WHAT WE CAN LEARN FROM EXTREME PROGRAMMING

Suzanne Smith
Converse College
Spartanburg, SC 29302
suzanne.smith@converse.edu

Sara Stoecklin
Florida A&M University
Tallahassee, FL 32307
stoeckli@cis.famu.edu

ABSTRACT
Extreme Programming is defined as a light-weight methodology for small-to-medium-sized teams developing software. It is called extreme because it makes use of extreme levels of good programming practices. Its focus is on coding, testing, user knowledge elicitation, and design. Extreme Programming advocates many ideas that have been known as good practices in software development. In this article, we examine extreme programming, some practices it advocates, and how we, as computer science educators, can utilize these concepts in a computer science curriculum, especially in introductory courses, to more effectively teach software development.

1 BACKGROUND
The concept of Extreme Programming was first defined by Kent Beck [1]. Recently, this new methodology for software development has gained much popularity within the object-oriented programming community. Extreme Programming was created for use by small, co-located teams developing software with requirements that were not clearly defined or were
changing. This methodology emphasizes team work and customer involvement throughout the software development. Communication and feedback are focal points in Extreme Programming. Communication is advocated not just between the customer and the analyst but also between the customer and the developers and within the development team. Feedback is achieved through early and continual testing. Proponents of Extreme Programming assert that its benefits include faster time to market, higher quality software, better customer satisfaction, and highly motivated development teams.

2 CORE PRACTICES OF EXTREME PROGRAMMING

The basic practices of Extreme Programming addressed in this article are pair programming, re-factoring, testing, continuous integration, and evolutionary design. It should be noted that there are many other practices advocated in Extreme Programming and that the core practices of Extreme Programming have been named and renamed by many authors. However, we are examining only the practices that we believe can be adapted for use in a computer science curriculum.

2.1 Pair Programming

Pair programming is the practice of two programmers/engineers sitting in close proximity and participating in the development effort of one programming unit. Both parties are working on the one programming unit, generally with each member performing a task the other is not doing. Typically, the pair is divided into one person entering code or test cases while the other is reviewing and thinking. For example, both members might discuss a method needed in their unit and decide to implement the new method. While one member is keying the method, the other member might be thinking of mechanisms for testing that method. Another example might be the situation where one member is keying a needed class, while the other member might realize that the pair has already developed a similar class and would begin thinking about what programming concept would make this needed class be developed in a different manner. The thinking member might realize that polymorphism, inheritance, or a design pattern usage would make a better design for the functionality of the class. The close proximity and focus on one programming unit promotes the sharing of ideas and encourages a better design to evolve. It has also been observed that the code resulting from pair programming is more defect free, does not take significantly more time to develop than if developed by one member, yields fewer lines of code, and is more satisfying to programmers [3]. Pair programming encourages the design of the programming unit to evolve as the actual programming activity is being conducted.

2.2 Refactoring

Refactoring, as defined by Fowler [5], is an activity of continuous re-design of a program unit to take advantage of programming techniques, especially object-oriented design and design patterns, to make the programs more reusable, simpler, and more efficient. Refactoring can occur at various times throughout the development process. For example, if during the
programming activity one of the programming pair determines that two methods are doing the same functions but with different data, the programming pair would refactor the code to take advantage of polymorphism. When a programmer finds two classes containing the same methods or methods with the same functionality, these classes again would be refactored. When code seems to be out of place, refactoring the code to another class makes sense. When code is repeated within a method, refactoring it by extraction makes it a more reusable method. This re-designing of a programming unit takes advantage of programming techniques of both object-oriented and pattern-oriented programming. A goal of refactoring is to yield programming units with a strong internal structure. In other words, the programming units are generally more reusable, simpler, object-oriented, pattern oriented, and maintainable. Additionally, refactoring allows developers to respond quickly to a change in customer requirements or technology.

2.3 Testing

Testing, as defined by Extreme Programming, includes ideas such as unit testing by programmers, functional testing by users, and automated testing to generate test units that mirror the actual programming units. These concepts fully support the testing of programming units to promote a defect free unit with each unit release. The most promising idea of automated testing involves generating testing units (classes which test a programming unit automatically). These testing units are built as the actual programming unit is being built. This type of testing drives the development process. The scenario works as follows: a programmer/engineer codes the programming unit, tests the unit, thinks of more tests for the unit, tests the unit again, thinks of more tests for the unit, tests the unit again and continues this iterative testing until no more tests can be uncovered. The theory is that this scenario yields well tested modules at the end of the iterations.

