Promiscuous Pairing and Beginner's Mind:
Embrace Inexperience

Arlo Belshee
Architect
Silver Platter Software
Pasadena, CA 91103
(503) 265-1263
a_xp@arlim.org

Abstract

Many traditional software practices stress the importance of programming in Flow. XP directly challenges the assertion that Flow is critical and proclaims Pair Flow.

Both Flow states are fragile. They are easily disrupted by outside distraction or task rotation. Both take a long time to enter. Furthermore, it takes days for a given pair to be comfortable enough with each other to be able to achieve Pair Flow at all.

My team at Silver Platter discovered that there is a third option to achieve high-efficiency programming. Our team spent the majority of its time in Beginner's Mind. Whereas Flow depends on stability, Beginner's Mind depends on instability, yet provides similar efficiency gains to a constant state of Flow.

This paper discusses one approach to achieve a constant state of Beginner's Mind. It shows how to use those most-central of agile programming practices — pairing and task allocation — to constantly reinforce this mind state.

1 Introduction

1.1 The Project and Environment

Ours was a reasonably typical enterprise distributed networking project. It consisted of around 1,000 C++ classes in 60 different program executables. Each customer deployment supported 10,000 to 500,000 clients. The system involved around a dozen different types of servers — accounting, system watchdog, state maintenance, and so forth. Security and reliability were the paramount concerns. Performance was second, and features were a distant third.

The company was a startup, so we were tight on both cash and time. The company was typically operating with between -30 and 180 Days 'Till Broke. Our contracts all had lead times of 3-5 years. This meant that sales had to start at the same time as engineering. Thus, engineering had to produce many sales demos and to frequently alter the product to more closely fit the needs of a particular customer.

Due to these influences, we chose a software process with rapid feedback and change. We ran the shortest iterations we could (1 week) to get the most data possible. We tracked our metrics closely, and we ran several experiments each iteration. We used the metrics to decide what worked and to what degree. We then adopted those things that worked and started the next set of experiments.

Chief among these experiments were variations on

- How to handle task ownership,
- How to assign tasks to people, and
- Which style of Pair Programming to use.

1.2 Flow and Pair Flow

The Flow mind state [1], [2] is one of intense focus. The entire problem and solution spaces are loaded into the developer's head. Programmers work orders of magnitude better when in Flow.

Pair Flow is similar to Flow. The solution and problem spaces are shared between the minds of the participants. Again Pair Flow works significantly better than pair programming without flow.

Unfortunately, it takes a long time to get into either Flow state. Both can be easily interrupted. Changing tasks or swapping pairs forces a restart. Pair Flow is more resilient

to interruptions such as the phone, but it still gets interrupted frequently throughout a typical day.

It often takes days for a given pair to be comfortable enough with each other to be able to achieve Pair Flow at all. This means that pairings tend to be long. The longer the mean time between pair swaps, the less effectively pair net distributes information through the team.

1.3 Beginner’s Mind

> “In the beginner's mind there are many possibilities, but in the expert's there are few.” [3]
>
> It is the open mind, the attitude that includes both doubt and possibility, the ability to see things always as fresh and new. It is needed in all aspects of life. Beginner's mind is the practice of Zen mind.

Perhaps first described by Zen Buddhists in its relation to No Mind, Beginner’s Mind is a state of few limits.

Beginner’s Mind is distinct from, but interrelated with, No Mind. Beginner’s Mind happens when the thinker is unsure of his boundaries. The thinker opens himself up and thoroughly tests his environment. No Mind is a meditative state in which the practitioner leaves behind all the dreck in his life, allowing himself to just be.

Modern psychology distinguishes between the two because Beginner’s Mind can also be experienced outside of No Mind meditation. In fact, most people automatically assume it when they are placed in a situation outside but near the limits of their comfort zone. If a person is otherwise comfortable with his environment but doesn’t understand one thing, then he will usually try stuff until he figures that one part out. This state of trying to reconcile one’s past experiences with an environment that doesn’t quite fit is Beginner’s Mind.

