Put It to the Test: Using Lightweight Experiments to Improve Team Processes

Michael Keeling

Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213, USA
mkeeling@neverletdown.net

Abstract. Experimentation is one way to gain insight into how processes perform for a team, but industry teams rarely do experiments, fearing that such educational excursions will incur extra costs and cause schedule overruns. When facing a stalemate concerning the use of pair programming one industry-like, academic team constructing a commercial-grade web application, performed a lightweight experiment comparing pair programming and programming alone using Fagan inspection. Through the experiment, the team learned that pair programming was not only faster than programming alone, but also required less effort and produced code of more predictable quality. Conducting the experiment required only eight hours of effort over six weeks (a mere 0.5% of the total effort during that time frame) and afforded crucial information for choosing the best practices for the team. As demonstrated by this experience, lightweight experimentation is cost effective and does not threaten project schedules.

Keywords: experimentation, Extreme Programming, pair programming, process improvement, process tailoring, scientific method.

1 Introduction

When faced with the choice of continuing to use pair programming, the Square Root team was divided. Half the team was dead against continuing on grounds that there wasn't time to waste in the schedule by pairing up. They suggested we drop pair programming and adopt Fagan inspection [1], a highly structured peer review technique, instead. The other half of the team enjoyed pairing and wanted to continue the practice, pointing to research that showed it can be faster than programming alone [2] and can produce code of similar quality to Fagan inspection [3]. Of course the research cited, like many software engineering studies, was based on work performed by undergraduate students working on toy projects and the results may not be applicable to industrial teams. Polarized, the team chose to put pair programming to the test by conducting a lightweight experiment. This paper will tell Square Root's story and demonstrate how simple it is to set up lightweight experiments for validating intuition coming out of retrospective meetings.

A. Sillitti et al. (Eds.): XP 2010, LNBIP 48, pp. 287–296, 2010.
© Springer-Verlag Berlin Heidelberg 2010

2 Project Background and Context

The SQUARE project was completed as part of Carnegie Mellon University's Master of Software Engineering (MSE) program [4]. The MSE program uses a capstone element, the Studio Project, which allows students to practice concepts that were learned through coursework in a realistic project setting. The team consisted of five members, all students in the MSE program.

While the project was conducted in an academic setting, the client was considered a paying customer and the students were all experienced engineers. Team members began the program with two to five years experience and varied technical backgrounds including quality assurance, data analysis, project management, and development on desktop applications, web applications, and real-time distributed systems. The project lasted four semesters (16 months) with varying time commitments from 9 hours to 48 hours per week depending on the semester. The project had a strict timeline, a budget of 4, 800 engineering hours, and was monitored by two senior faculty members of the MSE program.

During the construction phase of the project, the team used Extreme Programming (XP) [5]. No team members had prior experience with XP; however most had used at least some of XP's practices in their previous jobs. At least one team member had experience with each of the following: Test Driven Development, refactoring, pair programming, creation and use of a coding standard, continuous integration, and incremental development. Team members also had experience with a variety of other software processes, most notably the Personal Software Process, the Team Software Process, Scrum, and the Rational Unified Process.

SQUARE (Security Quality Requirements Engineering) is a nine–step process for eliciting security requirements [6]. The result of the project was a commercial-grade, web-based tool currently used by the Software Engineering Institute (SEI) for research and education, and by business customers using SQUARE on real projects [7]. The client was a senior member of the technical staff at the SEI and is the principal investigator for SQUARE.

3 Framing Lightweight Experiments with the Scientific Method

During one of the team's iteration retrospective meetings, team members were divided about whether pair programming should continue due to schedule concerns. Since the team still desired some form of peer review, Fagan inspection was suggested as an alternative to pairing. As we were unable to decide whether pair programming or programming alone would be a better choice for the team, we decided to settle the dispute by collecting objective data.

Following the decision to conduct the pair programming experiment, two team members volunteered to plan and monitor the experiment. These team members became known as the "experiment champions." In planning the experiment, the

Step in the Scientific Method	Square Root's Application
Ask a question	Is pair programming efficient enough to finish the project on time with the desired level of quality?
Do background research	Projections based on research indicate that the schedule will be tight. Other research indicates Fagan inspection and pair programming are about the same in terms of quality.
Construct a hypothesis	The Square Root team should be able to approximately achieve results from previous, academic experiments.
Test the hypothesis in an experiment	Use GQM to identify data, metrics. Identify and mitigate risks introduced by the experiment. Divide work into test groups so the groups are roughly equal for comparison.
Analyze data, draw conclusions	Analyze collected data and calculate metrics identified with GQM.
Report results	Present results at team retrospective meeting. Discuss how the team will improve processes based on the results.

