Part III

XTT: Extreme Technology Transfer—Introducing XP and AMs

Extreme Programming and other agile methodologies consist of prescribed practices. Some of these agile practices are based on previously established best practices. Practitioners and educators feel confident in using and teaching these best practices. In other cases, agile practices alter or depart from best practices. These are the harder practices to adopt and to have the confidence to teach. The authors in this section share experiences and techniques for teaching agile practices and transitioning teams to their use.

Part III begins with chapters written by university educators who have used various XP practices in the university classroom. First, in Chapter 21, Owen Astrachan, Robert Duvall, and Eugene Wallingford discuss their experiences in integrating XP into their classes. These educators have developed an innovative form of "pair programming" by having the instructor play the role of the driver. Jointly, all the students in the class play the role of the navigator and guide the teacher to a successful program implementation. Additionally, they have increased the number of "releases" required of their students to provide more timely and increased feedback. Last, they have introduced refactoring as a means to improve understanding of programs and design patterns.

In Chapter 22, Mike Holcombe, Marian Gheorghe, and Francisco Macias describe their experiences in integrating XP into their senior-level course on software engineering projects. In this course, the students build real projects for real clients. Customer satisfaction and high

quality are essential because maintenance is virtually impossible as the students graduate and leave their projects behind. The authors report success and a positive student response. Dean Sanders shares students' mostly positive perception of XP and XP's practice of pair programming, based on a pilot course, in Chapter 23. David Johnson and James Caristi share similar experiences in their software design course in Chapter 24. These two educators describe their successes and offer some suggestions for future use of XP in a software design course.

Chapter 25 presents the experiences of Ann Anderson, Chet Hendrickson, and Ron Jeffries in running their tutorial on user stories and the planning game. This chapter is extremely valuable for both educators and developers.

Joshua Kerievsky contends that XP's values of feedback and communication support continuous learning. This learning can enable personal and process improvement. In Chapter 26, Joshua suggests that XP be augmented with a learning repository and organizational support for study groups and iteration retrospectives.

The concepts of the XP release planning practice can be difficult to sell and internalize with students and professionals, including developers and businesspeople. In Chapter 27, Vera Peeters and Pascal Van Cauwenberghe present a playful but effective interactive game to teach these concepts. While planning the game, participants experience firsthand user stories, estimation, planning, implementation, functional tests, and velocity.

In Chapter 28, Moses Hohman and Andrew Slocum discuss an innovative practice they term "mob programming," a variant of the XP pair programming practice. With mob programming, groups larger than two work together to refactor code. Employing this practice, their team has strengthened its use of other XP practices, such as pair programming and automated unit test cases. They have also further embraced the XP values of communication and courage.

"Show me the money." Some managers and practitioners remain skeptical of agile practices and methodologies. They desire proof before transitioning from their "trusted" practices. In Chapter 29, Laurie Williams, Giancarlo Succi, Milorad Stefanovic, and Michele Marchesi offer an empirical model for assessing the efficacy of agile practices and the impact of their use in creating quality software.

Chapter 21

Bringing Extreme Programming to the Classroom

—*Owen L. Astrachan, Robert C. Duvall, and Eugene Wallingford*

Part III

In this chapter, we discuss several features of XP that 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 although we embrace the philosophy and change of XP, we have not (yet) adopted its principles completely.

Introduction

Extreme Programming (XP) [Beck2000] and other light or agile methodologies [Agile2000; Fowler2000A] have gained a significant foothold in industry but have not yet 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 [Williams2000; Williams+2000]. However, the general tenets of XP are less known, and the engineering background of many academic computer science programs facilitates adoption of process-oriented methodologies such as the Personal Software Process (PSP) even early in the curriculum [Humphrey1997; Hou+1998]. 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 from 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 [Beck2000], XP takes good practices of professional software development to extreme levels. XP introduces a planned and coherent methodology without becoming overly dictatorial or secretarial. The four values of XP are given as communication, simplicity, feedback, and courage.

As we explain in this chapter, these values form the foundation of our approach. Thus, we think we're following the spirit of XP, although we're certainly not following every XP practice (testing, pair programming, planning game, and so on) [Hendrickson2000].

From these core XP values, five principles are given as fundamental to XP. Our approach uses each of these principles: rapid feedback, assume simplicity, incremental change, embracing change, and quality work.

Ten "less central principles" from [Beck2000] are given, of 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—programs that are fully generic and 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 that 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 mind-set, 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 [Fowler2000B].

In this chapter, we report on three aspects of XP that we have employed very successfully. We have used XP in our introductory programming courses for majors, in advanced courses on object-oriented software design, and in programming courses for nonmajors. These three aspects are as follows:

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

Our Clients

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

- 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.
- We want our curriculum, assignments, and materials to be adopted and adapted by educators all over the world. Our materials must be simple and elegant, and support adaptation and refactoring to meet local demands. The process and materials must resonate with educators at a level different from the resonance we hope for with students.

