

# Extreme Programming: A Case Study in Software Engineering Courses

**Ghazy Assassa, PhD**  
Dept of Computer Science,  
College of Computer  
and Information Sciences,  
King Saud University  
ghazy@ccis.ksu.edu.sa

**Hassan Mathkour, PhD**  
Dept of Computer Science,  
College of Computer  
and Information Sciences,  
King Saud University  
mathkour@ccis.ksu.edu.sa

**Hmood Al Dossari**  
Dept of Computer Science,  
College of Computer  
and Information Sciences,  
King Saud University  
hzs24@yahoo.com

## Abstract

*Extreme Programming (XP) is considered as the most popular of the various flavours of “agile” software methodologies. In this paper, the authors explore the practices of XP in the context of software engineering education. The practices of XP are examined and as they influence students skills related to software engineering courses. The paper discusses an experiment that was carried out to apply XP in a software engineering course project in the department of Computer Science at the College of Computer and Information Sciences, King Saud University. The project lasted eleven weeks and included three releases. Results and feed back from students are reported and recommendations are highlighted.*

## Keywords

Extreme Programming, XP, Software Engineering, Students Projects, Educational Environment.

## 1. Introduction

Extreme Programming "XP" is considered as the most famous and prominent agile methodology. Since the development of XP methodology by Kent Beck [1], researchers in universities and managers in software organizations tried to evaluate the success of this new model. Researchers concluded that using XP in educational domain have many benefits [2]. In this paper we discuss an experiment for adopting XP methodology for software development of eleven-week course projects at a senior level software engineering course. The discussion will include reviewing related work, experiment setup, results and discussions of the experiment, and our conclusion and recommendations.

## 2. Extreme Programming

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

XP includes a set of values, principles and practices for rapidly developing high-quality software. XP is extreme in the sense that it takes

many well-known software development "best practices" to their logical extreme [3].

Communication is the first XP value. For example, XP takes the “best practice” of “good communication with the customer” to an extreme by recommending that the customer works in the same room as the programmers, interacting with the team as necessary. Besides, Each morning programmers participate in a short stand up meeting; this enhances the effectiveness of communication as participants become more close. XP was developed with four core values in mind: Communication, Simplicity, Feedback and Courage. From these values twelve practices were derived [4, 5].

### 2.1 XP Practices

The twelve practices of XP are discussed below:

1. **The Planning Game:** the business and development teams get together to decide on what features of the required system will be of maximum value to the business. The techniques for gathering requirements in XP are a radical departure from that of more traditional software methodologies. First, customer requirements are written in natural language, informal "User Story" cards, similar to use cases [6]. These cards are

never formalized, no relationships or dependencies between the cards are identified. Software developers place time estimates and customers assign priorities to each card. Together, the developers and the customers play the "Planning Game" in which the customer chooses those User Stories that comprise the most important content for a short, incremental deliverable of about one month. Each short implementation increment is accepted and tried by the customer. Then, the remaining User Stories are re-examined for possible requirement and/or priority changes and the Planning Game is re-played for the next implementation increment.

2. **Small Releases:** a simple system containing a useful set of features is put into production early and updated frequently in short cycles. XP heightens the pace of spiral development by having short releases of 3-4 weeks. At the end of each release, the customer reviews the interim product, identify defects, and adjust future requirements.
3. **Metaphor:** each project has a "system of names" and description which helps to guide the development process and communication between all parties. XP believes that each application should have conceptual integrity based on a simple metaphor, which explains the essence of how the system works. For example, one large XP project was a payroll system for Chrysler. The metaphor for this project was that the payroll system was like an assembly line where hour parts were converted to dollar parts, all parts were assembled and a paycheck was produced.
4. **Simple Design:** the simplest design is always used to build the application as long as it meets the current business requirements. Do not worry about future requirements as requirements change with time anyway. Refactoring practice (see below) will ensure that the design is of a high standard. XP strives for supremely simple designs. They stress that programmers should not try to predict future needs and to produce more complicated design accordingly. Developers should follow the simple design practice and "Do the simplest thing that could possibly work."