Testing in Extreme Programming involves two types of tests. The first type of testing is unit testing which is the testing of individual programming units such as a class. Developers write tests for every class they produce and even write the tests before writing the code. When they add functionality to the original class, they can test the prior functionality with the developed testing unit and add more tests for the new functionality. Unit tests must be 100% correct before testing continues.

The second type of testing in Extreme Programming is functional testing. Functional tests are scripts developed to test clusters of classes, perhaps in a use case, that meet the user requirements and meet performance and constraints prescribed by the users. Functional testing is typically use case driven tests. These tests are the responsibility of the users or customers while the unit tests belong to the developers.

2.4 Continuous Integration

Continuous integration is the concept of integrating new code into existing code and then utilizing the testing techniques defined by Extreme Programming. This practice yields units of
code that are continually tested during development. Using continuous integration, a programmer/engineer integrates new code into existing code after the existing code has been relentlessly tested. Therefore, the release of the change is done when it is tested and everyone knows about the new functionality at the time of the release. While this is dynamic, it should not pose insurmountable problems since the code is released only after thorough testing. This particular idea generates much discussion among the software practitioners, but the result is that it does advocate the thorough testing of modules prior to release as its main premise. On this main premise there is no resistance. It also recognizes that the integration of code in the object-oriented paradigm deserves some focus.

2.5 Evolutionary Design

Evolutionary design involves making iterations of a program within minutes rather than days. The programming pair defines each iteration of the problem and then implements these iterations. Upon completion, the problem is expanded to yield another iteration and then that iteration is implemented. The idea is explained as a planning game. The scenario of a planning game is to write a story (a lightweight use case) to define the problem, estimate the problem, make a commitment to develop the solution to the problem, develop the solution, present the solution, and then begin again by re-writing the story to define more of the problem. As the problem is expanded, the solution and programming architecture evolve. While experienced developers request some hierarchical nature to the stories, problems, and solutions, no one doubts the validity of this evolutionary design process for one use case. The statistics noted by Cockburn [3] support the use of evolutionary design.

3 EXTREME PROGRAMMING CONCEPTS IN COMPUTER SCIENCE CURRICULUMS

Since Extreme Programming proposes or uses concepts that are generally seen as good for software development, as computer science educators, we should be evaluating the role of this new methodology in computer science curriculums. In this next section, we will examine how the concepts of Extreme Programming, described above, can be utilized in a computer science curriculum, especially in introductory courses, to more effectively teach software development.

3.1 Pair Programming

Currently, the practice of pair programming is not utilized in programming courses in most colleges or universities. In fact, computer science faculty view joint work on programming assignments as plagiarism or cheating by students. Most courses in programming encourage students to design and code their own work. While there are exceptions to this observation [2], students are rarely exposed to more than one solution to a problem. However, Extreme Programming demonstrates that programmers can not only work well in pairs but also learn much from discussing code and design with another person. Students and faculty would benefit
from the inclusion of the practice of pair programming in programming classes. At a minimum, Extreme Programming demonstrates that the inclusion of the sharing and discussing various designs or examining the pros and cons of particular solutions to programming problems would be beneficial to students.

Harlan Mills did prescribe a similar method in his Cleanroom approach [4, 7]. In Mills' approach, one student programs the exercise while another writes the test to test the exercise. Then they change positions and redo the assignment. Each student receives two grades, one for the development of the exercise and one for the testing of the exercise.

An approach that we have tried recently utilizes pair programming in a teaching model different from the traditional lecture and laboratory. In this approach, students receive a short lecture on a topic and then proceed to the laboratory to do an exercise using pair programming. This approach was first utilized because of too few computers in our teaching laboratory. However, it proved so useful that we decided to keep the approach even when more computers became available. This approach was especially beneficial when we paired a good student programmer with a student who was struggling. In our approach, the instructor was always present in the laboratory during the programming exercise. The programming exercise was based on the concepts taught in the class, and students were rewarded if they found design patterns or other object-oriented concepts that made the code simpler (which is one of the goals of pair programming).

Research in pedagogical patterns for implementing pair programming and grading the results of such activities are still emerging. However, a recent study in the use of pair programming in computer science classes [8] appears to support the premise that this technique of Extreme Programming is an effective method for teaching programming concepts. Furthermore, this approach to learning is supported by studies done on cooperative learning [6]. Cooperative learning has been especially effective in the teaching of girls. Pair programming may be a useful practice in introductory programming classes in order to promote computer science as a major among female students.