Fig. 1. Summary of Square Root's experiment

champions fell back on a key lesson from grade school: the scientific method, a means of inquiry in which objective data is collected through observation to prove or disprove a hypothesis (see figure 1).

Since the greatest sources of apprehension surrounding pair programming concerned schedule and quality, we chose to focus on those areas in our experiment. To make these ideas measurable, we turned them into a series of hypotheses to be proved or disproved through data. We hypothesized that the team would be able to approximately reproduce the results of previous academic studies which examined pair programming [2] [3]. Specifically that

- Pair programming produces code of similar or better quality than individual programming with Fagan inspection,
- Pair programming requires 15 – 80% more effort to complete a feature than individual programming on features of similar size, and
- Pair programming requires 60 – 80% less calendar time to complete a feature than individual programming on features of similar size.

Why these hypotheses? Since we had the information from prior research, we felt it would be better to have a more precise benchmark for comparison. Less precise hypotheses, (e.g. pair programming will require more effort but be faster) would likely have worked just was well. The point of having a hypothesis at all was to create a catalyst through which we could understand what data needed to be collected.

3.1 Data Collection

The experiment champions used the Goal Question Metric (GQM) approach [8] to identify metrics necessary for measuring the outcome of the experiment. In the spirit of XP, metrics were kept as simple as possible. To assess quality we decided to count the number and type of issues discovered through inspection and pair programming as well as the number of defects discovered during acceptance testing. Issue and defect data was normalized by size, in this case method lines of code. To assess time and effort, we examined the number of hours spent developing features.

Balancing data collection and agility was a primary concern, so we strove to use as much of our existing data collection methods as possible. We needed enough information to objectively evaluate our hypotheses but did not want to negate the agility of XP. As the team was already using Microsoft SharePoint to track tasking effort, we simply added a checkbox to the task form to indicate whether task time was executed alone or as a pair.

While time and effort information was relatively easy to collect, defect data turned out to be more challenging. Since XP does not give specific guidance for classifying defects, the team borrowed defect classifications from the Team Software Process [9]. The greatest challenge was determining how to fairly compare the effectiveness of Fagan inspection to pair programming. Comparing the number of defects discovered during acceptance testing is straightforward, but only addresses functional defects. One of the greatest benefits of peer reviewing code is that it helps uncover issues that might otherwise go unnoticed, such as coding style or design issues. The Fagan inspection process naturally captures this sort of information, but pair programming is designed to eliminate interruptions and decrease the amount of time necessary to execute a code-inspect-fix cycle. Issues

Programmer 1: Marco　　　Estimate (hours): —

Programmer 2: Michael　　　Estimate (hours): —

date: 7-16-2009　Start Time : 5:36　　End Time: 7:39

Milestone/Feature: Create Project　　Task Points: — (Low/Medium/High)

Task Description: Dialog Boxes

Data / Documentation	Syntax	Build / Package	Assignment	Checking	Interface	Function	System	Environment
Comments, coding standard/style	Things that don't compile	CM, file maintenance, package stuff	Declarations, duplicate names, scope	Business rule violations, verify inputs	I/O, Method calls and references	Logic, algorithms	Performance, Timing, Java, design	Compile, test, hot keys, support
	卌		卌	卌	II	III		I

Instructions: Record defects in real-time according to the type of defect detected by marking the appropriate column. The current co-pilot should record the defects. This paper should be traded for the keyboard and mouse between the pair.

Fig. 2. A tally sheet used to record issues discovered during a pair programming session. Each tick mark is an issue discovered by the co-pilot.

introduced during a pair programming session should not be present in the final artifact. Thus, we needed to collect data concerning caught issues through pair programming in real time. To accomplish this, the champions created a simple issue tally sheet (figure 2). The co-pilot recorded caught issues on the tally sheet as the pair worked. The tally sheet was traded for the keyboard and mouse throughout a pairing session. Using the tally sheet turned out to be a fun and effective way to collect data, and helped keep the co-pilot engaged throughout a pairing session.

4 The Experiment Work Plan

Normally scientists create independent test groups (e.g. control and experiment) to isolate variables in an experiment. While it would be ideal from a scientific perspective to have two teams building the same software project, one using pair programming and the other programming alone, for obvious business reasons this ideal scientific environment doesn't make sense. Rather than paying two teams to build the same software, we divided the remaining project work into two test groups: programming alone and pair programming.

