Extreme Programming in the Liberal Arts Classroom: A Progress Report

Richard A. Brown
Professor of Computer Science
rab@stolaf.edu

Aubrey F. Barnard
Student
barnard@stolaf.edu

Matthew T. Bills
Student
bills@stolaf.edu

Michael W. Bongard
Student
bongard@stolaf.edu

Aaron F. Etshokin
Student
etshokin@stolaf.edu

Theodore M. Johnson
Student
johnsotm@stolaf.edu

Michael R. Zahniser
Student
zahniser@stolaf.edu

St. Olaf College

Abstract

Extreme Programming (XP), a "lightweight" or agile process for software development, adheres to twelve practices in a disciplined, non-traditional methodology. This process was applied in a 13-person team project for a course at St. Olaf, a liberal arts college, including both advanced and less experienced students. Observations and recommendations based on this experience are reported.

Introduction

Kent Beck's Extreme Programming (XP) (Beck, 2000) turns the traditional "waterfall model" of software development on its head. Instead of beginning by carefully defining the features desired for a software system, then thoroughly specifying design, then coding, then testing, all before each extensive software release, XP calls for building tests first, *then* implementing, *then* thinking about (minimal) design, in a frequent succession of completed end-to-end systems.

A program-first, design-later approach to software design, so contrary to messages we have diligently instilled in our students, sounds heretical and foolhardy, like a step backward. Yet Beck claims great effectiveness for the XP strategy, including unusually high productivity and software products with improved, not worsened, flexibility to adapt to changing requirements. A closer look shows that XP is far from a naive "program-first-without-thinking" approach. XP may break traditional rules, but the new methodology in fact incorporates a high degree of discipline and thoughtful reflection on the software development process. Beck argues that the existence of powerful modern software tools, such as integrated development environments, version control software, and languages such as Java that support rapid software development, enable a tightly cooperating software team to respond to changes during software development with such speed and fluidity that the team can safely skip highly structured, painstaking up-front planning. An open-minded consideration of reasoned challenges to standard approaches is appropriate for the study of software development in a liberal arts computer science program.

We applied XP methodology in a semester-long course-based programming project to build a practical networked application, employing a client-server architecture with a graphics user interface and database "back-end," during Fall 2003. The thirteen students in the class formed a single team; each had prior experience with waterfall-model software development in C++, but additional experience diverged greatly among individual students. Implementation was in Java and SQL, with communication structured using XML—new languages to most of these students. The instructor took on the end-user "client" role.

We will review the elements of XP, then describe our project more fully and make some observations.

The discipline of Extreme Programming

XP is an example of a *lightweight* or *agile process* in software design, along with such methodologies as ("Scrum", 2004). In reaction to the burden of extensive processes, comprehensive documentation, and á priori planning of traditional software development methodologies, agile software processes place greater value on interactions

among individuals, documentation on an as-needed basis that focuses on overall design rationale, collaborative relationships with customers, and flexibility to respond to changes ("Agile Manifesto", 2001), (Martin, 2002, Chapter 1). Beck argues that the cost of making significant changes in requirements during software development is no longer prohibitive, given modern software development tools. Hence, a software team may freely invest in production stages of the software development process, rather than focusing on thorough pre-production planning, as long as the team observes crucial safeguards such as good communication, constant feedback, and a focus on simplicity (Beck, 2000, Chapter 5).

Creating software with XP involves a series of short (e.g., two week) *iterations*, conducted using *XP practices* that include the following (Beck, 2000, Chapter 10):

- *the planning game*, a strategy for planning the work for an iteration (see below);
- *small releases*, i.e., iterations, each representing a complete end-to-end system in some sense;
- the use of *metaphor* to describe the architecture of a system;
- *simple design*, defined as a design that satisfies all current tests, has no duplication of logic in the implementation, includes every intention important to the programmers, and has the fewest possible classes and methods—a design minimally sufficient for the current needs rather than one that speculates on future needs;
- unit *testing*, or writing automated tests of all functionality before any implementation, creating an ever-expanding criterion for the evolving system to satisfy;
- *refactoring*, a policy under which any team member may rewrite existing code, no matter who originally wrote it, whenever rewriting leads to simpler design (e.g., elimination of duplicate logic);
- *pair programming*, with two people working at one computer, one person using the keyboard and mouse and thinking about expedient implementation, and the other person thinking more strategically, with frequent swapping of paired individuals (e.g., twice or three times per 8-hour day in a professional setting);
- *collective ownership*, in which everyone takes responsibility for an entire system, and any pair may make improvements in any part of the system that further that pair's task;
- *continuous integration*, or adding new code to the system on a daily basis or more frequently, as soon as all unit tests for the entire system succeed with that new code added;