Beginner’s Mind is the key behind the phenomenon of Beginner’s Luck: a person doing something for the first time often does it much better than he does after he’s practiced for a while. Because he tries more approaches, and tries them rapidly, a person in Beginner’s Mind is more likely to succeed at a task than one who thinks he understands how it works.

My team at Silver Platter discovered that Beginner’s Mind is a very efficient way to solve programming problems. However, Beginner’s Mind is generally a transitory state. As soon as a person has figured out the bounds of his current situation, he tends to drop to a lower-energy cognitive state.

Whereas Flow depends on stability, Beginner’s Mind depends on instability. We found that Beginner’s Mind can be maintained as a stable state by simply changing things around frequently enough — by surfing the edge of chaos.

1.4 Competencies Versus Skills

One of the key insights of Behavioral Interviewing [4], [5], [6] is that there is a difference between competencies and skills. The difference is simple. People can learn skills in a matter of months. People can’t learn competencies in less than several years. There aren’t many things that fall between — qualifications are almost always skills or competencies.

When organizing our team, this means that we needed to treat the two categories differently. Skills were transmitted extremely quickly around the team because we spent most of our time in Beginner’s Mind. We therefore assumed that any skill could be supplied by any member of our team.

Competencies, however, are unique to an individual. Many of them are even mutually contradictory. For example, it is difficult to find someone who is both Creative and good at Following a Process.

Behavioral Interviewing supplies us with around 30 such competencies. Every task requires more than one. Most tasks require three or four and could take advantage of a half-dozen or more. It is often impossible to find such a set in any two people, much less in any two people on the same team.

This understanding leads to the realization that Beginner’s Mind can’t provide everything needed to increase pair efficiency. Distinguishing between competencies and skills caused us to experiment with task ownership and assignment processes. We found processes that let us apply Beginner’s Mind to provide the skills, but still apply each competency when it was needed.

1.5 Promiscuity and Pair Net

Pair net is an effective means of knowledge transfer. While two people are paired, they share knowledge. When the pair splits for a pair swap, the knowledge then spreads to all four participants. In this way, knowledge will slowly but automatically spread around the group.

This knowledge transfer is automatic, and includes anything which comes up while a pair is working. The resulting network, pair net, tends to filter information for that which is used by the most people. The most useful information spreads the fastest.

In general, the most useful information gets passed in every pairing, and nearly all information is passed in a matter of a few pairings. Most of this information is passed in the first hour of a pairing.

As such, the primary limit on the rate of transmission is the number of different people that each person pairs with each day. It pays to be promiscuous.

2 Practices

2.1 Introduction

We discovered our practices by running experiments and analyzing the data. We came up with several options for each category. We then tried each one for an iteration or two and analyzed our metrics. Our primary metrics were velocity[1] and red card rate[2].

Only later did we explain why the chosen practices outperformed their competitors. Looking at the winning practices, and feeling the way the team operated, the ties to Beginner's Mind, competencies, and so forth are obvious. After we had discovered this trend, we used it to predict likely successful future practices. However, our adoption process was deductive, not inductive.

2.2 Give Tasks to the Least Qualified Person

There are three general strategies for deciding who works on which tasks: assign them to the most-qualified person, assign them irrespective of skill, or assign them to the least-qualified person. We tried all three approaches.

Interestingly, these data showed an overall increase in velocity when tasks were consistently assigned to the least qualified person. The difference was especially marked over long periods. Choosing the least-qualified strategy really pays off after the team has used it for several iterations, but outperforms the others even in the first iteration. The data on red card rate corresponded with those on velocity: the least-qualified teams produced the code that had the fewest surprises.

We didn't run these experiments while we were hiring. Therefore, we don't have data correlating task selection approach and ramp-up time. However, we assume that the least-qualified selection strategy also helps with new-hire ramp-up time as it leads to the fastest propagation of skills.

2.3 Task Naturals, not Domain Experts

Giving tasks to the least qualified person plays strongly into Beginner's Mind. However, the selection needs to be wary of the distinction between competencies and skills. The optimal worker for a task is the one that is the least skillful in that task, but who has any necessary competencies. A worker who lacks a required competency will not perform the task well, regardless of skill level — and will not enter Beginner's Mind.

