Bringing Extreme Programming to the Classroom

Owen Astrachan
Duke University
Department of Computer Science
Durham, NC 27708 USA
ola@cs.duke.edu

Robert C. Duvall
Duke University
Department of Computer Science
Durham, NC 27708 USA
rcd@cs.duke.edu

Eugene Wallingford
University of Northern Iowa
Computer Science Department
Cedar Falls, IA 50614 USA
wallingf@cs.uni.edu

Abstract
In this paper we discuss several features of XP we have used in developing curricula and courses at Duke University and the University of Northern Iowa. We also discuss those practices of XP that we teach as part of the design and implementation process we want students to practice as they develop programming expertise and experience. In theory the academic study of programming and software development should be able to embrace all of XP. In practice, however, we find the demands of students and professors to be different from professional and industrial software developers so that while we embrace the philosophy and change of XP we have not (yet) adopted its principles completely.

1 INTRODUCTION

Extreme Programming (XP) [3] and other light or agile [1, 6] methodologies have gained a signficant foothold in industry, but have not generated the same heat (or light) in academic settings.

Significant interest in pair-programming in an academic setting, and a resulting interest in XP, has been fostered by the work of Laurie Williams [16, 15]. However, the general tenets of XP are less known, and the engineering background of many academic computer science programs facilitate adoption of process-oriented methodologies such as the *Personal Software Process* (PSP) [11] even early in the curriculum [10]. However, we have had preliminary success in adopting and adapting principles of XP (and other agile methodologies) in classroom teaching and in the methods we teach and instill in our students. Although academic requirements, goals, and methods differ those in industry, we have found that many aspects of XP can be incorporated into the design and implementation of a university-level computer science and programming curriculum.

What's Extreme about XP?

As explained in [3] XP takes good practices of professional software development to extreme levels. XP introduces a planned and coherent methodology, but without becoming overly dictatorial or secretarial. The four values of XP are given as *Communication*, *Simplicity*, *Feedback*, and *Courage*.

As we explain in this paper, these values form the foundation of our approach. Hence we think we're following the spirit of XP although we're certainly not following every XP practice (e.g., testing, pair programming, planning game and so on) [9].

From these core XP values, five principles are given as fundamental to XP. As we explain in this paper, our approach uses each of these principles: *Rapid feedback*, *Assume simplicity*, *Incremental change*, *Embracing change*, and *Quality work*.

Ten "less central principles" from [3] are given from which we concentrate on the following four: *Teach learning*, *Concrete experiments*, *Open, honest communication*, and *Local adaptation*.

For example, as instructors we often face a tension in developing good (often small) example programs for students. The tension arises because we want to develop simple focused examples that teach a specific concept, yet we also want to show our students programs that are masterpieces of good design, i.e., programs that are fully generic, robust, and exemplify the best practices of object oriented design and programming.

XP advocates that we design the simplest possible solution that works well for the current set of requirements, not those we imagine will exist in the future. This helps relieve some of the tension of designing overly generic or optimized programs when creating example code.

Additionally, with this new mindset, we can now add new features to that example and show students how the code changes. In other words, we can give the students a peek into the process of creating programs. When we call this process refactoring, we can discuss a program's design in more concrete terms [7].

In this paper we report on three aspects of XP we have employed very successfully. We have used XP in our introduc-

tory programming courses for majors, in advanced courses on object oriented software design, and in programming courses for non-majors.

- Pair (teacher/class) programming as part of lecture.
- Small releases from student groups.
- Refactoring to understand programming and design patterns.

2 OUR CLIENTS

Embracing change within a university setting is different than industry because our clients are different. In fact, when using XP we are meeting the demands of two different client/customer groups.

1. We strive to develop programmers who appreciate simplicity and elegance, who love building software, and who understand the contributions of computer science to software design. The process we mentor and teach must resonate with our students and scale from introductory programming courses to more advanced courses in software architecture.
2. We want our curriculum, assignments, and materials to be adopted and adapted by educators all over the world. Our materials must be simple, elegant, and support adaptation and refactoring to meet local demands. The process and materials must resonate with educators at a different level than the resonance we hope for with students.

Our student clients take several courses each semester. They devote 20%-40% of their time to a course on programming depending on demands of other courses and the interest level we can maintain in our courses. We assume that students live and breathe solely for our courses, but we are also not surprised that other professors in other departments hold similar views about their courses. Thus it is difficult for groups of students to meet frequently or for extended periods of time outside of class.

The structure of the work students do in our courses varies from traditional lecture, to structured (time-constrained) labs, to unstructured group and individual activity in completing assignments. Our XP-based material typically takes more time to prepare and requires us to use XP practices to produce it.

