Managing a Project Course Using Extreme Programming

Carol A. Wellington
Department of Computer Science
Shippensburg University, Shippensburg, PA 17257, cawell@ship.edu

Abstract – Shippensburg University offers an upper division project course in which the students use a variant of Extreme Programming (XP) including: the Planning Game, the Iteration Planning Game, test driven development, stand-up meetings and pair programming. We start the course with two weeks of controlled lab exercises designed to teach the students about test driven development in JUnit/Eclipse and designing for testability (with the humble dialog box design pattern) while practicing pair programming. The rest of our semester is spent in three four-week iterations developing a product for a customer. Our teams are generally large (14-16 students) so that the projects can be large enough to motivate the use of configuration management and defect tracking tools. The requirement of pair programming limits the amount of project work the students can do outside of class, so class time is spent on the projects and teaching is on-demand individual mentoring with lectures/labs inserted as necessary. One significant challenge in managing this course is tracking individual responsibilities and activities to ensure that all of the students are fully engaged in the project. To accomplish this, we have modified the story and task cards from XP to provide feedback to the students and track individual performance against goals as part of the students’ grades. The resulting course has been well received by the students. This paper will describe this course in more detail and assess its effect on students’ software engineering background through students’ feedback and code metrics.

Index Terms - About four, – Agile Methodology, Extreme Programming, Software Engineering Education.

INTRODUCTION

Shippensburg University offers a software engineering concentration within its B.S. in Computer Science degree. One aspect of this concentration is that students must develop products using both plan-driven and agile methodologies so that they experience a broad base of disciplined software engineering methodologies. This is done through two courses: Traditional Life Cycle and Extreme Programming (XP). However, we have found that it is difficult to ensure that students are following the XP methodologies in a disciplined manner. This paper describes the challenges that arise in teaching XP and strategies for overcoming them.

EXTREME PROGRAMMING

Extreme Programming (XP) is one of a number of agile methodologies. These methodologies are designed to enable a team of software developers to deliver a product quickly while maintaining confidence in the quality of the product that plan-driven methodologies provide.

XP includes 12 “practices” [1][6] and our course includes them all. However, the challenges that we face were rooted in these practices (each of which will be discussed later in the paper):

- Test Driven Development
- Pair Programming
- The Planning Game
- The Iteration Planning Game
- Refactoring

COURSE STRUCTURE

The course is a 4-credit course that lasts for 14 weeks meeting for two hours on two days each week. The thirty-one students were divided into two teams of 15 and 16 students each. While these teams are large by XP standards, the scale of projects that they can undertake helps motivate the need for a rigorous development process as projects of this scope will not be successful if there is no process to coordinate the activities of each of the individuals. The course is designed for upper division students in the software engineering concentration, but does not require any previous software engineering courses. Their only experience is that of traditional computer science upper division students.

The course was divided into four segments: a two-week “technique spike” to introduce the concepts and techniques of XP and three four-week development iterations.

TECHNIQUE SPIKE

The first two weeks of the semester are spent developing XP skills through a series of labs (prior to initiating the course project). These labs make students practice these techniques: pair programming, test-driven development (TDD), designing for testability, refactoring and planning in XP.

PAIR PROGRAMMING

All of the labs and project activities were required to be accomplished in pairs. The students were instructed on the roles of the pilot and the co-pilot (this is described in more
detail in [5] where the pilot and co-pilot are called the driver and the partner) and were reminded to switch roles regularly during their initial labs.

**Test Driven Development**

The actual implementation phase of XP uses test-driven development (TDD) [9] in which the engineer first writes automated tests before building the production code. Our automated tests were developed in JUnit as a plug-in to Eclipse. The students found this environment to be quite comfortable especially when they realized that Eclipse also automates their interactions with our project repository in CVS.

Teaching the students how TDD works was accomplished with a lab to develop a small pair of classes. The lab started by leading the students carefully through TDD, so the steps were of high granularity (write this specific test, watch the test fail, make the test pass (and don’t implement anything else), “clean up”, re-run the tests . . .). As the lab progressed, the granularity of the instructions decreased, but continued to remind the students to follow TDD.

In addition to basic TDD, the lab required the development of mock objects to ensure that the students had the necessary background. This was accomplished by requiring the classes being developed to follow the Observer design pattern. That had the side effect of reviewing that pattern which the projects were likely to require. By the end of the lab, they had developed a non-trivial pair of classes using TDD.

**Designing for Testability**

One of the significant challenges for TDD is automating the tests for GUI aspects of the system. In particular, writing JUnit tests for code that is initiated through mouse actions or that builds visual output is cumbersome at best. Therefore, it is important that the GUI portion of the system be as “thin” as possible. To achieve this, the students completed a lab on the Humble Dialog Box design pattern [3] outside the context of the project. Essentially, this pattern allows all of the logic driving the GUI to be automatically tested and leaves only very simple widget logic untested. This eliminated the need to write GUI-specific automated tests and helped our students maintain their interest in TDD.