- a *"40-hour week,"*, meant as a limitation of overtime in a professional situation (never a second week of overtime in a row);
- an *on-site customer*, a future end-user of the desired system, who sits on the team; and
- *coding standards*, voluntarily agreed upon in advance by the entire team, emphasizing communication and facilitating practices such as swapping pairs and refactoring.

The planning game seeks to determine the scope of the next iteration rapidly by combining user priorities and technical estimates. The goal of the game is to maximize the value of the software produced by the team, deducting the cost of development and risk incurred during development, where *risk* is the estimated potential for inaccuracy in cost estimates. The strategy calls for investing as little as possible to produce the most valuable functionality (as assessed by the end-user representative) in the least time, while reducing risk, then to iterate. During the planning game, the end-user representative on a team describes desired features as *stories* in terms of the project's metaphors, and the software development team devises tasks to accomplish the stories in an iteration. The user decides scope, priority, requirements for an iteration in terms of stories, and sets iteration completion dates. The software developers estimate the time or effort required for tasks or stories, identify technical choices forced by the stories, decide how tasks will be implemented, and determine detailed scheduling. The moves of the planning game represent a structured negotiation between the user and the software developers, with the user focusing on functionality represented in stories and developers forecasting time requirements and risk.

Benefits claimed for XP include improved productivity, a favorable experience for programmers, and higher end-user satisfaction.

Although pair programming has enjoyed increasing visibility in the CS education community since 2000 (Williams and Kessler, 2002) and (Hanks, 2004), and sessions on teaching with XP have begun to appear at conferences (Eckstein, 2003), (Hazzan et al., 2004), attempts at applying the complete collection of XP practices have appeared rarely if ever in projects for liberal arts CS courses.

The ACE project

St. Olaf's course CS 284, *Client-Server Applications (CSA)*, seeks to explore the computer science behind modern web-based applications by surveying basic concepts and technologies and by applying them in a team project. CSA counts as a core course toward St. Olaf's newly developed CS major curriculum (Brown, 2003). Rather than focusing on higher-level web software architecture as in (Lee, 2003), this course

surveys Java-level network programming, client and server technologies, graphics user interfaces (GUIs), and databases with SQL. All students have taken a Scheme-then-C++ introductory sequence, including a course in software design featuring a life-cycle based team software project in C++ using waterfall-model methodology, UML language for planning object design, thorough "plan-first" documentation in multiple phases using locally developed XML document templates, and a modest multi-stage ethical analysis.

In the Fall, 2003 first offering of CSA, the majority of students had no prior experience with Java or SQL programming, nor with XML programming models or document type definitions. One senior in the class spent a month full time in Summer 2003 planning for the CSA course project together with the professor. Of course, this plan was discarded when we actually began the course project using XP methodology. However, having a student on the team who had already thought in depth about how to build such software provided a counterpart of the experience that professional programmers would likely bring to such a project. Several other seniors brought extensive student project backgrounds to the work, from other course work, undergraduate research projects, and internship experiences. Thus, the class fell naturally into two categories of students, an advanced group and a group with little if any CS experience beyond the course's prerequisites.

The goal of the software project was to support online management of requirements for students toward their major and/or concentration programs. In Mathematics, Statistics, and Computer Science, St. Olaf has traditionally negotiated the requirements for a major or concentration on an individual contract basis, referring to general guidelines for those programs. This system sought the following goals: to assist students in creating, submitting, and retrieving specific contract proposals that satisfy the guideline criteria; to aid program directors in viewing, responding to, and approving those student contracts; and to support aggregate analysis of collected contracts. The CSA students dubbed this system *ACE, the Academic Contract Explorer*. The instructor presented some initial implementation decisions: Java language for client and server, using an applet for the client; SQL database for storage; and XML for expressing both individual contracts and *credentials*, i.e., representations of guidelines for a major or concentration program.

The members of the class entered the project team in stages.