The team agreed unanimously that our quest for information should not prevent us from shipping working software to our client. To ensure that the experiment did not negatively impact our ability to ship, we explicitly managed risks associated with the experiment using methods pioneered by the SEI [10].

The greatest risk we identified concerned the effectiveness of pair programming. There were doubts among the team as to whether pair programming would allow us to ship on time. To mitigate this risk, rather than apply pair programming exclusively during an iteration, we implemented features using both pair programming and individual programming with Fagan inspection during the same iteration. This way, if pair programming negatively impacted the project, we retained our ability to ship at least part of the desired features for that iteration. If two iterations in a row failed to ship promised features, the experiment was to be immediately terminated.

In order to maintain the desired test groups for the experiment and mitigate the pair programming risk, we modified our planning process as shown in figure 3. At the beginning of each iteration, our client chose the features to be completed during an iteration using XP's planning game. We articulated features as use cases and estimated the relative size of each use case with use case points [11]. Before starting the iteration, each of the picked features was assigned to one of the test groups based on point values so that each test group had approximately the same number of points. Because we used points as an estimate for size, it was easy to divide the work for each iteration into test groups. To complete the analysis, we used the actual size of developed features since the estimates may not accurately reflect how much code was actually created.

The experiment work plan was stored in the team's wiki alongside our other team processes so it would be highly visible and easily accessible. The experiment wiki page included the hypotheses, the GQM analysis, complete instructions for

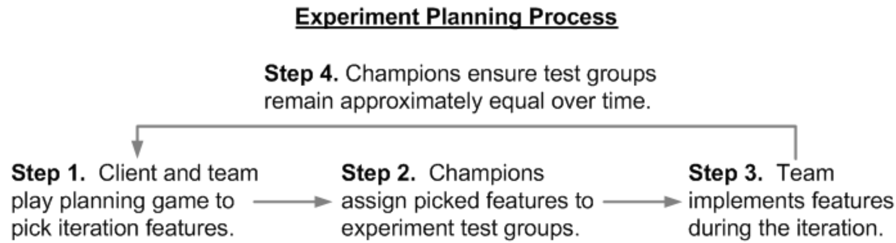


Fig. 3. The modified team planning process for accommodating experiment test groups

data recording, the experiment work plan, and definitions for what it meant to "pair" or "work alone." Real time pair programming tally sheets were collected daily and the raw results published on the team's shared documents repository. The updated experiment feature list was also made publicly available [12].

5 Experiment Results and Discussion

The experiment required a total of three iterations to complete. Preliminary results were discussed during the team's iteration retrospectives, providing immediate feedback. Complete results[1] were presented to the team at the conclusion of the experiment.

5.1 Observations about Software Quality

We found that code produced using pair programming allowed slightly more defects to escape to acceptance testing than programming alone (figure 4). Reflecting on the results during our retrospective meeting, one team member noted, "I feel like I am more focused on finding issues during inspections than I am while pair programming." Other team members agreed that the focus when pairing is on writing working code, not finding issues. This intense focus on functionality could explain why more defects escaped when pairing.

We also observed that pair programming had a higher issue yield according to our real time statistics than Fagan inspection. While this may seem contradictory (since more defects escaped with pair programming), this observation is likely a byproduct of the real time issue tracking. Specifically, since tally sheets were only used to measure pair programming in real time, we don't know how many issues were uncovered and fixed during solo programming before an inspection. Certainly some issues are uncovered and fixed when programming alone. In addition, certain issues, such as those related to the environment, simply can't be uncovered during a Fagan inspection.

[1] Though the experiment proved to be extremely valuable for the Square Root team, the specific experiment results and our conclusions on pair programming and inspection may not apply generally because the sample size consists of only a single team.