5. **Testing:** XP follows a "Test-first" approach, that is before new features are added, tests are written to verify the software. The software is then developed to pass these tests. Software developed with XP is validated at all times. Two types of testing is carried out, unit and functional testing.

5.1. **Unit Testing.** Extensive, automated white box test cases are written before production code is produced. These automated tests are added to the code base. Before a programmer can integrate their code into the code base, they must pass 100 % of their own test cases and 100% of every test that was ever written on the code base. This ensures that the new code implements the new functionality without breaking anyone else's code.

5.2. **Functional Test.** Traditionally, project management techniques have been based on a developer's own assessment of how much of their task has been completed. Alternately, XP promotes the use of functional test case tracking for calculating project completeness. XP terms this assessment "Project Velocity." Functional test cases are based on customer scenarios. When a functional test case is successfully passed, it can be considered that a specified functionality has been implemented properly. Project completeness is based on the percentage of functional test cases that have been passed. Team members can unequivocally compute this measure.

5.3. **Automated testing:** It isn't enough to write tests: you have to run them. Unit tests are all collected together, and every time any programmer releases any code to the repository (pairs typically release twice a day or more), every single one of the programmer tests must run correctly. One hundred percent, all the time! Developer have interest to use appropriate automated testing frameworks, e.g. JUnit and NUnit, to control and simplify the task of repeated testing and continuous integration. This means that programmers get immediate feedback on how they're doing. Additionally, these tests provide invaluable support as the software design is improved.

6. **Refactoring:** Refactoring is the process of improving the code's structure while

preserving (not improving) its function [7]. XP advocates refactoring code continuously and explicitly. This is a technique for improving the design of an existing code-base. Its essence is applying a series of small behavior preserving transformations that improve the structure of the code. By doing them in small steps you reduce the risk of introducing errors [8].

7. **Pair Programming:** programmers using XP are paired and write all production code using a single machine per pair. This helps the code to be constantly reviewed while being written. Pair Programming has proved to produce high quality code with little or no decrease in productivity [9].
8. **Collective Code Ownership:** all the code belongs to every member of the team, no single member of the team owns a piece of code and anyone can make changes to the codebase at any time. This encourages everyone to contribute new ideas to all segments of the project.
9. **Continuous Integration:** software systems are built and integrated several times a day; at the very least all changes are integrated into the main codebase, on an integration machine, at least once a day. As a result, there are many product builds each day. Each build is tested using the associated test cases.
10. **40-Hour Week:** programmers in an XP project normally adhere to a 40 hour working week in order to maintain productivity and avoid burn out. It was found that during crunch periods when overtime is worked, the artifacts that are produced are poor.
11. **On-site Customer:** one or more customers who will use the system being built are allocated to the development team. The customer helps to guide the development and is empowered to prioritize, state requirements and answer any questions the developers may have. This ensures that there is effective communication with the customer and as a result less documentation will be required.
12. **Coding Standards:** everyone on an XP project use the same Coding Standards which makes it easy to work in pairs and share ownership of all code. One should not

be able to tell who worked on what code in an XP project. An agreed upon coding standard should be defined and followed.

## 2.2 Other XP Strategies

**Incremental change:** Big changes can be risky and prone to failure so only small changes are recommended.

**Small initial project investment:** XP projects are started with a small number of developers and then built up, as more developers are required.

**Stand up Meetings:** meetings are held physically standing up to keep the meeting brief, at the same time each day. The purpose of this is for members to report problems but no solutions are proposed. The developers then leave the meeting and ponder on the solutions.

**Tracking progress:** a designated team member is responsible for tracking the progress of other team members.

**Minimal documentation:** documentation is kept to the barest minimum.

**Teach Strategies:** to enable staff to learn, e.g. how much testing should be done.

**Experiment:** experiments are carried out to reduce or eradicate the risk of incorrect technological decisions.

## 3. Background work

Many studies have been carried out on the use of XP in educational environment [11, 12, 13, 14, 15]. The study of [11, 13] concluded that it is not advisable to teach and practice entire traditional and agile methodologies in one semester course, and suggested a hybrid process that includes both agile and traditional practices if the students had only one software engineering course.