- Two advanced students with strong backgrounds who already knew Java well immediately began working on an authentication strategy that would allow users to enter their standard campus passwords without fear of compromise in this student-developed software system. Meanwhile, the remaining students studied Java, principles of GUI construction, network programming, and elements of concurrency through a series of labs.

- The remaining advanced students worked at double pace on labs, then joined the authentication pair to form the *first-iteration team*. That iteration produced a rudimentary functioning end-to-end infrastructure system for the project.

  A brief introduction to XP practices and philosophy for the entire class took place before this first iteration, but only the first-iteration team carried out the initial planning game. The instructor played a dual role of general manager/end-user representative: he communicated the strategies of XP methodology and structured planning-game meetings by identifying specific goals and time limits within each meeting; and he provided the stories and assigned their priorities.

- All students joined together for two subsequent iterations, once everyone had seen presentations of all the Java-based material. The instructor continued serving as both manager and end-user representative.

Material on databases, SQL, and XML was presented in parallel with the second and third iterations. Students received introductions to the Eclipse IDE, CVS revision control system, and other technical systems during in-class laboratory sessions near the beginning of the second iteration.

The class met as a whole for two 85-minute periods per week; two or three additional key planning sessions took place during a weekly campus meeting time. In addition, students arranged pair-programming times on their own, and met as a group on many Saturdays.

To document contributions to the project, a web-based time logging system was developed, with drop-down menus for easy contemporaneous entry of time durations and identifying information. The log page supported itemization of activities (e.g., particular stages of planning, pair programming, communication such as e-mail), indication of whom one was pairing or teaming with, and entry of descriptive notes.

Messages sent to a course e-mail alias, heavily used for project purposes, were archived for retrieval by anyone in the class.

Observations

Our experience leads us to the following observations.

Quality of educational experience.

Our strongest and best-prepared student team members exhibited remarkable leadership and initiative in the project. They claimed personal ownership of the process,

researching XP methodology on their own, independently initiating and conducting the regular Saturday meeting times, making impromptu presentations at meetings, and carrying out XP practices to the best of their abilities. The instructor set the stage in various ways, such as enunciating the nature of this XP experiment, orchestrating team meetings that took place during class time, and encouraging leaders (individually) to make a good faith effort to include all team members. But the team as a whole accepted the challenge and pursued it, and the leaders, who comprised about half of the class, brought an outstanding and cooperative attitude to the effort, demonstrating long patience with their less experienced teammates. For example, knowing that they could accomplish tasks quickly and effectively among themselves, the leaders nevertheless sought to pair with newcomers in the second iteration, and even decided to give formal responsibility for completing third-iteration tasks to those newcomers.

However, less experienced students found it difficult to join the ongoing project effort. A few of those with minimal prior backgrounds for the course showed initiative and made significant contributions. But many of the less experienced students did not feel prepared for the task, and although they expressed initial enthusiasm about the project, commitment waned for some as the project continued. By the end of the term, the disparity of commitment between the leaders and those who remained behind on their Java assignments had become a frustration and disappointment among some of the leaders, who privately and respectfully expressed their concerns to the instructor in person and in an end-of-course questionnaire.

An analysis of student log entries and archived e-mail messages quantifies the perceived differences in effort among team members. In Figure 1, students are identified according to their absolute ranking in number of hours reported via electronically submitted time logs, e.g., student S3 reported the third largest number of actual hours. An asterisk denotes the graduating seniors, who also constituted the first-iteration team. Columns in the table represent actual amounts reported through two months of logs and archived project e-mails, multiplied by 1.0 for members of the first iteration team and by 1.5 for those who joined the project for only the second and third iterations. These weight factors roughly compensate for the additional time that first-iteration team members could contribute to the project, since lengths of iterations were approximately equal, excluding breaks. These figures actually underestimate the full contributions of seniors, since logging was unavailable for some of the first iteration; the figures also exclude early authentication work carried out by students S1 and S2. Columns of Figure 1 indicate the number of hours reported in logs as being spent on the project (outside of class), the number of logs submitted, the number of times a students received or gave a citation of collaboration in a log, and the number of project e-mails sent. These entities represent simple software *metrics*, or quantitative indicators of performance. While no single metric provides a sufficient measure of the work, collectively they present an aggregate image of contribution toward the team project.