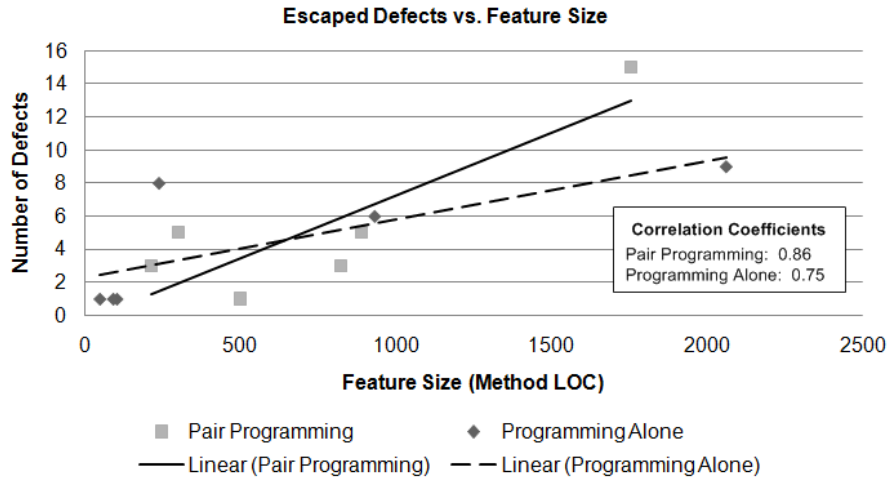


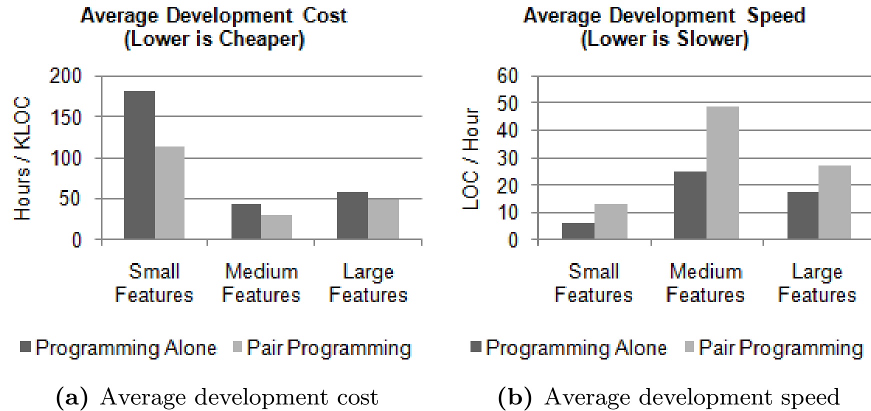
Fig. 4. The correlation coefficient was 0.86 pair programming and 0.75 for programming alone

The team was pleased to discover that code produced using pair programming had more predictable quality as a function of size (4). In other words, though slightly more defects escaped to acceptance testing, we could better predict how many defects would exist. This is likely because programming in pairs overcomes individual biases in experience, mood, concentration, and other variable facts of day-to-day life that might cause quality to differ when programming alone. The team agreed that having a partner helped maintain focus on the work.

5.2 Observations about Effort and Schedule

We hypothesized, generally, that pair programming would cost slightly more but allow us to develop code faster. Much to our surprise, these hypotheses turned out to be wrong, in a good way. As hypothesized, pair programming allowed us to develop code in less time. Surprisingly pair programming also required less effort in most cases. Specifically we found that pair programming required 11% – 40% less effort than individual programming with Fagan inspection. Dropping inspection altogether, pair programming would have required between 26% less and 12% more effort than programming alone, less than hypothesized. In terms of calendar time, we found that pair programming required 54% – 62% less calendar time to complete a feature than programming alone, slightly less than hypothesized. This data is summarized in figure 5.

With this information in hand, there were no doubts that pair programming would have an extremely positive impact on our schedule, and that without pair programming we may not have been able to complete the project on time!



(a) Average development cost **(b)** Average development speed

Fig. 5. Summary of effort and speed results. Pair programming consistently out–performed programming alone in both cost and speed.

5.3 Additional Observations

In addition to the hypotheses we set out to test, the team noticed other interesting things about pair programming throughout the course of the experiment. Pair programming is popularly touted as an alternative to inspection in which all code is peer reviewed; however we did not find this to be true. The standing order in our experiment was for features from the paired test group to use "100% pair programming." Recognizing that this was not realistic, we set an experiment threshold stating a percentage of effort that must be spent pairing for a feature to be considered "paired." In spite of this standing order, pairing never consumed more than 95% of the effort and averaged only about 85% of the effort for a paired feature. Prior to the experiment, pair programming was encouraged, but the team paired for only 60% of the effort for a given feature when we did pair. No matter what peer review technique is used, peer reviewing all code is difficult to achieve.