The results of [12, 14] showed that the planning game and the 40-hour week practices were the most successfully established XP-practices; in particular the continuous integration and pair programming practices were not successfully followed.

The study of [13] for ten-week software engineering course indicated that the XP teams were unable to adopt many XP practices, with

weak customer engagement, a lack of collective code ownership, and batch integration.

The work of [14] showed that it was possible to use the XP methodology successfully for final year capstone projects, but that students need to be actively coached in the skills necessary to make XP.

According to [15], the key factors producing successful XP outcome for a project course were: skilled tutors were able to act as Extreme Programming coaches for the teams; timetabling and physical facilities that strongly support group working; and external clients who were willing and able to engage with the Extreme Programming processes.

#### **4. The Experiment**

A description of the experiment setup is given hereafter including student's previous exposure to XP, students background, duration of the experiment, ATM project scope, group formation, tools, and customer and XP mentor activities.

##### **Student's previous exposure to XP**

Students participating in this experiment had no previous exposure to XP. Therefore, they had been introduced to the practices of XP in the first two weeks of study. As for instructors, the experiment is the first one in using XP for developing course projects.

##### **Students background**

The experiment was conducted on senior-level class of eight students enrolled in CSC540 Software Engineering course offered by the department of Computer Science at the college of Computer and Information Sciences, King Saud University. Students' background related to the experiment includes passing two programming courses in C language, CSC 112 and CSC 113, and a first course in Software Engineering, CSC342, as well as two database courses CSC380 Fundamentals of Database Systems and CSC383 Advanced Database Management Systems. Students also had completed their BSc graduation projects, CSC 496, CSC 497, using traditional methodologies, mostly waterfall approach.

##### **Duration of the experiment**

The experiment was conducted within a timeframe of a single semester, fall 2004. Within CSC540 software engineering course, students are usually required to present assignments,

discuss papers, and develop a course project. In addition, they have to pass midterm and final exams. Typically, students would have some 10-12 weeks conducting the course project. In our experiment, the project lasted eleven weeks including two weeks at the beginning for project and methodology definitions and one week at the semester end for presentation. It should be noted that the current experiment may suffer from the fact that the time allowed for the experiment is relatively short (eight weeks) thus allowing only for small scale projects.

##### **ATM project scope**

Since the aim of this study was not to deliver a software product to a customer, a simple but rather real problem had to be given to the students participating in the experiment. In the context of simple problem, it has been pointed out [11] that the disadvantage of a simple problem, such as a roman number converter, is that somehow it is not real enough - the consequence is that students loose confidence after they return from their course because some of the subtle aspects of XP have not been explored. Taking this into consideration, we selected a real-world problem, namely, simulation for an ATM system for which student were asked to develop the corresponding system. The system scope was limited to providing the following basic services to the user: ATM access 'login', change PIN, cash withdrawal, cash deposit, transfer to another account, balance inquiry, and mini statement.

**Group formation:** The 8 students participating in the experiment were asked to form 4 groups each comprising 2 students. During the first two weeks of study, the XP methodology was rapidly explained and a comparison with the waterfall model was highlighted. Also the ATM project was succinctly presented. Three groups of students selected to develop the ATM project using the XP methodology while the fourth group selected the WF approach.

##### **Tools**

Because of time constraints to complete the project within the semester time frame, we preferred to allow students to select the tools they have more experience with. The three XP teams developed their projects using the different tools. Team 1: ASP.net and SQL Server, team 2: Delphi and Access, and team 3: Java and SQL Server. The WF team worked with C# and Access database.

### Customer and XP mentor Activities

In carrying out this experiment, two functions had to be defined, namely the customer and XP mentor functions. The customer and, to some extent, the XP mentor functions were assumed by the first two authors while the third author partially covered the activities of XP mentor.

## 5. Results and Discussions

This section presents the results and discusses the following issues: partial adoption of XP, on-site customer, planning game, Shodan and IBM surveys, customer and communication level and response to changes in requirements.