3.2 Refactoring

Refactoring is also rarely utilized or taught currently in computer science courses. Many educators in curriculum meetings are debating such items as what language or when to introduce the concepts of objects and when to require students to write their own classes. The debates on when and where to introduce the ideas of refactoring or what is good design have not become the focus of many academic debates. While we teach polymorphism, most textbooks do not have adequate exercises that require students to think of it as a solution to a problem. With the exception of the model-view-controller pattern, most computer science textbooks do not debate different design techniques, do not conduct design reviews, and do not require the use of patterns. Computer science courses often do not encourage or reward good design techniques. In fact, except for a few upper-level specialized courses on software engineering topics, computer science courses do not attempt to define the techniques of design.
This article is not so bold as to recommend the teaching of refactoring in a particular course. However, it does seem apparent that the traditional topics taught in the computer science courses need to be critically evaluated. Traditionally, the CS2 or Data Structures courses taught those items we thought were critical to understand procedural programming and the patterns needed for the procedural programming paradigm. These included building procedures and learning patterns such as linked list, stacks, and queues. With the integration of object-oriented and pattern-oriented programming in the computer science curriculum, it is time to revisit the topics taught and defend the inclusion of some refactoring and good design topics in these courses.

3.3 Testing

Testing is a topic that has been long overlooked in the computer science curriculum. There is no specific place in the typical computer science curriculum where testing is taught as a topic which requires some programming exercises. Generally, students learn the terms black box and white box testing but rarely are they asked to create tests which represent one or the other type of testing. Most computer science educators do not ask student to generate testing units for programming assignments, and rarely do students have a good understanding of how to investigate a programming unit in order to determine how to adequately test the unit. Students generally have the attitude that if it works once, it is tested. The concept of exhaustive testing, complete testing or iterative testing is typically not introduced in programming classes such as CS1 and CS2. Additionally, faculty who grade programming assignments rarely rigorously test student programs.

Harlan Mills [4, 7] advocated a tight integration of testing with development in the early 70s, but few curriculum adopted this approach. Most of the programming courses could include a section of testing. For example, when teaching if statements, the process of testing if statements should also be addressed.

Decomposition of testing into unit tests and scenario testing has been advocated for the last few years. However, now that a testing unit is normally defined as a class and scenarios are defined as those paths through use cases, it is even more important to integrate these types of tests into our computer science curriculum. When classes are first developed by students is the time to introduce class testing. When use cases of collaborating classes are first developed by students is the time to introduce scenario-based testing.

The approach that we have taken in our attempt to introduce more teaching into the CS1 course is to teach and practice testing techniques in the laboratory portion of our CS1 class. Extensive focus is given to testing during the first few weeks of a semester. Students learn different types of tests and have programs (provided by the faculty member) for which they must build tests. After generating tests individually, we discuss the pros and cons of the tests which individuals have produced in order to determine better tests for the code. The idea of systematically testing units of code is conveyed nicely to the students in this approach because they spend so much time actually building and running tests. They also gain a better
understanding of functional requirements during this exercise. The students are then required to produce tests for all programming assignments during the semester.

3.4 Continuous Integration

Of all the concepts advocated by Extreme Programming, this concept is certainly in contrast to how we normally teach computer science courses. We often have students build units that are one-time-throw-away units. These units require no integration. Rarely do we ask students to integrate new functionality into existing code or to integrate working programming units.

In contrast to how we teach, this concept may be the easiest to integrate into the computer science curriculum. One approach is to have one large program developed incrementally during the first programming course. Techniques have evolved that allow the students to develop the first portion of the assignment and after that portion of the assignment is completed, the student is given a correct version on which to build the new functionality for this system. Large programming assignments built incrementally in this manner would be easy to implement in the programming courses and certainly would yield an approach to development of system that supported the concept of continuous integration.

3.5 Evolutionary Design

During the age of procedural programming, educators gave students various hints on how to conduct evolutionary design on programming assignments. In walking the halls of the computer science buildings, the instructors could also be heard telling another student what to try and not to try in implementing a programming assignment. Incorporating this practice into programming courses could be designed to support the programming concepts being taught. This practice might work especially well in a programming course with a laboratory component.

4 CONCLUSION

Certainly, computer science educators fall short of teaching the elements of Extreme Programming to students. For many educators, this approach may not be something they would like to advocate. However, it is certainly worth investigation if the idea is working in industry and research proves it to be a productive paradigm. We, as computer science educators, need to research the use of this development paradigm and investigate further the integration of its proven practices into the computer science curriculum.

REFERENCES