Our student clients take several courses each semester. They devote 20% to 40% of their time to a course on programming, depending on the 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.

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, which we call the *navigator*. A typical scenario, used from beginning to advanced courses, is outlined in the following two sections. First we explain the process from a student view; then we elaborate on the process from a faculty developer perspective.

Student View

- 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, and principles of computer science and to engage students in the process.
- 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 a potential solution.
- 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.

- The final program is added to the day's Web site for reflection and 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-class-work 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.

- 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.
- 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.
- 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.

Small Releases Mean Big Progress

Students in our nonmajor's programming course as well as our first- and second-year major's courses sometimes work on large projects in groups, in which they may be given as much as three weeks to complete the project. The projects are not designed to require students to work full-time for the duration of the assignment; instead, the schedule is typically padded to allow them time to work out group meetings, to do other course work, and to learn the topics necessary to complete the assignment. Typically, the assignment is discussed repeatedly in and outside of class during this time, but rarely do most curricula require students to demonstrate their progress until the final deadline. This practice has caused mixed success in these large projects—sometimes groups fail to deliver even a compiled program!

This past year, we have changed to requiring many small releases before the final project is completed, giving the students only a few days to a week to submit part of the final product. 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 and then coding in long spurts until it is finally done. This style of working gives the computer science department a reputation for being hard and requiring all-night coding sessions. Although 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.

- Communication between group members is generally very tenuous unless they are all in the same room. Because 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 on it, or adding to it.
- One way of dealing with the communication problem is to meet once at the beginning of the project and break it 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.

- 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 had to integrate their code more often, they typically had something that they could run while they were working.

Students reported that this led to even more intragroup 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 more 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 published new versions of the game as often as every 20 minutes! Many of these features were not part of the specification for the game but ended up being 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 release 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, demonstrating their project, discussing its design, and planning for the next deadline. In fact, the role of the course staff is often crucial to realizing truly big progress from these small releases.

For example, in our advanced programming course, we have asked students to complete a LOGO interpreter and programming environment [Papert1980]. 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 with respect to the project [Cockburn2001]. 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 postmortem, reflecting on the group experience [Kerth2001].

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 the details 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.

Refactoring for Learning Design

We use refactoring both to improve the quality of student programs and to help students understand the 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 [Astrachan+1997]. 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 being useful in solving problems: They appreciated the designs only as rules to follow 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 [Budd1998] that introduce, first, a simple bouncing-ball simulation, then a game that fires cannonballs at a target, and finally a pinball game. During these examples, we build the inheritance hierarchy shown in Figure 21.1 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 the balls to decelerate, or not, in any of the programs (according to friction or some other property). Initially, they create an additional subclass for each kind of ball, leading to the hierarchy shown in Figure 21.2.

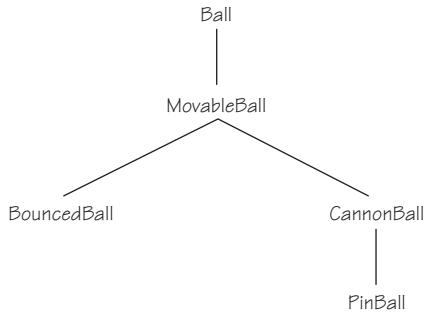

FIGURE 21.1 Initial ball inheritance hierarchy

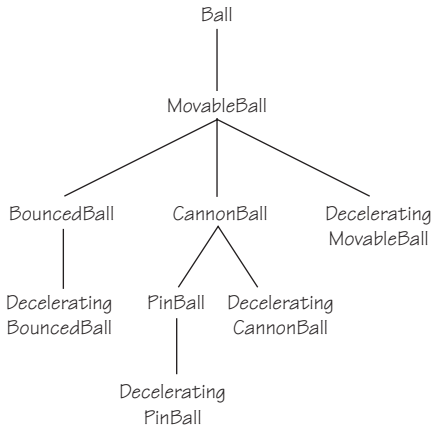

FIGURE 21.2 First attempt at extending the ball inheritance hierarchy

For most students, this is a simple solution, easy to understand and implement. However, the students also realize that there is a lot of duplicated code, because 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 made 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. Because all balls adhere to the same interface, they can be substituted for each other. A movable ball can be used where a cannonball can or where a decelerating pinball can. 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 how to add its decelerating behavior. We show the students the diagram, shown in Figure 21.3, 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 but do not know it as such [Gamma+1995]. 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 ask-

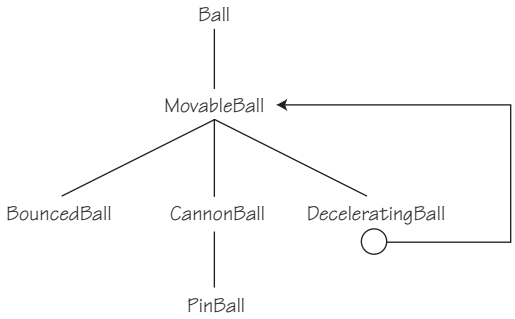

FIGURE 21.3 Refactored ball inheritance hierarchy

ing them to 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).

