The four developers were 5-6th year students with 1-4 years of industrial experience in software development. Team members were well-versed in the java programming language and object-oriented analysis and design
approaches. Two weeks prior to project launch the team performed a self-study by studying two basic books on XP [i.e., 4, 8]. A two day hands-on training on XP practices, the development environment and software configuration management tools was organized to ensure that the team has a basic understanding on XP issues and the technical environment. Table 1 shows the details of the technical environment used for the development of eXpert system.

Thus, this study focuses on a development team that is novice to extreme programming practices. The team worked in a co-located development
environment. The customer (i.e., a representative from the research institute) shared the same office space with the development team. The office space and workstations were organized according to the suggestions made in the XP literature to support efficient teamwork. Unused bookshelves, as an example, were removed in order to
have a maximum amount of empty wall space for user stories and task breakdowns, architecture description, etc.

Results

As indicated earlier, due to a lack of empirical data from XP process, this paper concentrates on reporting concrete metrics data obtained from the first two releases. Thus, while qualitative data has been collected on XP practices and the process, they will be reported elsewhere.

The concrete metrics involve effort usage for each task and XP practice with a precision of 1 minute, development work size using automated counters for Java and JSP, development time defects (including type, severity), post-release defects (found by 17 allocated system testers) and the number of enhancement suggestions made by testers. Work commit size was drawn from the CVS tool. The quality of the data obtained was systematically monitored by the project manager, dedicated metrics responsible and the on-site customer.

Collected data	Release 1	Release 2	Total
Total work effort (h)	195	190	385
Task allocated actual hours	136 (70%)	95 (50%)	231 (60%)
# LOCs implemented in a release	1821	2386	4207
Team productivity (loc/hour)	13.39	25.12	18.21
# User stories implemented	5	8	13
# User stories postponed for next release	0	1	1
User story effort (actual, median, h)	10.1	8.3	9.2
User story effort (actual, max, h)	63.1	26.9	63.1
# Tasks defined	10	30	40
Task effort (actual, median, h)	11.7	2.9	4.1
Task effort (actual, max, h)	32.3	8.8	32.3
# post-release defects	4	5	9
Post-release defects/KLoc	2.19	2.10	2.14
# post-release enhancement suggestions	11	12	23
Pair programming (%)	92	73	82
Customer involvement (%)	5	6	5.3

Table 2. Concrete data from two-week releases

Table 2 shows the data obtained from the first two releases. The total column shows the cumulative data from the two releases. Total work effort dedicated to project work remained constant in the first two releases. However, the direct hours dedicated to tasks was reduced from 70% to 50%. None-task allocated work was the effort spent to planning-game, data collection, project meetings, brainstorming, coaching and not anticipated extra pre-release testing. The team productivity did however improve from 13.39 to 25.12 loc/hour. The first release contained tasks not
related to user functionality such as finalizing the technical set up of the development environment. This explains partly the increase in team productivity.

The team implemented five user stories in the first release and eight in the second. The median user story size (hours) for the first release was 10.1 hours and 8.3 for the second. The maximum size of a user story in the first
release was 63.1 hours. In the second release, the maximum size was reduced to 26.9 hours. While only 10 tasks were defined for the first release, the second release contained already 30 tasks. Similarly, the median size of a task was reduced from 11.7 to 2.9 hours, and the maximum size of a task was reduced from 32.3 to 8.8 hours. 17 testers were allocated for a brief (i.e., max 45min) and intensive (i.e., testing was to be performed within four hours from system release) user functionality test. Testers discovered four defects after the first release and five defects after the second release.

The defect density for the first release was thus 2.19 defects/KLoc and 2.10 for the second. The defect density was found to be satisfactory giving an indication of the overall product quality. Testers
also proposed 11 enhancements (i.e., new or improved functionality) after the first release and 12 following the second release.