Selecting implementers who are least-qualified decreases the ability of a team to develop domain experts. This is actually a good thing. Instead of domain experts, the team tends to develop task naturals.

The difference between a domain expert and a task natural is exactly the difference between a skill and a competency. A domain expert is the person on a team who is best at a skill. A task natural is someone whose competencies and interests align with a particular type of work — such as data modeling, bug hunting, or testing.

Skills and experience within a particular domain are easy to measure. Because of this, they tend to become the primary metrics that people use to determine who is "most qualified" to do a task at hand. By selecting against skills, we distributed experience around the team. Soon, everyone had the skills to work in any problem domain involved in our project.

However, once skills are roughly equivalent, it quickly becomes visible who has what competencies. Talents tend to crosscut domains — when working on a typical system you might want a natural domain modeler, a tester, and a task simplifier, for example.

The need for different talents becomes especially visible on bugs. These often require a large number of talents applied in a particular order. Each talent helps get around one problem, but a different talent is needed for the next problem. Even if there is another problem in the future that will need the same talent as the one just solved, the team can't work on it. The next problem can't be seen until the current problem is resolved.

> Our most heinous bug required many talents to fix. It resulted from the compiler implicitly choosing two different calling conventions on the two sides of a DLL boundary. As a result, both the function and the code calling it tried to pop the function arguments off the stack. The system would repeatably crash in some totally unrelated code.
>
> The system called the death function, which returned. The system appeared fine, but the stack and frame pointers were computed. The calling method then called something else with its bad frame pointer. This propagated through the call stack until something eventually dereferenced a pointer held on the stack. That variable's offset was actually located in compiler-generated frame storage for a register whose value was NULL, resulting in a crash.
>
> To solve this bug, we needed our best data analyst. We needed our compiler guru. We needed our expert tester. And we needed our most creative guy — he was the only one who could come up with more

[1] We estimated each task in arbitrary effort units (CU). Our velocity was the number of CU completed per pair-week. This took into account changing headcount, and vacations. We applied an experimentally determined "ramp up factor" when we were adjusting to a new hire.

[2] Any code-related task which was not on the board at the end of the planning game was put on a red card. These included bugs, any refactoring that took more than 15 minutes, unanticipated dependencies, and the like. Our red card rate was red CU for the week / total CU for the week, expressed as a percentage.

things to try. So we swapped frequently, rotating as soon as the task changed.

In 18 months of C++ development, this bug was our hardest challenge. It represented the longest time that we failed to make forward progress.

We put one pair on it. It took us 6 hours.

2.4 Team Owned, Pull-based Task Assignment

Who is responsible for completing which tasks? Different XP teams have different answers. Again, we tried several approaches and used metrics to discover which worked best for our team.

We attempted individually owned tasks and team owned tasks. For each grouping, we attempted push-based and pull-based assignment. For individually owned tasks, we tried both just-in-time and per-iteration assignment periods. This results in a total of 6 combinations.

An individually owned task is one that a single individual is responsible for completing. He will pair with others to work on it, but he will never rotate off it and is personally accountable for its completion. In contrast, team-owned tasks are the responsibility of the team as a whole. Anyone can work on them at any time.

In push-based assignment one person delegates tasks to others. This may be a manager, a Senior Engineer, an architect, or some similar person. In pull-based assignment, people grab tasks that they want to work on off of a shared space, such as a cork board. They tell the team why they should do the task, and then take it.

The data below clearly indicate that more flexible work assignments got more work done. The most efficient method was team owned tasks with pull-based assignment. Again, this gave marked improvements in both our velocity and our error rate. The difference was especially marked among the larger tasks — tasks that took over ½ a pair-day were completed much more quickly and predictably when they were team owned.

