A Comparison of Pair Programming to Inspections for Software Defect Reduction

James E. Tomayko
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

ABSTRACT

Even though convinced of their efficacy, software development managers are looking for ways to reduce the cost of inspections. Recent work shows that inspections by two engineers have as good results as by using a larger team of five. This indicates that pair programming might be effective as an inspection technique, as other research shows that a meeting is not necessarily better than not meeting. An experiment was conducted to see if pair programming reduces defects more than formal inspections. Results indicate that pair programming is more effective. A defect rate of 9.6 per thousand lines of code, much lower than that of a heavier method, were achieved. The implications for teaching are explored.

One of the principal practices of the agile method eXtreme Programming (XP) is pair programming (Beck, 2000). Advocates of XP and pair programming claim that it helps find defects sooner, as a different set of eyes examine all products as they are made (Williams & Kessler, 2000). They liken the pairs to an inspection team (Fagan, 1976).

Earlier research seems to support this contention. A study by Porter and Johnson of inspection teams meeting versus those that do not meet indicates that the results are the same, so it would seem that it is good to have inspections, but these can be by individuals (Porter & Johnson, 1997). Further experimentation by Porter et al. of different sized teams of two, three, and five that did meet indicated that the team of two had similar effectiveness to the team of five, so, apparently, “three’s a crowd” in terms of inspections as well (Porter et al., 1997). This is potentially good news for managers trying to reduce the cost of inspections, since teams of two can be sufficient. This can also support the claims of the effectiveness of pair programming. We tried an experiment to find out if this was true. We wanted to know whether to teach...
full formal inspections or simply to emphasize defect reduction as a side product of pair programming.

**MOTIVATION**

The experiment was to find out if pair programming was as effective as formal inspections. Also, there was an opportunity to acquire data that would be useful in further evaluating XP versus a heavier weight process. This data is mostly in terms of productivity and documentation.

**HYPOTHESIS**

The following was tested:

*Pair programming is as effective as formal inspections meetings in reducing defects in software work products.*

Since the results of XP are largely embodied in the code, code inspections would, of course, dominate. Heavier methods are more production-of-documents-based. Therefore, code inspections would be only a goal of some of the inspections – many would naturally be of documents.

**EXPERIMENTAL DESIGN**

**Subjects**

Carnegie Mellon University (CMU) has a program with the South African government to train software engineers. The students are chosen from female computer science graduates of traditionally disadvantaged all-black schools. They are brought to one site near Cape Town and are given a year’s instruction in remedial computer science courses, introductory software engineering courses, and some core courses for a master’s degree, all via distance education. They are then brought on campus for the spring and summer semesters of the following year. They probably could not be admitted to CMU directly from their undergraduate programs. The year in South Africa is spent raising these women to nearly the level of recently admitted CMU graduate students. The final residence period completes the transformation to a master’s graduate basically indistinguishable to one that was on campus the entire time.
As part of their degree program, there is a practicum consisting of the application of course material to a large project. Since these women appeared to be of about equal ability, based on similar backgrounds. They had had identical CMU preparation except for very few electives, so they provided a homogenous group for this experiment. Eight students of the 12 in the first cohort volunteered to participate in this experiment, and were divided into two teams of four.

**Design**

We gave the two teams an identical problem, basically it is the one stated in Tomayko (2001). This is an implementation of a spacecraft computer system using command, data processing, and attitude control computers as primaries. These are backed up by software instantiations of these running as background jobs on experiment computers. The idea is that computers have become so powerful in the quarter-century since the launch of the Voyager probe, that excess cycles on certain experiment computers can be utilized as a back up for one of the three main computers. Originally, it had actual machines as back up. Never has a primary computer completely failed on any probe, so the back up is unlikely to be used, anyway. This scheme results in size, power, and weight savings, the three major concerns of spacecraft designers.