Knowledge sharing is another commonly known benefit of both Fagan inspection and pair programming, and the team made several observations about this as well. Generally the team felt that Fagan inspection was better at sharing high-level information, such as design strategies and systemic architecture concerns, while pair programming was better at sharing low-level, detailed information related to the environment, programming language, and programming in general. Team members also observed that it was more difficult to fix issues discovered during Fagan inspection due to a lack of context. One team member noted at our iteration retrospection meeting, "I hate going back and fixing the issues we discover during inspections. The issues come up days after I've banged out the code and it takes me forever to remember the context." Indeed, inspection meetings always lagged behind the completion of a feature to allow time for the inspection team to prepare.

6 Conclusion

Once the experiment concluded and the data was analyzed, the team chose to stick with pair programming. Though pair programming allowed slightly more defects to escape on average, predictability and consistency were more valuable. In terms of cost and schedule, pair programming was the clear winner. Setting up and executing the experiment turned out to be an easy and fun way to resolve an otherwise difficult conflict by turning a battle of wills into a comparison of data.

It took two people one hour to plan the experiment, a negligible amount of effort to adjust the experiment plan between iterations after the planning game and to collect data as we worked, and two people three hours to analyze data once the experiment concluded. All told, this accounts for only 0.5% a percent of work conducted over three iterations. This time investment was a small price to pay to validate gut feelings, boost team confidence through knowledge, and settle process disputes with data.

In executing this experiment there were a few key ideas that helped us keep the experiment light. The experiment itself was relatively small. The variables under test were kept to a minimum. Rather than comparing whole processes we compared only two practices. We chose to focus on a small handful of related hypotheses. We leveraged our existing data collection practices heavily. We planned the experiment so that it would require only a few iterations to complete, though in hindsight it would have been better if the experiment had ended after two iterations (four weeks).

Scientific thinking, scaled down, is an excellent way to gain insight for team retrospectives, but hard data is no substitution for team discussion and group reflection. Science is a method of discovery. Teams should consider running experiments to learn more about the processes they use and how those processes really work for the team. This sort of intrinsic curiosity is healthy for teams and might even strengthen work in other engineering areas. Further, lightweight experimentation is an excellent means for guiding continuous process improvement.

Given the inherent gap between research and industry, publishing experiment results in the form of whitepapers and blog posts would benefit the software industry as a whole. Think about how great it would be to see results from other industry teams experimenting with design practices, unit testing versus inspection, user stories versus use cases, or estimation techniques. As this experience demonstrates, statistical significance, strict control groups, and lofty academic goals are not necessary to make great gains in knowledge. All that's needed is a touch of science.

Acknowledgments. I would like to thank my fellow Square Root teammates (Sneader Sequeira, Marco Len, Yi-Ru Liao, and Abin Shahab) for their work on the SQUARE project. I would also like to thank our studio mentors, Dave Root and John Robert for their guidance throughout the project.

References

1. Fagan, M.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15, 182–211 (1976)
2. Williams, L., Kessler, R.R., Cunningham, W., Jefferies, R.: Strengthening the Case for Pair Programming. IEEE Software 26, 19–25 (2000)
3. Phongpaibul, M., Boehm, B.: A Replicate Empirical Comparison between Pair Development and Software Development with Inspection. In: First International Symposium on Empirical Software Engineering and Measurement, pp. 265–274. IEEE Computer Society, Washington (2007)
4. Garlan, D., Gluch, D.P., Tomayko, J.E.: Agents of Change: Educating Software Engineering Leaders. IEEE Computer 30, 59–65 (1997)
5. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston (2000)
6. Mead, N.R., Hough, E.D., Stehney, T.R.: Security Quality Requirements Engineering (SQUARE) Methodology. Technical report, Software Engineering Institute (2005)
7. SQUARE Tool, `http://www.cert.org/sse/square-tool.html`
8. Basili, V., Caldiera, G., Rombach, H.: The Goal Question Metric Approach. Encyclopedia of Software Engineering 1, 528–532 (1994)
9. Humphrey, W.S.: Introduction to the Team Software Process. Addison-Wesley, Boston (1999)
10. Nelson, C.R., Taran, G., Hinjosa, L.d.L.: Explicit Risk Management in Agile Processes. In: Agile Processes in Software Engineering and Extreme Programming, pp. 190–201. Springer, New York (2008)
11. Cohn, M.: Estimating with Use Case Points. Methods and Tools 13(3), 3–13 (2005)
12. Square Root Project Archive and Experiment Data Set (available indefinitely), `http://dogbert.mse.cs.cmu.edu/MSE2009/Projects/SquareRoot/`