3 LECTURING USING PAIR PROGRAMMING

We use a didactic form of pair programming in our large lecture courses. The instructor is the driver while the class as a whole (from 40 to 180 students) works together as the second member of the pair-programming team we call the *navigator*. A typical scenario, used from beginning to advanced courses, is outlined in the following. First we explain the process from a student view, then we elaborate on the process from a faculty developer perspective.

Student View

1. A problem is posed that requires a programming solution. The problem and its solution are intended to illustrate features of a programming language, elements of software design, principles of computer science and to engage students in the process.
2. A preliminary, partially-developed program is given as the first step to the solution. Students read the program and ask questions about it as a program and potential solution.
3. The instructor displays the program on a projection screen visible to the class (each student has a written copy) and adds functionality with input from the class. This involves writing code, writing tests, and running and debugging the program. The instructor drives the process, but the class contributes with ideas, code, and questions.
4. The final program is added to the day's website for reflection, completeness, and for those students unable to attend class. Both the initial and final programs are part of the materials available to students.

We have tried a variety of standard active learning techniques in this form of pair programming: calling on random students to contribute, breaking the class into small groups to provide solutions, and making pre- and post-classwork questions based on the programming problem.

Educator View

The instructor who drives a programming problem and solution must develop a complete solution beforehand and then refactor the solution into one that meets the needs of the instructional process as described in the previous section. This process may take more preparation time and require more responsibility from the instructor during class time than a traditional lecture.

1. The instructor finds a problem and develops a complete program/solution to the problem. The solution is developed using XP, but the goal is a simple, working program which isn't always the right instructional tool.
2. The program must be refactored until it is simple enough to be understood by the student client while still achieving the intended didactic goals. This simplification process is often easier in introductory courses because the programs are smaller. In some cases, especially in more advanced courses, a problem and its solution must often be completely reworked or thrown out when they're too complex to be used in a one-hour lecture.

3. Parts of the program are then removed so the program can be completed as part of an instructor/class pair-programming exercise. The instructor has an idea of what the solution could be, but the solution developed during class isn't always the one the instructor pared away. Instructors must be comfortable with accepting and using student input, going down knowingly false trails for instructional purposes.

4 SMALL RELEASES MEAN BIG PROGRESS

Students in our non-major's programming course as well as our first- and second-year major's courses work in groups on large (to them) projects lasting up to three weeks. We allow three weeks to complete a project not because a project requires that amount of time, but to allow students time to work out group meetings, to do work in other courses, and to learn the topics necessary to complete the assignment. A project is usually the focus of class lecture and discussion during the duration of the project, but typically there is a single artifact produced at the end of three weeks. The practice of producing one release/artifact has caused mixed success in these large projects — sometimes groups fail to deliver even a compiled program.

This past year, we changed to requiring many small releases before completion of the final project, giving students between two and seven days to complete a release. We then work with our teaching assistants to look at these releases and provide groups with feedback while they are still working on the project. Using this practice, every group successfully completed the project and the quality was much higher than what we experienced in previous semesters.

Student View

Many students abuse the time given in a large project by ignoring the project until the last minute then coding in long spurts until it is finally done. This style of working gives our courses a reputation for being hard and requiring all-night coding sessions. While this process may make sense in the context of juggling all the demands placed on a student, it leads to many problems when creating a good software project:

1. Communication between group members is generally very tenuous unless they are all in the same room. Since no one is certain when a specific feature will be worked on, it is hard to count on a feature getting done, let alone planning to use it, improving upon it, or adding to it.
2. One way of dealing with the communication problem is to meet once at the beginning of the project and break it up into chunks that can each be managed by one student working alone. The students then meet again at the end of the project and attempt to integrate their individual parts into a working whole. The first step goes well but, unfortunately, the last step rarely does for average groups.
3. When dividing the work, some students may have much more to do than others in the group, either because some features were not understood well enough when the project was planned or because one student got very excited about a part and added many extra features. Additionally, most students do not understand the details of the other parts of their project.

Making small releases has helped relieve these problems simply because it requires the group to communicate more often. Not only can the course staff better monitor the group's progress, but so can the students. Because they have to integrate their code more often, they typically have something that they could run while they develop and code. Students reported that this led to even more intra-group communication because having a running program gave them more to discuss with their group members: how to improve specific features, curiosity about how other parts of the project were implemented, and plans to determine what parts remained to be done.

Students also reported that they were actually proud of their projects. Many more groups were inspired to add additional features as they worked with their programs to make them easier to use or more interesting. In one case, students were asked to complete a game that could be run from a web page. One group told other students in their dorm about the web page and soon had a large user community. As people played, they made suggestions for new features. The group would publish new versions of the game as often as every twenty minutes! Many of these features were not part of the specification for the game, but were rewarded with extra credit.

Educator View

The instructor developing small releases for large projects must do some more work to take advantage of these benefits. First, one must decide what will be required for each release and schedule these deadlines as if they were real, including minimizing conflicts with the university schedule. In essence, each becomes an assignment in itself. This extra work is balanced in some sense because it may make it possible to better plan the order of topics in the course.