student	logged hours	log count	citations received	citations given	emails archived
S1*	117.2	27	21	28	18
S2*	87.8	32	17	23	17
S4	61.3	33	22.5	24	21
S3*	57.1	28	10	21	12
S5*	39	17	30	11	4
S6*	36.7	11	5	17	4
S8	32.3	19.5	30	7.5	6
S9	29.9	16.5	9	16.5	10.5
S7*	25.4	9	12	15	4
S10	21.1	13.5	7.5	7.5	9
S11	11.5	9	4.5	4.5	3
S12	0	0	10.5	0	1.5
S13	0	0	6	0	3
mean	39.9	16.6	14.2	13.5	8.7
mean of seniors	60.5	20.7	15.8	19.2	9.8
mean of non-seniors:					
all	22.3	13.1	12.9	8.6	7.7
excluding S12,S13	31.2	18.3	14.7	12	9.9

Figure 1: Weighted analysis of student contributions to the ACE project over a two-month period. Asterisks (*) indicate seniors (first-iteration team). Weights roughly compensate for those who joined the project at the second iteration.

A comparison of means shows that senior students surpassed non-seniors on average in every category derived from logs submissions, even after weighting and underestimation of work done by the first-iteration team. The greatest contrast appears in number of hours on task reported: even after omitting the two students who made no log submissions, the average senior logged about twice as many hours as the average non-senior after weighting.

We believe this dichotomy between more and less experienced students can be addressed by providing more support for learning Java. Two 85-minute class meetings per week was simply too little contact time for all the needs of the course: presenting Java and background concepts to advanced students, and later to younger students; working with one and sometimes two groups of advanced students; and providing laboratory support for the Java exercises. Laboratory support suffered as a result. Note that these same Java lab materials had proven effective for a comparable audience in prior courses which included adequate support time for learning. Also, non-seniors who took advantage of instructor office hours and later, optional catch-up lab sessions became full participants in the project: student S4 became a project leader by any measure, even without the weighting factor; and other non-seniors performed well in several categories. We conclude that a larger proportion of less experienced students will likely become stronger project contributors in future offerings of the course, provided that they receive sufficient support in learning the basics.

XP practices

The XP practices fit liberal arts values in many ways. The entire methodology relies heavily on good communication skills, for example, when playing the planning game and when pair programming. The high degree of teamwork required and the many opportunities for cooperative leadership develop qualities we associate with a liberal arts education. And XP's alternative approach to software development demands a high degree of open-mindedness for those accustomed to other, more standard software methodologies.

We found it best to adhere to the entire set of XP practices, as far as possible. For example, we tried phasing in unit testing after our first iteration, believing this might make a reasonable transition to the methodology in the early going. But then, we lacked unit testing's objective, automated criterion for determining when an iteration was complete. Unit-tests add a measure of confidence in one's product for programmers at all levels of ability. (We readily assert that no amount of testing can prove a program correct; our students are steeped in this truth before arriving at the CSA course.) Furthermore, retrofitting unit tests for the first iteration while simultaneously creating unit tests for the second iteration was a complex and difficult task. Next time, we will definitely apply the unit testing philosophy from the outset.

Pure XP assumes that the entire team can work in the same room throughout the work week, an impractical assumption for us. Indeed, even scheduling time to work in pairs presented significant challenges for our students, with their other courses and activities. We attempted to compensate for this shortcoming with electronic communication. Many of the messages sent to the class e-mail list simply stated that the sender would be in the lab at a specified time and invited team members to pair with that person. One advanced student even developed a small networked application to support team communication needs, as a personal spare-time project. Even with electronic support, the basic problem of effectively finding meeting times for pairs and larger groups remains as a major challenge.

Classes incorporating XP projects will benefit from all available scheduled class meeting time. Our next offering of CSA will include three 85-minute meetings per week instead of two.

Course management.

In XP, project managers direct work largely by serving as a coach for team members, intervening when necessary, and by tracking metrics of a team's progress and posting the results (Beck, 2000, Chapter 12). The manager chooses which metrics to track and post, thus drawing a team's attention to specific issues. This image of coaching, intervening, and tracking project management largely describes the role that the instructor took in our project, except that little metric feedback was posted (only reviews of progress toward task deadlines during class meetings).