To implement this system, one team trained for using XP, while the other used the Team Software Process (TSP)\textsuperscript{sm} with the scripts and procedures as described in Humphrey (2000). It consists of a project organization with scripts to support all activities. Results include a comprehensive set of documentation. All the students had an introduction to TSP procedures, certainly enough to follow the book. We chose TSP because it is inspections-centered. The XP team got a copy of Beck (2000) and had follow up sessions with the instructor.

**Variables**

The primary item of data was the number of defects. For the XP team, these were primarily “escapes.” We did not think that it would be useful to count every error that was identified by a pair during unit tests, since these were done before release of the code to the rest of the team. The XP practice of developing the tests first and then constantly checking progress by running them would reveal the sort of defects found by developers before formal inspections using conventional methods. Therefore, we instructed the XP team

\textsuperscript{1}The Team Software Process is a service mark of Carnegie Mellon University.
to count defects that showed up in integration, after an object passed unit tests. Both teams used Java, so the XP team could use JUnit [http://junit.org] for testing. The TSP team tracked any defects found in inspections, so those found by individual developers were not tracked.

Each team reported raw defects, but in order to normalize the data and to have a meaningful reference, defects per thousand lines of code were also reported. This required that the lines of code be counted, giving a rough measure of productivity, as both teams had the same amount of calendar time, 12 weeks. Since both teams had some familiarity with the Personal Software Process (PSP)sm both used the guidance for determining a line of code given in Humphrey (1995).

**Threats**

Inconsistent counting of defects and incorrect assumptions about the ability level of participants are the chief threats to meaningful results in this experiment. It can be argued that the constant use of unit tests actually finds most of the defects, whereas the software is sometimes not even tested before formal inspections are done. We think that software released for public inspections has been carefully inspected by the developer, especially if they use PSP, so that heightens the comparison to pair programming, as we are then measuring defects that have escaped capture by either means. Defects found in inspections are thus roughly equal to those found as by integration by the XP team, which is only a part of the process of a TSP team. In both cases, the verification means used in the immediately next step in development reveals the data.

It is possible that ability levels skewed the results. Note that we opened these projects to self-selection. This means that the XP team that selected the most leading edge, potentially innovative process was made up of women that took chances and tried to experiment, which were the more able of the group. This was indicated by higher GPAs and the instructor’s judgment. The unfamiliarity with XP and the need to capture certain data is thought to have held them back sufficiently to make them about equal to the team following TSP.

**Data Collection**

The TSP team simply submitted the inspection results form, and basically counted only the major defects on it. The XP team submitted their data irregularly. At first, they reported at the end of cycles, then once a week.

---

1The Personal Software Process is a service mark of Carnegie Mellon University.
CONFORMANCE WITH XP

How much of the XP practices did the XP team use, considering that they were in an academic setting? Most, because they had considerable flex time due to the fact that this project course and an elective were all that they were taking in the summer semester. The most difficult XP practices to follow in an academic program are pair programming, the 40-hr week and having an on-site customer. They were able to schedule a block of time each day and had a workroom with a dozen computers to themselves, making pair programming and simulating the spacecraft computer network easier. They emulated the 40-hr week by limiting the time they spent on the course project to the number of hours required for credit. This course was 24 units, a unit being 1 hr of work per week for a semester. They worked those hours on the project, the elective and normal life filled up the rest. The instructor was the client, usually on-site, but not in the room. The instructor physically entered the XP team’s workroom and asked if there were any questions each day that they were on campus. They also prioritized the stories, as the method requires.

RESULTS AND DISCUSSION

For this experiment, the pair programming XP practice and an inspections-heavy process (TSP) are examined. Results indicated other aspects from the comparison that almost makes it a comparison between XP and TSP. These are noted, but are not the main purpose, which is only to compare an inspections-driven process to XP to find out if pair programming gives the same or better results.