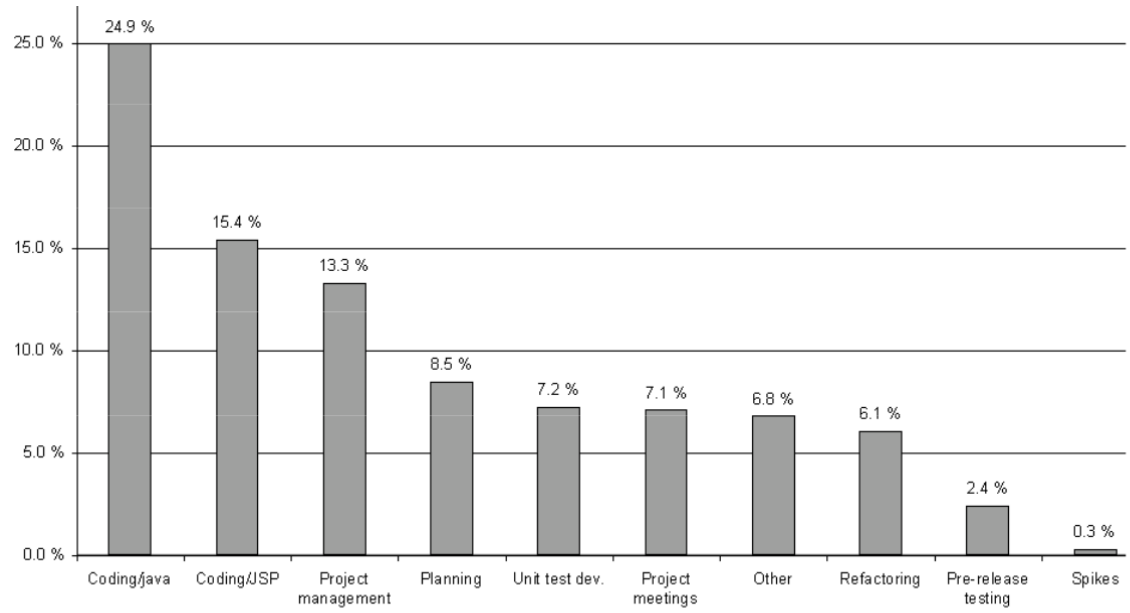

Figure 1. Effort distribution (%)

Pair programming was extensively practiced in the development of the first release (92%) but was reduced to 70% in the second release. While the customer shared the same office with the development team and thus was
present close to 100% of the total time, the actual customer involvement was only 5% in the first release and 6% in the second release. Figure 1 displays the overall effort distribution for the first

two development cycles. Data shows that in this project roughly 9% is required for planning the release contents. Project management activities, which include data analysis, monitoring the progress of the project and the development of project plan require 11% of the total effort. Coding in terms of unit test development, production code, development spikes and refactoring take
54% of the total effort. Daily project meetings took 7% and the overhead caused by data collection was only 2%. All the design documentation including architectural description is displayed in the walls of the development room.

The effort spent on an “design” phase is only 2%. This is actually the time used for architecture design. The simple design practice involved in the pair programming coding was not separately tracked. The principal amount of effort was spent directly on tasks, 68% in the first release and 54% on the second release. User stories and tasks were documented and displayed in the walls as well. The system documentation for the maintenance purposes will be produced in the last two releases when the system architecture and the user functionality has stabilized enough and is not subject to constant changes.

Case Study B

The course was held in the summer term of 2000. Partici- pants were CS graduate students who needed to take a prac- tical training course as part of their degree requirements. Most of the students had experience with team work, but only one with pair programming. Only one student had de-
veloped moving graphical displays of the sort needed for the project. twelve students began the course; one dropped after the first three weeks because of lack of time; the rest completed.

In the first three weeks, students solved small program- ming exercises to familiarize themselves with the programming environment and to learn X P practices. The exercises introducedjUnit (the testing framework used throughout the course), pair programming, the test practices of X P (write
test cases before coding, execute them automatically with jUnit), and refactoring. The remaining eight weeks were devoted to a project on visual traffic simulation. The course language was Java. All students had experience with Java from their early undergraduate courses. Table 1 summarizes the course details.

Table 1. Summary of course details.

Number of participants	12
Qualification	graduate students
Course language	Java
Testing tool	jUnit
Version control	CVS
First exercise	Matrix class
Duration	240 - 660 min
Second exercise	Visualization tool (VT)
Duration	240 - 630 min
Third exercise	Extension of VT with HTML or text output
Duration	120 - 420 min
Project	Traffic simulation
Duration	8 weeks, about 40h per team total

The first exercise covered the implementation of a matrix class. The second exercise was a small visualization tool combined with a scanner to read data from a text file.