Tracking task deadlines served to move the project along, but other metrics could have helped shape and improve the team's process in specific ways. For example, sharing statistics (without names) on hours logged per person with the team as a whole might well have encouraged under-performing team members to spend more hours on the project; publishing the number of e-mail requests for pair partners sent to the course e-mail alias might have promoted such communications; later, determining how many pair-partner requests were satisfied and looking for trends among such figures might have encouraged pair activities. Although our student leaders didn't need such prods, we believe that metric-based feedback would inform less-experienced students about project expectations and where to put their energies, particularly if we also notify individual students of their own ratings relative to each metric scale.

We will probably use advanced students as a "first-iteration team" again, anticipating that an alternate-year core course will always include students with a broad range of experience. However, we must proceed more intentionally with integrating the remaining, less experienced students into the second-iteration effort. We tried consciously pairing advanced students with younger students, and we tried assigning task responsibilities to the younger students, but neither of these approaches alone solved

our integration problem. Preparing the less experienced students better in the Java language and using progress metrics as a motivational strategy will undoubtedly help. In addition, we will probably add some explicit training in effective pair programming to the course (Williams and Kessler, 2002). We hope that measures such as these will address this non-trivial problem of integrating new people into the ongoing team project.

Finally, we plan to use a student teaching assistant in our next offering of CSA, ideally a veteran of the prior offering. For example, this person could: assist students with Java laboratories during class meetings while the instructor works with any advanced team(s); assist with technical setup for the project; and serve as an after-hours resource for the team as a whole. If the teaching assistant already knows XP methodology, he/she could convey those principles and practices as well.

Conclusion

To summarize our observations:

- An XP project gives advanced students an opportunity to develop their leadership skills and broaden their experience in software development methodologies.
- Less experienced students naturally require additional direction and training in the skills needed for an XP project.
- XP projects demand communication skills and cooperative work, highly appropriate for the liberal arts.
- Compromises on individual XP practices appear to undermine the results of that methodology.
- XP projects in college courses struggle for meeting time, prompting extended course-meeting schedules and electronic communication for arranging meetings.
- As an XP project manager, an instructor coaches, intervenes only if necessary, tracks metrics assessing project progress, and posts analysis of metrics to motivate the team.
- For classes with multiple levels of student background, better prepared students may serve as an "advance team" within the XP framework. In that case, integration of remaining students into the ongoing project requires careful attention; explicit training in effective pair programming may help.

Acknowledgments

Besides the co-authors, the students in the Fall 2003 CSA course were Robert L. Crawford, Noah J. Dove, Rylan Z. Gibbens, Ari J. Gronning, Joel J. Johnson, Joel S. Landsteiner, and Justin E. Von Stroh. Although not everyone could be listed as a co-author, the students listed here made important contributions while advancing their applied and project skills in an experimental course that turned out to have higher expectations than anticipated.

Bob Breid, Systems Administrator at St. Olaf's Department of Information and Instructional Technologies, helped with the authentication system.

References

"Agile Manifesto" (2001). Manifesto for agile software development. Retrieved March 11, 2004, from `http://agilemanifesto.org/`.

Beck, K. (2000). *Extreme programming explained: Embrace change*. Addison-Wesley.

Brown, R. A. (2003). Requirements for a CS major. Retrieved March 11, 2004, from `http://www.stolaf.edu/depts/cs/academics/major.html`. St. Olaf's CS major was introduced in 2002.

Eckstein, J. (2003). The first extreme educational symposium: A special forum for teachers in extreme programming and agile processes. Retrieved March 11, 2004, from `http://www.xp2003.org/EduSymCfP.html`.

Hanks, B. (2004). Pair programming bibliography. Retrieved March 11, 2004, from `http://www.cse.ucsc.edu/ brianh/PairProgramBib.html`.

Hazzan, O., Bergin, J., Caristi, J., Dubinsky, Y., and Williams, L. (2004). Teaching software development methods: The case of extreme programming. In *Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education*, pages 448–449. Panel.

Lee, A. H. (2003). A manageable web software architecture: Searching for simplicity. In *Proceedings of the Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education*, pages 229–233.

Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Process*. Prentice Hall.

"Scrum" (2004). Scrum development process. Retrieved March 11, 2004, from `http://controlchaos.org/`.

Williams, L. and Kessler, R. (2002). *Pair Programming Illuminated*. Addison-Wesley.