In terms of the main thing we were trying to find out, whether pair programming is as effective as inspections in reducing defects, the answer appears to be “yes.” There were 9.6 defects per thousand lines of code made by the XP team, with 19.7 made by the TSP team. Another TSP team made up entirely of co-located Navy civilian engineers in Newport, Rhode Island had 20.07 defects per thousand lines of code on a different project at the same time. The closeness of these two numbers indicates that we may have found the midpoint of the error rates of TSP teams. Of course, it may be argued that constant unit testing reduced the overall defects as well. However, if we must then say that the XP method resulted in reduced defects, then pair programming remains as a big part of the success.
The XP team’s defect identification rate was as follows in Figure 1. The overall reduction in the number of defects found over time makes sense if you believe that code becomes more stable as time progresses. Also, the 15 June and 6 July reports are for entire cycles, and would be larger than the others, which were of 1 week.

Figure 2 shows the results for the TSP team. Note that this trend follows that typical of inspections-heavy methods like PSP and TSP. Defects are found early in the process. The efficacy of the inspections is demonstrated by the fact that there were no defects found in testing. Some may find this preferable to
the almost constant finding of defects in XP, no matter how small their number. In TSP, when the defects are injected is also monitored, and is thus part of this chart. Note that, eventually, the two metrics are the same.

In order to derive the defects per thousand lines of code metric, it was necessary to count lines of code as well. Table 1 shows the results for the XP team, which divided the system into seven computers, of which three Experiment computers contained the backup software.

Note that the Command Computer was easily the largest and most complex, it contained the fault tolerance and networking software. It was followed by an approximately equal amount of code for the Attitude and Data computers and their backups. The seventh computer was a simulated ground computer, which would transmit commands and receive data. The implementation distributed this software over the several computers, to make the simulation more realistic.

Table 2 shows that the TSP team used a different decomposition for counting. Note that the attitude and data computers are about the same size as

<table>
<thead>
<tr>
<th>Computer name</th>
<th>Lines of code (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>1421</td>
</tr>
<tr>
<td>Command experiment</td>
<td>1445</td>
</tr>
<tr>
<td>Attitude</td>
<td>392</td>
</tr>
<tr>
<td>Attitude experiment</td>
<td>394</td>
</tr>
<tr>
<td>Data</td>
<td>345</td>
</tr>
<tr>
<td>Data experiment</td>
<td>358</td>
</tr>
<tr>
<td>Ground</td>
<td>634</td>
</tr>
<tr>
<td>Total LOC</td>
<td>4989</td>
</tr>
</tbody>
</table>

Table 2. Component Sizes for the TSP Team.

<table>
<thead>
<tr>
<th>Components</th>
<th>Lines of code (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Computer</td>
<td>140</td>
</tr>
<tr>
<td>Command experiment</td>
<td>215</td>
</tr>
<tr>
<td>Attitude computer</td>
<td>410</td>
</tr>
<tr>
<td>Attitude experiment</td>
<td>402</td>
</tr>
<tr>
<td>Data computer</td>
<td>348</td>
</tr>
<tr>
<td>Data experiment</td>
<td>401</td>
</tr>
<tr>
<td>Heart Beat</td>
<td>100</td>
</tr>
<tr>
<td>Socket Server</td>
<td>117</td>
</tr>
<tr>
<td>Total</td>
<td>2133</td>
</tr>
</tbody>
</table>
the XP implementation. This team used the same programming language and roughly the same implementation, but they built as simple command line interface for commands to the spacecraft. However, even with the fault tolerance (Heart Beat) and networking (Socket Server) software added in to the Command Computer, it is much smaller than that of the XP team. Isolating the fault tolerance software is a requirements error, since this too should have been in a backup. However, the network software is independent of the simulation and probably should have been split out by the XP team.

Both teams demonstrated their software to the client, the XP team almost daily, since the frequent builds kept the software current; the TSP team only at the end of cycles. A purely subjective assessment is that the XP team’s software felt more “complete,” and that using it was more satisfying than that of the TSP team, a subjective judgment that the XP team did a better job in the same amount of time. Part of this assessment is due to the fact that the XP team had a more mature product, as indicated by size. Another part is that there was nearly constant client contact.