### Partial adoption of XP “sub-practice”

In the current experiment, only the XP practices pertinent to small scale projects were focused on “sub-practice”. The sub-practices included those contributing to rapid feedback and learning process, namely, planning game, pair programming, collective code ownership, unit testing, simple design, refactoring, and use of coding standards.

### On-Site Customer

Due to real-world constraints, there was no real customer that could be present 100% of the time on site; as indicated above, customer and XP mentor activities were simulated by the authors. On the average, students had three hours per week contact with the simulated customer and 1-2 hours with the XP mentor. To enhance communication between development teams and 'simulated' customer, we established a web site where we posted the stories and suggested project releases and deadlines; we also exchanged email messages extensively. In addition, in order to avoid delayed decisions for questions requiring an immediate response, we allowed students to contact us by phone, mobiles, and SMS messages.

### Planning Game

The ATM requirements were discussed with the XP groups and many proposals for releases were evaluated. Based on the time frame of the project and the availability of students, students agreed upon three releases. After detailed discussion, the students identified seven stories to be included in the three releases. Release 1 included three stories, ATM access 'login', change PIN, and cash withdrawal. Release 2 included two stories, cash deposit, and transfer to another account. The last two stories, balance inquiry and mini statement, were included in release 3. It

should be mentioned that the planning game practice was implemented with full success with all XP groups. In the class room, we discussed with students a template for the customer story and students 'developers' added some features they judged necessary for the understanding of the requirements. We, as 'customers' provided the students with the full description of the seven stories.

## 6. Results of Shodan and IBM Surveys

To assess the adherence of students to XP practices and to get a feed back on students' acceptance of the XP methodology, two surveys were used, namely, Shodan Adherence survey and IBM survey. Details on both surveys are given hereafter.

### 6.1 Shodan Adherence Survey

Shodan Adherence Survey [16] is designed to assess how far the students followed XP practices. It is a subjective means of gathering adherence information from team members. The term 'Shodan' means 'black belt'. The survey is composed of 15 questions on the extent to which each individual on a team uses XP practices (testing has been split to three categories and stand up meetings were added to the practices). A survey respondent self-reports the extent to which he or she used the practice, on a scale from 0% (never) to 100% (always). An overall score for the survey is computed via a weighted average of each response.

$$\text{Adherence to XP practice} = \frac{\sum (\text{practice score} * \text{practice weight})}{10 \text{ over 15 practices}}$$

### Shodan survey questions

For each question on table 1, the students were asked to use the following scale:

10: Fanatic (100%)	4: Common (40%)
9: Always (90%)	3: Sometimes (30%)
8: Regular (80%)	2: Rarely (20%)
7: Often (70%)	1: Hardly ever (10%)
6: Usually (60%)	0: Disagree with using this practice
5: Half 'n Half (50%)	