**Refactoring**

Refactoring is the process of improving the quality of the design or implementation of the system without changing its functionality[4]. While there are a great number of standard refactorings, we focused on the ones that are automated by Eclipse with which the students became quite comfortable. The students practiced some of these refactorings in the TDD lab.

**Planning**

The final topic of the technique spike was planning. For this topic, the students were instructed on the philosophy behind planning in XP and the details of the Planning and Iteration Planning games.

In order to help track our progress during an iteration, we enhanced story cards by adding a section that allows the tracking of the tasks related to a story as a progress indicator. The format of these story cards is shown in Figure 1. They are designed to be folded in half during the planning game as the task tracking information is only needed after the iteration is underway.

**Course Structure**

Many of the other aspects of this course were structured around other XP practices.

**Iteration Length**

XP recommends that each iteration last an average of two to three weeks[2]. However, since iteration planning requires the entire team, it must be done during one or two class periods. This lengthens the time it takes to start an iteration, so we extended iterations to be four weeks long with planning in the first week and a demonstration to the customer on the Thursday of the last week of the iteration.

**Stand-Up Meetings**

XP promotes short meetings each morning where each individual reports on their progress and any challenges they have encountered. Our students started every class period with a 15 minute stand up meeting.

**Visibly Tracking an Iteration**

All of the story and task cards were posted on a wall in the classroom and the students were encouraged to keep that information up-to-date. This gives everyone easy access to information about who is working on which tasks and promotes communication amongst the team.

**Maintaining Accountability**

The most significant instructional challenge in this course is to ensure that every student is using the techniques rigorously. The students see that the deliverable is the code, so they are often tempted to abandon some of the more challenging techniques. It is critical that that see that XP is NOT the absence of a process, but is a rigorous process in itself.

The first step in encouraging them to follow the process is to make the project large enough to intimidate them. When the project is significantly larger than anything they have developed, they are motivated to use the process. In addition, the larger projects demand larger teams and that motivates the need for standards in communication, tools, and development strategies.

In order to verify that each student was following the process and identify students who need help developing the skills required to follow the process, we modified the task card format. In addition to the usual data, our task card includes a
status box and a place to list the classes that were modified as a result of the task.

The status box facilitates communication between the students and the instructor. When a student completes a task, they put a yellow sticker on the task and note the time they spent on the task and the classes that were affected by the task. The faculty member then reviews the student’s work and puts a “review” sticker on as follows:

- Green if the code is correct, includes appropriate test cases, and matches our coding style,
- Red if any of the green criteria are not met, and
- Blue if there are circumstances that prevent an accurate assessment of the student’s work (generally, this means that changes related to another task have conflicted with this student’s work).

While these stickers help the students understand XPs expectations, they also help in identifying the students who need help in learning to follow the process. Also, at the end of each iteration, statistics about the success student had with his/her tasks are reported and form the basis for a portion of the grades. Each iteration, the percentage of tasks each student selected in the Iteration Planning Game and received green stickers was computed. “Personal deliverables” accounted for 50% of the students’ grades and was computed as the average of those percentages. Note that the students were not penalized if their initial status was not green – only the final state of the task was used.

Results

STUDENT DRIVEN PROCESS MODIFICATIONS

At the end of the first iteration, both teams failed to deliver the bulk of the functionality in the selected story cards. At the scheduled presentation to the “customer,” they showed code that exhibited the required behavior, but had developed very little of the required user interface. For example, one project had this story: “The user must be able to purchase a lot after specifying the stock, the price per share, the number of shares and the total fees of the transaction.” While they had developed a Stock class with a “buy” method with the appropriate parameters, they had not developed user interface that allowed the user to see the results of that transaction. In fact, they had not even developed a screen that allowed the user to input the required information. Clearly, something was terribly wrong.

After the completion of the iteration, we performed a post mortem to diagnose what was clearly a systemic problem. The students decided that they had focused on the internals of the system because that was the functionality with which they were comfortable; they had previously developed databases and logic, but very few of our classes require significant user interface development.

Given the source of their struggles, the students designed a change to the process to try to ensure that subsequent iterations would be more successful. They created the role of “story advocate” whose responsibility is to specify user level tests for the story, coordinated the development of the tasks associated with the story with those tests in mind and execute the tests to verify the completion of the story prior to delivery to the customer.

With the creation of story advocates, the students initiated a sticker system similar to that used on the task cards. When all of the tasks of a story were complete (at least yellow stickers), the story advocate would run his/her tests and, if the tests passed, put a yellow sticker on the story card. That would signal the instructor to verify that required the functionality existed and the faculty member would give the story a red or green sticker as appropriate.

There are two important things to note about this modification. First, and most notable, the students, without any prodding, designed a solution that is consistent with the motivations behind XP. In fact, they extended the concept of TDD to include customer level tests. This innovation did help them deliver user functionality in subsequent iterations. Second, this is not an innovation that XP has overlooked. In true XP environments, the customer (or a representative of the customer) is a part of the development team and is responsible for developing acceptance tests for every story. Essentially, the story advocate is compensating for our lack of an on-site customer.