The third exercise extended the visualization tool with out- put in HTML or a text format. The remaining eight weeks were spent with project work. The project was the imple- mentation of a traffic simulation with cars, traffic lights, and trains. The students met weekly for working together in six pairs. They paired voluntarily with different partners for each exercise and the project. The loose coupling of pairs within a team is a common practice in XP. After the first three programming problems, one student left the course and one of the instructors (Matthias) filled in, in order to have six full pairs. However, Matthias tried not give his team an unfair advantage by providing hints that others wouldn't get.

For the project, the students were divided up into two teams with three pairs each. Each team was tasked with the full project, including the graphical representation, the functionality for the moving cars, the right-of-way rules, and the traffic light control. Figure 1 shows a snapshot of the traffic simulation with seven cars, a crossroad, and a railway. The railway was meant as an extension, but never implemented due to lack of time

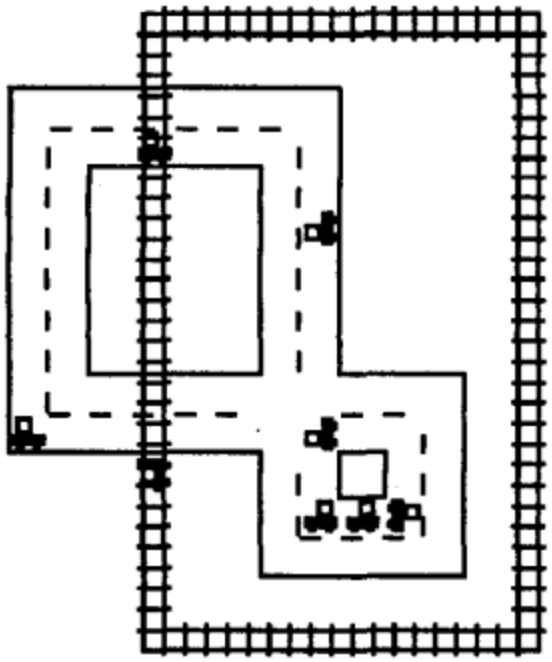

Figure 1. Snapshot of the traffic simulation.

The instructor plaid the role of the customer, telling the students which features to implemented next. In XP par-lance, the instructor simulated the planning game. Students filled out a total of five questionnaires. The first
was handed out at the beginning and asked about education and practical experience. The others were filled out after each exercise and at the end of the project. These forms contained questions about XP practices, what the participants learnt, where they had difficulties, and suggestions for improvements. The tables 2-4 at the end of this study sum up the results.

Experiences

Pair Programming

The students were asked to pair with different partners of their own choosing for each exercise and the project. Students had no problems adapting to pair work. All but one team enjoyed this style of collaboration and the resulting problem solving process. The exception was a team in which one member wanted to design, while the other wanted to get the task done. Neither student enjoyed the experience, which resulted in having an over-design on the one hand and a bad design on the other. These two students were later placed into different project teams.

Insights about the productivity of pairs vs. individuals cannot be expected from a case study such as this. However, the authors have some observations about the division of work in pairs. The social rules about pair programming by Williams [8] do not say enough about how work should be structured. Students noted that it is a waste of time to watch a partner during rote coding, such as writing a bunch of get and set methods. One student suggested that the partner not typing should perform a constant review of the code, while others suggested different kinds of work such as reading the Java API documentation. One student preferred to meet only for implementing a complicated algorithm while others liked to pair permanently.

Most of the pairs used the following two-display tech- nique: On one display, they implemented while the other showed the relevant Java documentation. When a pair ran into difficulties, one team member consulted the documentation while the other studied the code. When asked if a single display would have been enough, 75% declined. Al-
though this approach was used in a Java environment, it is an indication to soften the strict rule of pair programming at a single terminal.

Another aspect of pair programming is learning from each other. The students confirmed that they learned something from their partners. The topics ranged from matrix algebra to editor commands. 43% of the participants stated that they had learned something from pair programming, but this effect declined with the duration of the course. In summary, pair programming still suffers from some waste of time and from an unclear division of work. It is also unclear whether the main benefit of pair programming, higher quality, could be achieved by a less personnel-
intensive approach such as pair inspections

Iteration Planning

Given a set of new features to implement, XP's guide- lines say to develop the simplest possible solution. The rationale is that software changes are cheap and no time should be wasted developing unneeded generality.
This recipe proved problematic. All our students con- tinuously planned for the future. In order to get students to focus on just the next set of features, the instructors had to publicly abandon one speculative feature after another.