XP Practice	Weight	Description / Question to be answered
Automated Unit Tests	6%	You run automated unit test (such as JUnit,) each time you make a change. What % of your changes are tested with automated unit tests before they are checked in?
Customer Acceptance Tests	3%	Make sure both the developers and the customer know what they want <i>What % of your requirements have corresponding acceptance tests specified by the customer?</i>
Test First Design	3%	Write test cases, then the code. The test case is the spec. <i>What % of your code line items were written AFTER an automated test was developed for the corresponding scenario?</i>
Pair Programming	12%	Two people, one computer. One thinks strategy, the other tactics. <i>What percentage of your work (design, analysis, coding) was done in pairs?</i>
Refactoring	10%	Rewrite code that 'smells bad' to improve future maintenance and flexibility without changing its behavior. <i>What % of the time do you stop to cleanup code that has already been implemented without changing functionality?</i>
Release Planning	6%	Customer and developers trade items in and out of the plan based on current priorities and costs. Adaptation is favored over following a plan. <i>Do you allow for changes in release plans/requirements after each iteration based on customer feedback and current implementation?</i>
Customer Access	6%	On Site Customer is best, you can use chat, etc. to quickly verify requirements and get feedback. <i>What % of the time do you get quick interaction with your customers when needed?</i>
Short Releases	6%	You have frequent smaller releases instead of larger, less frequent ones. This lets the customer see how it's going and lets you get feedback. <i>How close are you to having releases that are about 3 months with interim iterations of a couple weeks?</i>
Stand Up Meeting	6%	The team takes 10 minutes each day to review what needs to be done each day and assigns user tasks to team members.
Continuous Integration	10%	Code is checked in quickly to avoid code syncup / integration hassles. <i>How often do you syncup and check in your code on average? (10 = 3 times a day, 8 = once a day)</i>
Coding Standards	5%	Do you have and adhere to team coding standards? Besides brace placement, this may include things like logging and performance idioms. <i>How often do you follow your team standards?</i>
Collective Ownership	8%	You can change anyone's code and they can change yours. You don't get stuck when the expert is busy on vacation. People know many parts of the system. <i>How often do people change code they did not originally write?</i>
40-Hours Sustainable Pace	5%	People need to be effective over the long haul. <i>How well do you pace yourself?</i> Example Scores: 10 - I maintain a sustainable pace and the same high rate of output. 5 - I work longer than what I consider a sustainable pace, but still produce at a high rate and feel only a little burnt out. 2 - I work beyond a sustainable pace and feel burnt out. My code isn't at its usual high quality.
Simple Design	8%	Keep it simple at first; do the simplest thing that could possibly work. You don't follow the philosophy of "I'll include this because the customer might possible need it later" even though the feature isn't in the requirements. Also, you do not spend a lot of time on design documents. <i>How often do you succeed in 'Keeping it Simple'?</i>
Metaphor	6%	A single, overarching metaphor is used to describe the system. It is used by developers to help communicate ideas and to explain concepts to customers. <i>How often do you feel this is true of the systems you develop?</i>

**Table 1: Shodan survey questions**

### **Shodan Survey Results**

Results of the survey showed that the overall Adherence to XP practice was on the border of being acceptable in the range of 55 - 60%.

At release level, the survey showed that there is an improvement, of the order 5%, from release 1 to release 3. This was attributed to the more experience student got as they move from the first release to the last release.

### **IBM Survey Results**

The XP teams were asked to respond on the IBM survey question: What do you think about Extreme Programming? The results are as follows: No for the question 'I tried it and I hate it', No for the question 'It's a bad idea, it'll never work', Yes (33%) for the question 'It's a good idea, , but it'll never work', and finally Yes (66%) for the question 'I've tried it and love it'.

### **Customer and Communication level**

We tried to compensate for the lack of on-site customer by various means of communication including direct contact within the lecture time, mentoring, web communication, phone, mobile, and SMS messages. In our opinion, we succeeded in compensating for the lack of a real on-site customer. However, the true of things is that we are instructors attempting to replace the real customer and we are not present 100% of time on-site with developers. Under this assumption, we asked the developers to produce our requirements as instructors. The real customer will not be such knowledgeable about requirements, and underlying XP practices. Feedback from the XP teams about the communication level with the 'simulated' customer showed an overall high acceptance. In conclusion, we think that this practice is reasonably met in the current experiment.

### **Response to changes in requirements**

Two weeks before the system delivery (during XP release 3), we introduced an additional small functionality in the requirements of the last story and we asked XP and WF teams to incorporate the extra functionality in the system. Unfortunately, WF team was unable to respond to the changes in requirements while the three XP teams were successful at adding the extra functionality.

### **Feedback**

Feedback from students showed that they have practiced some of the XP practices before even

knowing about the existence of such methodology; in particular pair programming that they used to practice within their graduation projects. Survey showed that they liked and are willing to work XP in their future developments. We quote here some of the comments we received from students:

- "In general, we are very comfortable with XP methodology and we will use it in future projects, if nothing else is specified by user's non-functional requirements. The main reason is that it is code-oriented and leads to develop software without the upfront activity of detailed analysis, design and heavy documentation that are traditionally needed as for example in the waterfall methodology",
- "In general, I like the XP and I would like to take another project that uses the XP model, in particular I liked the planning game and small size of documentation",
- "The use of the XP in this project was good one, The project was small project, easy to manage, and easy to implement",
- and finally " The XP has some practices that we enjoy doing like game planning, small releases, incremental integration and sustainable pace".