Additionally, each release must be checked and feedback given to the group as quickly as possible. This is even more important because these mini-assignments build toward a single final project and feedback after the assignment is over is all but useless to the group. In our courses, we typically meet with the group as a whole once a week during the project, demoing their project, discussing its design, and planning for the next deadline. In fact, the role of the course staff is often crucial to realize truly big progress from these small releases.

For example, in our advanced programming course we have asked students to complete a LOGO interpreter and program-

ming environment. [14] They had three and a half weeks to complete the assignment and we gave them six deadlines: three required written submissions and three required running code. The first two deadlines attempted to get students to think about the project by asking them to explain specific design issues and use cases [5] with respect to the project. For each of the next three weeks, they turned in successively larger releases of their project, the last being the final version. In each case, they were told to focus on getting the current, smaller, set of requirements finished rather than trying to show that they had started, but not finished, all parts of the project. Finally, after the final version was submitted each student in the group was asked to complete an individual project post-mortem, reflecting on the group experience. [13]

An unexpected benefit of these small releases was that the course staff was able to grade the projects more quickly and give better feedback because they already knew the details of the code. They had learned them as the project was built instead of having to learn them after the fact. Teaching assistants reported that student groups were more open when talking with them if they started from the beginning of the project as opposed to only starting a dialogue after the project was complete (and the student's grade was more clearly on the line). This resulted in faster, higher quality grading.

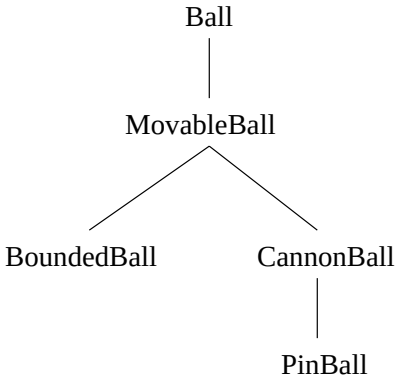
5 REFACTORING FOR LEARNING DESIGN

We use refactoring both to improve the quality of student programs and to help students understand basic tenets of object oriented software design.

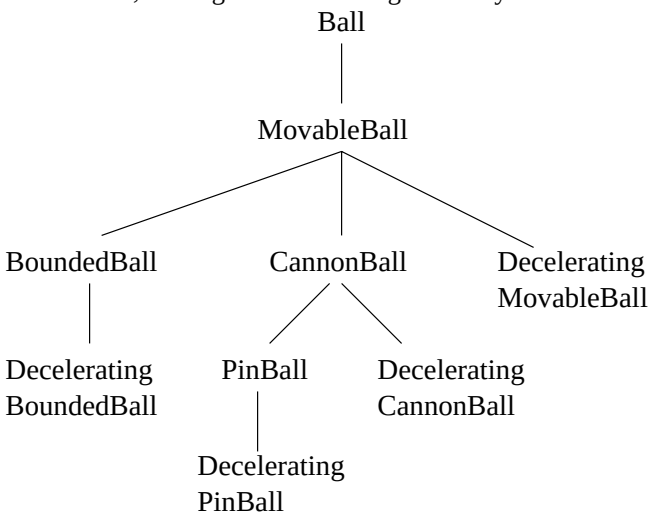
For many years we used a form of apprentice learning in which we provided simple, elegant designs that students implemented in solving problems [2]. The idea was to instill a sense of elegance by experiencing our designs. However, students were not able to internalize the design principles simply by filling in a finished design. Students would not use the principles in our designs because they could not appreciate them as useful in solving problems: they appreciated the designs only as rules to follow in order to receive a good grade.

Now we ask students to develop the simplest (to them) working solution they can to solve a problem. We then ask them to change their solutions to accommodate changes in the problem specification. We help them understand how to refactor their solutions to incorporate design patterns and fundamental software design principles that they are able to appreciate in a more visceral way because their solutions can be refactored to accommodate the changes.

For example, we start with a series of examples from Budd [4] that introduce a simple bouncing ball simulation to a game that fires cannon balls at a target and finally to a pinball game. Over the course of these examples we build the following inheritance hierarchy for the balls used in each game in which each kind of ball responds differently to the `move()` message.