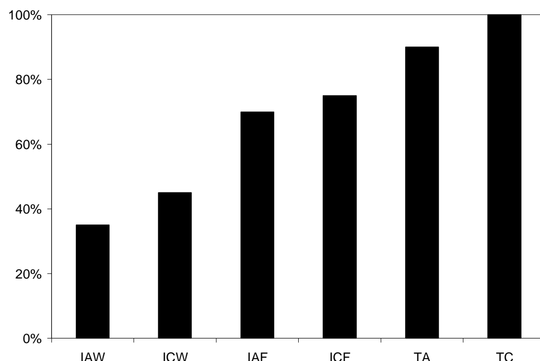

Figure 1. Task ownership and velocity

The letter codes for the categories are (T)eam owned vs (I)ndividually owned, then (A)ssigned vs (C)hosen, and finally (W)eekly vs (F)lexible duration of accountability. For each category, a mean velocity was measured across a couple of iterations. Velocity has been normalized to a percentage of the highest mean and the columns arranged in ascending order.

The worst two columns are those with weekly task assignments. All team-based assignment methods beat all individual accountability approaches. The error in these numbers is about 10%, which means that the 70% difference between the worst and the best methods is quite meaningful. The 5% to 10% difference between pull assignment and team leads is not.

2.5 Team Owned Tasks

The more flexible the work assignments, the more a team can take advantage of its natural talents. Many tasks are best solved by the combinations of 4, 5, or even more talents. It is very rare to find a set of people from whom you can pull any pair necessary for any problem. However, it is quite possible to pull five people who combine to give the five necessary talents.

Team based responsibility allows you to do exactly that. Because no one individual is tasked with finishing the problem, each person can be switched in to a given task only and exactly when required. This means that each person spends more time applying their specialty to your tasks at hand and more work gets done.

Furthermore, people who natively think in the most appropriate way to solve a problem tend to develop better solutions. This results in a measurable difference in bug rate. Finally, programming is more fun when each person spends more time applying his talent.

Pull-based task assignment was advantageous for the same reason. Although we all had a very good idea what the talents of our team members were, each person still knew himself best. By using pull-based assignment, each person could argue where his talents applied to the problem. We could also take into account day-to-day effects, such as sleep deprivation or romantic problems.

Our team also believed strongly in the concept of the working manager — our nominal manager spent about 60% of his time writing code. Management tasks were just more things that the team had to do. Because of this approach, we ended up without a domain expert in management, but with several naturals at the various parts of the job.

We had three people who were the team's ultimate spokesmen. Our planning and experimentation were done by anyone on the team, but there was a natural who stepped up to help when it was needed. Similarly, we had

naturals in the behavioral competencies of Energizing, Leadership, and Team Building. All of these were different people.

2.6 Pair Churn

It is generally assumed in the XP community that pairs are assigned on a per-iteration basis. I have spoken with members of teams that experimented with per-week or per-two-day assignments. There aren't many of them. We decided to experiment with pair duration.

From our informal straw poll, it appeared that few teams had tried short pairings. We had a gut feeling that these might be worth trying. So we tried pair durations of 1 hour, 90 minutes, 2 hours, ½ day, 1 day, and 3 days.

These smaller times were shorter than our task length at the time at which we performed the experiment. Therefore we needed some way to decide when to swap. We instituted the dreaded egg timer. When the timer went off, any pair who had not changed since the last time the timer went off had to change.

Additional pair swaps sometimes happened between scheduled times. These additional swaps were encouraged, but not very common. Most pairs would fall into their task and forget to swap until the buzzer went off — even if they completed a subtask or ran into a need for someone else's talent.

We tried two methods of performing a pair swap. In the first, one member of the initial pair stayed with a task until its completion. In the second, whoever had been with a task longer switched away to a new task with every swap. In the second approach, an individual would swap on to a task and work as a beginner, stay during the next swap acting as a teacher for the next new person, then swap away.

Pairing strategy had the tremendous long-term effect on our productivity. Therefore we gathered extensive data on approaches to pair swapping. Rapid, alternating swaps achieved peak velocity.