For example, at the beginning of the project, a street editor was mentioned that would simplify the construction of the traffic scenarios. The students also heard about trains and level crossings. The instructors made it clear that these were speculative extensions that would probably not be im-
plemented. But once these ideas were out, our students would continuously think ahead to accommodate trains, level crossings, and the street editor. At the end of the project, it became

clear that they had always planned for these features. Yet the features were never added, because time ran out. We wonder what would have happened if we
had mentioned overtaking cars.

Testing

There are two aspects of testing in XP: first, to write the test cases before coding, and second, to make them execute automatically for regression testing. Writing test cases before coding is a substitute for specification. What exactly do the methods do, what parameters do they take, and what are the (testable) results? This approach was new to most students. Only 25% applied it to their development prior to the course, see table 4.Most students adopted it naturally right from start, some needed our intervention in the early stage of the project, but one pair adopted it not until they needed the test cases for restructuring the code. This pair developed the Java class for the crossroads without having written the accompanying test class.

The instructor urged them to write the tests. At the next meeting, they had the test cases written, but they had also changed both the underlying representation and the interface. They cleaned up the code while establishing the necessary test cases at the same time. The result was that there was no running program for two weeks. Had they had the test cases, they could have first concentrated on restructuring and then evolving the interface, while always having a running system for the rest of the team.

The pairs building the graphical display were unable to provide fully automated test cases. They wrote the test cases (traffic scenarios) and watched the display for errors. To automate these tests would have required storing bitmaps and comparing them, which seemed too much effort under the circumstances. At the end of the course, the students were convinced of the benefits of writing test cases prior to coding. It is
the testing approach that the students considered the best practice in the final review of the course. 87% stated that the execution of the test cases strengthened their confidence in the code and all of them were planning to try out this practice after the course, see table 4. All students saw jUnit as a suitable test framework

Refactoring

The students never got to a point where they needed to refactor. One team had a complete design that did not need to be improved, the other team had a situation (the crossroads example in the preceding section) where one team sort of refactored a prototype, but without the benefit of test cases. Lack of refactoring may be caused by a combination of several factors: the small size of the project and doing full rather than minimal designs.

Scaling
Within iteration planning, team members break down the requirements from the Planning Game into small pieces. Later, these pieces are processed in pairs. The division of the requirements requires that the team members agree on a common terminology. Otherwise, team members lose a lot of time. The communication problem is one limitation to the team size of X P because larger teams face much more communication overhead than smaller ones, this limitation is a bit relaxed if team members have worked together before.

Conclusions and Open Questions

1. Pair programming is adopted easily and an enjoyable way to code. However, it is unclear what type of work not to do in pairs and how best to structure pair in- teraction. Additional research is needed to compare the effectiveness of pair programming with reviewing Techniques.

2. Design in small increments (“design with blinders”) is difficult. Holistic design behavior may be difficult to abandon and more research is needed to test whether this is actually a good idea. If one wants developers to design in small increments, at least one pair member should be trained in it.

3. Writing test cases before coding is not easily adopted and is sometimes impractical. Is it really essential to write the test cases first and then the code, or is it possible to do it the other way around?

4. Due to the communication overhead, XP as is does not scale well. It is definitely meant for small teams (6-8 members).

5. X P requires coaching until it is fully adopted.

Presentation Report guidelines

1. Presentation reports should be typed only on one side of the paper with 1.5-line spacing on A4size sheet (210 x 297 mm). The margins should be: Left - 1.25", Right - 1", Top and Bottom - 0.75".
2. The font style for the different sections:
    - **Font type**: Times New Roman
    - **Font Size**: chapter headings **16,** main heading **14,** sub-heading **12,** body of the text **12.**
3. **One copy of the report is enough.**

*Presentation report should be soft-bound.

JSS MAHAVIDYAPEETHA

JSS Science and Technology University

"Title of the Presentation"

BY

Student Name <USN>

Submitted to

Shalini K C
Assistant professor,
Department of Computer Science and Engineering
JSS STU, Mysuru-06

2021-2022

Department of Computer Science and Engineering