## **7. Conclusion and Recommendations**

The objective of the current study was to apply XP using three releases and two-to-three weeks development iterations. Three XP teams each composed of pair senior students worked their Software Engineering course projects in a time frame of eleven weeks. We acted as the customer and XP mentor while the students represented the development team. The aim of the project was to produce an ATM system. The results of the experiment and the many discussions we had with students suggest that the XP is very suitable for such small-scale projects. We guess the main challenge in XP is the on-site customer and work the planning game with inexperienced customers.

It was observed that XP teams produced the required product with full functionality and less work. In addition, the response to changes in requirements is more successful when applying XP methodology as compared to the Waterfall methodology. An important observation is that the adherence to XP practices increased with the

progress of releases. Because of the small size of the sample, additional experiments should be carried out in order to generalize the outcome. We have plans to carry out additional experiments at junior level students.

The researchers concluded that using XP in educational domain have many benefits like a strong commitment to the project development on the part of both students and supervisors, less skilled students showed more progress than probably would have been the case using a traditional methodology, and XP teams produced better software than those using traditional model.

We would like to highlight the following recommendations:

- Extreme Programming practices should be incorporated more solidly into earlier courses of the curriculum.
- Additional experiments are needed to provide some insight in the use of XP in the educational domain and to establish some useful XP metrics.
- There is a noticeable lack of experiments on the use of XP for large scale projects; future software experiments should target filling this gap.
- Software cost estimation models are needed to reflect the effect XP practices on cost; in particular pair programming and small releases practices.

## References

1. Beck, Kent, Extreme Programming Explained. ISBN 0-201-61641-6, Addison-Wesley, 2000.
2. Williams, L., Kessler, R. "Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom", Journal on Computer Science Education, March 2001.
3. Brewer, John and Design, Jera "Extreme Programming FAQ", 2001, Available: <http://www.jera.com/techinfo/xpfaq.html>.
4. Williams, L., and Upchurch, R. "Extreme Programming For Software Engineering Education?", 31st ASEE/IEEE Frontiers in Education Conference, October, 2001.
5. Akpata, E and Riha, K "Can Extreme Programming be used by a Lone Programmer?", Systems Integration, 2004.
6. Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach, ACM Press, 1992.
7. Jeffries, R., Anderson, A., and Hendrickson, C., "Extreme Programming Installed", Reading, Massachusetts, Addison-Wesley, 2001.
8. Fowler, Martin, "Refactoring: Improving the Design of Existing Code", 2003, Details available at: <http://www.martinfowler.com/books.html/refactoring> and <http://www.xp2003.org/conference/TutorialsDescr.html#T22>
9. ManLui, Kim and Chan, Keith C.C. , "When does a pair outperforms two individuals?", in Proc. Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering,, Genova, Italy, 2003. Available: <http://www.xp2003.org/slides/15.pdf>
10. Cockburn, Alistair., Williams, Laurie. "The Costs and Benefits of Pair Programming." Available URL :<http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF>
11. Srikanth, H., William, L., Wiebe, E. Miller, C. Balik, S. "On Pair Rotation in the Computer Science Course." , IEEE 17th Conference on Software Engineering Education and Training(CSEET'04), 2004.
12. Shukla, A. and Williams L. D. "Adapting Extreme Programming For A Core Software Engineering Course." Conference on Software Engineering and Training (CSEE 2002), Covington, KY USA, 2002.
13. Lappo, P., "No pain, no XP: Observations on teaching and mentoring extreme programming to university students", Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, Universit'e de Cagliari and Free University of Bolzano-Bozen, 2002, pp. 35-38.
14. Noll, J. and Atkinson, D. C. "Comparing extreme programming to traditional development for student projects: A case study" , Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 4th International Conference, XP 2003', Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 372-374.
15. Keefe, K., Dick, M. "Using Extreme Programming in a Capstone Project.", Australian Computer Society, 2004.
16. Shodan Input Metric Survey, Available URL: <http://C2.com/cgi/wiki?ShodanInputMetric>.