Conclusion

No single practice of XP stands on its own; instead, it must be reinforced by the practices of XP [Beck2000]. For example, designing for the current requirements as simply as possible works only if you are willing to pause to refactor any part of the code as needed. And you can feel comfortable refactoring code only if you collectively own and understand all the code. Pair programming helps promote this collective ownership. In this chapter, 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. We have introduced some of the ways in which our students differ from professional programmers currently practicing XP. Instead, we have attempted to design our curricula and methods to help students practice certain aspects of XP automatically 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 12 practices. However, we feel that more growth is

still possible by incorporating some additional practices. Here are some that we are beginning to experiment with.

- We would like to emphasize testing more in our advanced programming courses. Using tools like JUnit (see http://www.junit.org), we would like to automate the testing process so that students test their code each time they compile. If something did not pass a test, we hope they would be motivated to fix it immediately rather than ignoring it. Initially, we feel that we would have to write these tests for them to get them into that habit.
- All instructors advise their students to design (or plan) before writing their code, and 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 of the team's vision of the project.
- 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.

Au: Break in URL okay?

References

[Agile2001] *Manifesto for Agile Software Development*. http://agilemanifesto.org/. 2001.

[Astrachan+1997] O. Astrachan, J. Wilkes, R. Smith. "Application-Based Modules Using Apprentice Learning for CS2. *Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education,* February 1997.

[Beck2000] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[Budd1998] T. Budd. *Understanding Object-Oriented Programming with Java*. Addison-Wesley, 1998.

[Cockburn2001] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[Fowler2000A] M. Fowler. "Put Your Process on a Diet." *Software Development*, December 2000. http://www.martinfowler.com/articles/newMethodology.html.

[Fowler2000B] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[Gamma+1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hendrickson2000] C. Hendrickson. *When Is It Not XP?* http://www.xprogramming.com/xpmag/NotXP.htm. 2000.

[Hou+1998] L. Hou, J. Tomayko. "Applying the Personal Software Process in CS1: An Experiment." *Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, 1998.

[Humphrey1997] W. Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley, 1997.

[Kerth2001] N. Kerth. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House, 2001.

[Papert1980] S. Papert. *Mindstorms*. Basic Books, 1980.

[Williams2000] L. Williams. "The Collaborative Software Process." Ph.D. diss. University of Utah, 2000.

[Williams+2000] L. Williams, R. Kessler. "Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom." *Computer Science Education*, March 2001.

About the Authors

Owen L. Astrachan is professor of the practice of computer science at Duke University and the department's director of undergraduate studies for teaching and learning. He has taught programming in a variety

of environments for more than 20 years, each year searching for a simpler, more agile approach in helping students understand computer science and the craft of programming. Professor Astrachan is an NSF CAREER award winner, has published several papers cowritten with undergraduates on using patterns in an academic setting, and is the author of the textbook *A Computer Science Tapestry: Exploring Programming and Computer Science with C++*, published by McGraw-Hill. He won the Distinguished Teaching in Science award at Duke in 1995, and the Outstanding Instructor of Computer Science award while on sabbatical in 1998 at the University of British Columbia. Owen can be reached at ola@cs.duke.edu.

Robert C. Duvall is not an actor, but a lecturer in computer science at Duke University. Before moving to Durham, North Carolina, he did his undergraduate and graduate work at Brown University, where he helped move the curriculum from a procedures-first approach using Pascal to an objects-first approach using Java. He has also taught with Lynn Stein's Rethinking CS101 project at MIT. Primarily, he enjoys using graphics, object-oriented frameworks, and patterns to make current research understandable and usable by novice programmers. Robert can be reached at rcd@cs.duke.edu.

Eugene Wallingford is an associate professor of computer science at the University of Northern Iowa. He has been writing object-oriented programming for more than ten years, in CLOS, Smalltalk, C++, Java, and Ruby. He spent many years doing research in artificial intelligence, where he built applications that used domain knowledge to solve problems in legal reasoning, product design and diagnosis, and ecological systems. Eugene still does some work in AI but spends more of his time studying the structures of programs and helping students learn how to write programs. His work with programming and design patterns developed in tandem with a view on how programs grow in the face of changing requirements, which in turn found a compatible philosophy in Extreme Programming. Eugene can be reached at wallingf@cs.uni.edu.