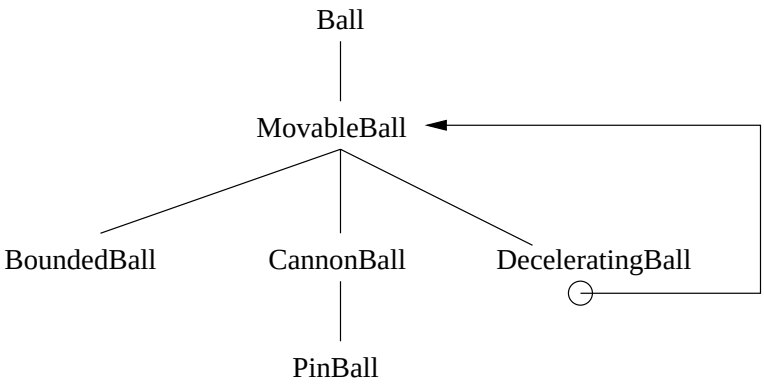


The students are then asked to allow some balls to decelerate in the programs (according to friction or some other property). Initially they create an additional subclass for each kind of ball, leading to the following hierarchy.



For most students this is a simple, easy to understand and implement solution. However, the students also realize that there is a lot of duplicated code since each decelerating subclass changes `move()` in the same way. In particular, it is easy to motivate that a change made to one subclass will need to be changed in all the subclasses. Moreover, any new kinds of balls will need a decelerating subclass in addition to their own.

Students understand that this is not an ideal solution and are primed to find a better way to solve this problem. Since all balls adhere to the same interface, they can be substituted for each other. A movable ball can be used in place of a cannon ball can or a decelerating pinball. Using this principle we show students how to implement a decelerating ball that takes another kind of ball as an argument and delegates the bulk of its work to that ball and adding its decelerating behavior. We show the students the diagram below that characterizes our solution and ask them to refactor their first solution to fit this model.



In this case, they are using the decorator pattern [8] but do not know it as such. After going through another example, we show them the general pattern, but by then they have internalized it and can explain when it is useful. Instead of telling them the pattern and asking them understand it from some abstract description, we have shown a concrete example and motivated them to find a better solution (which just happens to be one for which we already have a name).

6 SUMMARY

No single practice of XP stands on its own [3], instead it must be reinforced by all practices of XP. For example, designing for the current requirements as simply as possible only works if you are willing to pause to refactor any part of the code as needed. You can only feel comfortable refactoring code if you collectively own and understand all the code. Pair programming helps promote this collective ownership. In this paper we have discussed several ways for academics to embrace the changes espoused by advocates of XP.

Currently, our students do not necessarily practice XP when they program outside of the classroom. Instead, we have attempted to design our curricula and methods to help students practice certain aspects of XP and to understand how these practices can improve the way they think about programming and program design by giving them a view of how programs are constructed.

Thus we feel our efforts are certainly in the style of XP even if we are not doing all twelve practices. However, we feel that more growth is still possible by incorporating some additional practices.

1. We have begun to emphasize testing in our early courses by providing test programs for students to use. However, we want to move toward student-coded test programs. Using tools like JUnit [12] we would like to automate the testing process so that students test their code each time they compile.

2. All instructors advise their students to design (or plan) before writing their code, sometimes beginning students even follow that advice (but it is hard to avoid the lure of the computer). We have begun to incorporate the planning game, along with metaphor (or vision), to make this phase of the project more useful, fun, and concrete for the students. Instead of simply asking students to create a UML diagram, we ask them to make stories, or use cases, and create a project web page that acts as an advertisement the team's vision of the project.

3. It is especially hard with group projects to make sure that everyone in the group understands the entire project. To promote better understanding of the overall project, we would like to move students around within and without their group. Additionally, this would force groups to take on new members during the project and have some plan and materials to get new members up to speed on the project's design.

REFERENCES

[1] Manifesto for agile software development. `http://agilealliance.org/`.

[2] Owen Astrachan, James Wilkes, and Robert Smith. Application-based modules using apprentice learning for cs 2. In *Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, pages 233–237. ACM Press, February 1997.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[4] Timothy Budd. *Understanding Object-Oriented Programming with Java*. Addison-Wesley, 1998.

[5] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[6] Martin Fowler. Put your process on a diet. *Software Development*, December 2000. `http://www.martinfowler.com/articles/newMethodology.html`.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[8] Erich Gamma, Richard Helm, Ralph Johnson, , and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] Chet Hendrickson. When is it not xp? `http://www.xprogramming.com/xpmag/NotXP.htm`.

[10] L. Hou and J. Tomayko. Applying the personal software process in cs1: An experiment. In *Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, pages 322–325, 1998.

[11] W.S. Humphrey. *Introduction tothe Personal Software Process*. Addison-Wesley, 1997.

[12] `http://www.junit.org`.

[13] Norm Kerth. An approach to postmorta, post-parta and post project reviews. `http://www.retrospectives.com/`.

[14] Seymour Papert. *Mindstorms.* Basic Books, 1980.

[15] Laurie Williams. *The Collaborative Software Process*. PhD thesis, University of Utah, 2000.

[16] Laurie Williams and Robert Kessler. Experimenting with industry's pair-programming model in the computer science classroom. *Journal on Software Engineering Education*, December 2000.