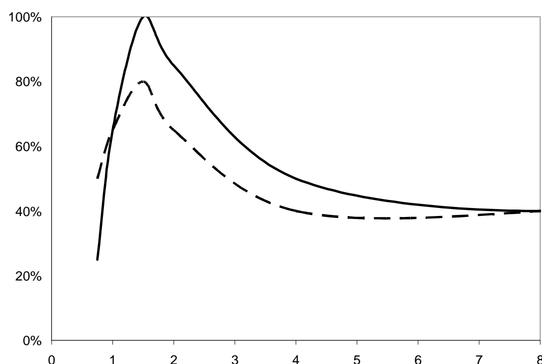

Figure 2. Swap rate and velocity

The above chart shows how mean velocity varied with pair swapping technique. The velocities have been normalized to a percentage of the highest mean velocity. The x-axis shows the number of hours between pair swaps. The dotted line shows pair swapping where the expert stayed; the solid line shows alternating pair swaps

At short pair intervals it is better to develop an expert, because that is the only way to transfer knowledge from one pair to the next. For intervals longer than an hour, alternating pair swaps work better. It apparently took our team approximately an hour to transfer sufficient knowledge to develop a new domain expert.

The graph is compressed to show only pair swaps up to 1 day. This makes it easy to see the shape of the curve near the 90-minute optimal point. However, we did note that longer pair times had slightly higher mean velocities. Our data suggested that 4 and 8 hours were bad, but longer times rose to just above 50% of our peak. Both curves appeared to flatten out at times over 1 day, at which point there was no measurable difference between them.

We took these measurements when we were a team of 6. We took a similar set of data points when we had grown to 11. The curve had the same shape, but it had shifted right by ½ hour. The crossover point was at 90 minutes, and the peak at 2 hours. We hypothesize that this was because each person had been away from any given part of the code for longer, so it took a little longer for us to get up to speed.

The curve did not, however, stretch out. It just shifted right. This indicates that neither total team size nor unfamiliarity with a given portion of the code base had any effect on a person's velocity once he had gotten up to speed.

> The final nail in the coffin of long pairings, however, came when our weakest coder went on a three-day vacation. At the time there were only three people on the team. With one person on vacation, we could no longer swap. Because it would only last for a couple of days, we assumed the effects would be minor.
>
> Not thinking much of it, we attempted to maintain the same intensity and velocity that we had the week before the vacation. Since 100% of our code was pair-written and we had only had one pair before, we didn't think we would notice that much difference. Besides, we still had the pair that performed best together.
>
> We were wrong.
>
> By the end of the first day we were exhausted. On the second day, we made it a couple of hours before we needed to call a rest. Even after the break, we couldn't maintain the velocity that had been so easy

before. We spent half of the third day neither paired nor programming.

By the time our missing teammate came back on Thursday, we were both begging for a pair swap. We then went right back into 90 minute swaps. We had fully recovered by noon on Thursday, and finished the week out at our previous velocity.

2.7 Continuous Beginner's Mind and Creativity

When people are in Beginner's Mind they learn faster and achieve more. Similarly, people tend to be more creative when they only partially understand a situation. Because they don't know all of the limits yet, they don't have as much difficulty seeing past them.

Pair churn ensured that every pair had a member in Beginner's Mind at all times.

In addition to gathering the metrics, we also asked people how they'd felt under various approaches. One of the most commonly stated responses was that the swaps were too frequent. It took people about 90 minutes to get fully up to speed on a new problem, and then they'd get swapped away. Most people felt like they were constantly drinking from the fire hose, unable to catch up.

We talked about this in a couple of weekly retrospectives. We discussed Beginner's Mind. After a couple of weeks, everyone saw how much more they were learning than they had in any other situation in their lives. The fire hose became a thrill ride. It became a challenge.

We found that with pair swaps below 90 minutes, some information was lost with each pair swap, requiring the new pair to frequently ask questions of the person who had just left the task. With longer pairings no information was lost, but after 90 minutes the pair's velocity dropped notably.

On a related subject, teaching is a great way to learn. This is especially true if the teacher is relatively new to the subject.

Alternating 90 minute swaps caused each pair to contain one person in Beginner's Mind and another who was teaching the subject he'd just learned. This strategy proved to be a phenomenally effective combination. It strongly outperformed the situations where we developed an expert by leaving a person on the task for a longer duration.

Pair churn also maximizes the effect of pair net. Information flows better. Everyone masters tools faster. Everyone learns how the data are organized in each system faster. Everyone learns new coding techniques faster.