PAIR PROGRAMMING

Pair programming for students has been used in a number of situations (examples in [7],[8]) and our results do not differ significantly from previous results. After the technique spike labs had introduced the subject, the students exhibited little resistance to pair programming. In particular, most of the students prefer pair programming and it allowed students whose skills were not as well developed to be valuable team members (and to learn from their peers). This was assessed by observing the students (the instructor had all of the students in previous courses and had information necessary to make an anecdotal comparison) and by the students themselves (the opinions of their peers’ value on the project improved throughout the semester).

Our students have worked in pairs in previous courses and, therefore, were reasonably comfortable with the pairing process. We did not manage the pairs and found that the students did not switch pairs as often as XP advocates. This is probably a result of the fact that previous courses have not required switching of partners at all. In the future, we would encourage switching by measuring (and posting) the percentage of possible pairings that the team had used to make them aware of their choices about switching.

In addition, pair programming did cause some logistical problems because some students had difficulty arranging for pair programming outside of class. While students were allowed to play with strategies by developing throw away prototypes and researching technologies alone, most of the class time after the spike was spent in pair programming to alleviate this challenge. In addition, this gave ample time for ongoing mentoring in XP techniques by the faculty member.
FIGURE 1
MODIFIED STORY CARD.
CODE QUALITY

The projects developed by the teams in this class were very successful. The teams developed an average of 152 production LOC per student which resulted in significant, user visible functionality. The resulting code was reasonably well designed, though inexperience is still evident in the overall design. The following table summarizes some code metrics for the two projects from Fall 2004.

<table>
<thead>
<tr>
<th>TABLE 1 CODE METRIC RESULTS</th>
<th>Stock Eval</th>
<th>Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>max method length</td>
<td>45</td>
<td>71</td>
</tr>
<tr>
<td>avg method length</td>
<td>4.629</td>
<td>6.237</td>
</tr>
<tr>
<td>avg Cyclomatic Complexity</td>
<td>1.355</td>
<td>1.592</td>
</tr>
<tr>
<td>max Cyclomatic Complexity</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>max nested block depth</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Max num parameters</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Max num attributes</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>avg weighted methods per class</td>
<td>7.217</td>
<td>8.643</td>
</tr>
<tr>
<td>max weighted methods per class</td>
<td>31</td>
<td>51</td>
</tr>
<tr>
<td>max classes per packet</td>
<td>25</td>
<td>23</td>
</tr>
</tbody>
</table>

Most of these metrics show that, on average, the code is clean. The maxima of these statistics (which point out more complex portions of the code) are generally found in the GUI sections of the system. This is consistent with the observation that TDD was not being used rigorously in the development of those portions of these systems (see discussion below).

REFACTORING

We practiced some refactoring techniques during the technique spike and, in general, the students applied those refactorings appropriately. In addition, they often “refactored” by re-designing and re-developing portions of code that had begun to rot. While both teams performed some refactoring, one team embraced it much more than the other. That team reserved about one third of its project velocity (the amount of development they expected to be able to complete) for refactoring in the third iteration.

TEST DRIVEN DEVELOPMENT

While our use of story and task cards created a structured environment that encouraged the students to abide by the subset of XP strategies we were studying, test-driven development was not natural for our students. The teams developed relatively less test LOC than would be expected in TDD. If they were carefully following TDD principles, we would expect to see much higher ratios of test LOC to production LOC. (In fact, that ratio may be significantly higher than one to achieve complete coverage of the tests.) One team had about the same number of production and test LOC, but the other had 50% more production LOC than test LOC. This means that, in both teams, large portions of their code were not covered by their tests.

In fact, the students’ adherence to TDD was probably worse than the LOC measures predict. Even for code that was covered by tests, much of the code was developed prior to writing the tests and they developed the tests only to get the green sticker on the task card. Clearly, we need to work on having them follow TDD better in the future. In the future, the following changes could improve our use of this technique:

- Measure the coverage of tests associated with each task (this would require some tool development) before giving the green task stickers,
- Practice the Humble Dialog Box design pattern more thoroughly. The GUI code was least covered and the source of much of our difficulties. While we completed a lab using this design pattern, further practice would have helped the students generalize its use better.

CONCLUSIONS

This course has been quite successful and the strategies we added to planning helped ensure that the students followed XP practices. In general, the students saw the value in those practices. Each student wrote a final paper summarizing their experiences and the comments in those papers underscore the fact that they understand that XP is not the absence of a process.

The only significant weakness we found was in the students’ ability to stick to TDD particularly for GUI-related classes and we have strategies for addressing that challenge in future offerings of this course.

In addition, the next time we run this course, we will use Cruise Control to automatically run the builds and generate code coverage statistics. This should increase students’ awareness of their test writing skills and should provide opportunities to mentor students on TDD strategies.

REFERENCES