The teams had about 288 hr to implement the software and prepare the final report. This translates to overall productivity of 17.3 lines of code per hour for the XP team, versus 7.4 lines for the TSP team. This is a side effect of the methods used. The TSP status meetings are longer than the XP “stand up” meetings, and the TSP team wrote two versions of a requirement specification and design document, a development strategy document, a project management plan, a configuration management plan, and a vision document. These were also inspected, and their defects counted against the total since the method called for producing the documents as a way to reduce overall defect rates.

As Pete McBreen says in *Software Craftsmanship* (2002), the amount of documentation produced is a function of how involved the developers are in later implementations and maintenance. XP assumes that the programmers, craftsman-like, will remain with the software after release for enhancements. TSP takes the “what if the developers were hit by a truck approach,” so you would expect lower production of lines of code because of the much higher production of documents. The XP team decided to produce an integration plan, but that was the only formal document. In short, they were more “agile,” more value in programs than documents.

Overall code productivity was much greater for the XP team. This resulted in more actual defects than the TSP team. However, if we compared equally-sized products, the number of defects of the XP team would be less.
During the spring of 2002, we repeated the XP team’s performance on the spacecraft problem by giving it to four teams of industry-experienced graduate students and four teams of undergraduates. They averaged 10.3 defects and 1,827 lines of code per team. Size ranged from 885 lines to 5,142.

Part of the reason why XP teams continue to finish with fewer defects is simply that they have less opportunity to make mistakes. The teams that replicated the experiment stayed very close to XP practices and did no documentation save stories and the metaphor. Therefore, defects found in the inspection of documents did not happen because there were no documents. Whether the defect rate would continue to hover in the 10 lines per code range is dependent on whether the lack of documentation hurts maintenance.

In this study, lots of other things are going on besides pair programming and inspections. Thus, these factors are not isolated. Therefore, the study is limited.

**IMPLICATIONS FOR TEACHING**

The implications for teaching lie in choosing a method, and what to do about inspections. As for choosing a method, if the software is needed yesterday, and it is important to show significant progress quickly, and there is some expectation of keeping the team together for the life of the product, then XP is the choice. If the software is to be handed over to others to be maintained and if there is no particular rush to implement it, then TSP is a useful process. For example, once the author worked on the software for the Mission Equipment Package for the RAH-66 Comanche helicopter, finishing in 1988. The software is expected to fly for the first time in 2002. Even if the client had not already required one, a document driven method should be used, as few of the original engineers are still on the project. This would be true even if the project had not moved from Kansas to Delaware, further reinforcing the choice of method.

As to what to do about inspections: if XP, and thus, pair programming is the choice, the data implies that separate formal inspections do not have to be held. However, the use of pair programming could be easily incorporated into TSP, and likewise additional inspections could be an extended process in XP. In fact, Laurie Williams has defined a method of using PSP as the framework of a software development process using pair programming in her dissertation “The Collaborative Software Process (CSP).” So, this is a documented way to
incorporate pair programming into TSP. Research shows that a second inspection of the same material can have a positive effect (Porter et al., 1997), although the expense of doing them may not be acceptable except in life-critical applications. However, mandating inspections as an addition to XP would help mute the criticism that XP lacks enough rigor to be used for such applications.

NEXT STEPS

A further experiment comparing XP with TSP on a different project is currently underway in the Software Development Studio of the Master of Software Engineering Program at Carnegie Mellon University. The results are so far anecdotal, but the XP team had a working prototype the first semester, the only time in 13 years that that happened. The difficulty of the project was comparable to prior ones. The XP is essentially finished half way through the second semester, with the TSP team significantly trailing. In summary, XP appears to result in higher rates of defect reduction and productivity.

REFERENCES