One telling example of rapid pair net happened accidentally to the Silver Platter team. Around 10 am I was driving. While doing a bit of copy and paste, I accidentally hit Ctrl + Shift + V instead of Ctrl + V.

In Visual Studio, Ctrl + Shift + V operates a paste stack. It remembers everything that you have copied in the past. Pasting inserts the top of the stack. Pressing the keys again before doing anything else replaces the just-pasted stuff with the next item in the history. You can continue to pres the key combination to go back further in your history. This makes it easy to copy from a couple of sources at once and paste them all together.

My partner and I noticed this and spent a few minutes figuring out what the weird behavior was. We then went on with our work. Over the rest of the day, we swapped as normal. Once in a while, the paste stack would be useful, so I'd teach it to my partner.

Around 4 that afternoon I was again driving. My navigator saw me doing some copy and paste and took the keyboard to show me a neat trick — the paste stack. I was surprised that he'd seen it, so I stood up and asked the bullpen how many of them knew about the paste stack.

All 11 people had learned about it that day.

3 Results

3.1 New Hires

One unanticipated side effect of this team environment was its effect on new hires. Over the course of one year we quadrupled the team size. Such drastic growth under extreme project and business pressure has been the death of several start-ups over the years, yet we didn't find it to be that big a deal.

Our most difficult new-hire ramp up occurred after we had fully adopted our process. The new employee had never before programmed in C++, nor had he ever heard of functional programming or performed OOP. We performed heavy template metaprogramming throughout our code base and had a system of around 600 classes at the time.

Furthermore, the new hire had a lot of enthusiasm, but wasn't very technically adept. He wasn't good at analysis and didn't really understand data. We hired him because he had a good knowledge of our customer's domain and a strong mathematical background.

The first week after we hired him, our velocity dropped, as expected. The second week, our velocity was back to where it had been before the hire. By the

end of the third week we had improved our overall velocity, and the new guy could do any task on the board. He could sit down with any of the rest of us on a part of the system he hadn't seen before, figure out how it worked and contribute. He'd pretty much figured out both functional and OO programming and could read a template metaprogram — commonly considered to be one of the most difficult aspects of C++.

In the fourth week, he was pairing with our next new hire during that hire's first week. He was confident and skilled enough to take any task off the board — even in a part of the code base which he'd never seen — and teach the new guy how it worked. Furthermore, the rest of the team had sufficient confidence in him to have no qualms about him taking on this challenge. No one even bothered to monitor his pairings with the new guy.

3.2 Pair Promiscuously!

Promiscuity, it turns out, is a good way to spread a lot of information through a group quickly. Rapid partner swapping ensures that a good idea, once envisioned, is soon practiced by every pair. Replacing individual accountability with team accountability empowers each person to do those tasks at which he excels — and allow someone else to take over for his weaknesses.

Each of our practices provided the team with more flexibility and better communications. More creative ideas were formed, and each idea was automatically disseminated to the entire team by the end of the day. Each person was expected to continuously learn what was happening and contribute in a very short amount of time. Working on this team often felt like drinking from a fire hose, but it was empowering.

The data show that we were more productive the more promiscuous we were — as long as we remained with each partner long enough to exchange knowledge. What they don't show is that we also had a lot more fun. It took the team a little time to adjust to the more rapid pace, but working with that team was a career high point for every person involved.

4 References

[1] Mihaly Csikszentmihalyi. *Flow — the Psychology of Optimal Experience*. Perennial, 1991.

[2] Wikipedia. http://en.wikipedia.org/wiki/Flow_(psychology)

[3] Suzuki, Shunryu. *Zen Mind, Beginner's Mind*. Weatherhill.

[4] Training is available from Management Team Consultants, Inc. 415.459.4800.

[5] Lynda Ford. *Fifty Behavior Based Interview Questions*. http://www.fordgroup.com/january_2001_article.html

[6] Lynda Ford. *Are You Looking to Find Better Candidates to Hire? Try Behavior Based Interviewing!* http://www.fordgroup.com/article1.html