XP and Junior Developers:
7 Mistakes (and how to avoid them)

Richard Lawrence
Avanade
richardlaw@avanade.com

Abstract
In two years of leading agile projects at Avanade, an organization heavy in recent college hires, I’ve learned a number of lessons about how to be successful with junior developers on an agile team. Indeed, my most successful team has been my most junior.

I discovered that my biggest mistakes with junior teams were related to partially adopting XP. Where a senior team, by virtue of its experience, can adapt to replace missing XP practices, a junior team reveals the extent to which the XP practices form a whole and are best adopted together.

1. Introduction
Avanade, like many consulting firms, is structured as a pyramid with many more junior consultants than senior consultants. There are several reasons for this. It’s partly cultural; it’s easier to hire someone right out of college and grow them into the kind of consultant we want than to hire someone already set in their ways. But it also makes sense financially. Avanade makes a higher margin on junior consultants, and customers often prefer to pay less per hour for work that can be done by a junior resource.

At Avanade, we’ve always had teams with more junior than senior developers, but in the past, most of the college hires were testers on large waterfall projects, and I could easily form a team of mid-level and senior developers for an agile project. Recently, we’ve been hiring more aggressively at colleges and pushing to include at least one recent college hire on each project. As a result, I’ve gone from one or two fairly junior developers on most of my teams to my most recent team, half of which was right out of school with no development experience.

I had a unique opportunity to analyze the effects of various agile practices with junior teams. After over two years of leading agile projects with all mid-level and senior developers, I led two different teams at the same customer, largely on the same project and codebase. Both were approximately half junior, half mid-level or senior. The junior members of the second team were somewhat more junior, having never been on a professional software project in any role, while the junior members of the first team had previously been testers on other Avanade projects. I’ll refer to the teams as Team 1 and Team 2.

Team 1 started building the product from scratch and worked for six months to deliver the first major customer-facing release. Six months later, Team 2 adopted Team 1’s code. The first iterations were spent fixing defects that were discovered in production, mostly related to quirks in production data. Beginning with the second part of iteration 2, Team 2 developed new features for the product for the next four and a half months.

Using iteration and project retrospectives, we were able to identify some of the challenges associated with junior agile teams and to correct many of those in the second project. The second team was both the most junior and most successful team I have led in my career.

2. Mistake #1: Avoiding Pair Programming
In my experience report last year [1], I discussed the cultural changes associated with introducing agile at Avanade. I mentioned that I hadn’t been able to get pairing to catch on, but that I’d found alternatives in frequent code reviews and good tests. I was wrong.
I’ve always found pairing to be the most difficult part of XP to get teams to adopt. With Team 1, I approached pairing as an optional, if possibly beneficial, practice. It lasted less than one iteration before developers decided they would rather work alone. In our iteration retrospective, we discussed that since pairing didn’t catch on, we’d institute weekly code reviews and ensure that we had good acceptance tests for every story.

During the project, the code reviews and tests seemed to be working. The quality of the code under review improved from week to week, and each story passed its acceptance tests. But there was a cost that code reviews and tests failed to uncover. Some of the code produced by solo developers on Team 1 that made it into production was appallingly inefficient, difficult to maintain, and hard to understand. Allocating two hours per week to code reviews allowed us to review less than 20% of the code produced; full coverage would have taken more than a day per week. Thus, the defective code was only discovered later, when a feature broke in production or when Team 2 experienced it as technical debt when adding a new feature.

In the six month gap between the two projects, during which I had to fix several features in production and refactor some bad code, I recognized that pairing could have prevented some of these problems. I resolved that my next team would try harder to make pairing work. I had been tentative about proposing pairing with Team 1, believing that it would take hold better if I allowed team members to try it in small doses and discover on their own that it worked. With Team 2, I took the opposite approach. I imposed full-time pairing as a standard practice in the first iteration; if it hadn’t worked, retrospectives would have allowed us reduce pairing. I discovered that the benefits of pairing were most fully realized when it was done full-time.

By the second iteration of the next project, pairing had taken root on Team 2. The experience of mandated full-time pairing in iteration 1 was good enough that the team adopted pairing as its own standard. No production code was committed without having been created by a pair. At one point, one team member committed solo code that broke another pair’s functionality. The resulting team-wide conflict was resolved largely without my intervention, firmly establishing full-time pairing as a team norm.

Thanks to full-time pairing, quality went up dramatically. And in the project retrospective, the developers reported that pair programming was one of their favorite things about the project. Though it’s hard to measure, my sense is that we actually got more done in pairs than the same developers would have individually.

3. Mistake #2: Not Switching Pairs Frequently

Some of the key benefits we’ve seen with pairing are knowledge transfer and continuous code review. But unless we switch pairs at least once a day, those benefits aren’t fully realized. In the second iteration retrospective with Team 2, we observed that we were switching pairs as little as once a week and that the less we switched, the more defects were committed and the less other team members knew about the full range of stories in the iteration. While full-time pairing did provide continuous code reviews, pairs developed blind spots for their own code and defects made it into source control. At one point, the only developers who understood a critical feature were out of the office, and we realized that our knowledge sharing needed to improve.

We made a conscious effort to rotate pairs at least once per day. As a result, each member of the team understood the complete product better, developers gained more experience working across the various technologies used in the application, and fewer defects made it into the code repository.

4. Mistake #3: Letting the Architecture Emerge

With Team 1, I took the more hands-off approach to the architecture that I had used in the past with more experienced teams. Team members regularly discussed the emerging architecture and patterns, and I assumed that things were going well with the overall design. As it turned out, there were architectural problems throughout the application. These included duplicated or poorly factored code,
inconsistent application of patterns, and inefficient data access and management.

In one case, Team 2 found a pattern that they nicknamed “The Big Ugly.” A developer on Team 1, desiring to use a domain model rather than built-in .NET data objects, created an object hierarchy that used eager loading to create an in-memory representation of almost all the data for a facility. Then, he used that object hierarchy in a number of places in the application. If, for example, a report needed data for a single month, initializing the object still loaded the entire history of data for that facility. This worked fine in Team 1’s small test database. But by the time Team 2 joined the project, there were over 200 facilities across 6 years, and this object model created major performance issues. This technical debt slowed down Team 2’s work on new features.

Having identified “The Big Ugly,” the members of Team 2 allocated additional refactoring time to estimates of any feature that touched the flawed code. Our backlog worked out such that we were able to refactor “The Big Ugly” from the codebase while completing new features, never having to create a technical story dedicated to that refactoring effort.

With an experienced team and a fairly straightforward project, it may be possible to trust the team to make good technology decisions and let the architecture emerge. But with my junior teams, this didn’t work. To avoid architectural problems with Team 2, I assumed an “Agile Architect” role; I produced some code, but largely focused on guiding the emerging architecture and mentoring the team towards good technical decisions. By the end of Team 2’s effort, the overall design of the product had improved, facilitating a smooth handoff to the customer’s maintenance team. I’ve come to believe that even a senior team would benefit from having one person responsible for ensuring that a well-designed, coherent architecture emerges, and I plan to allocate time for this on future projects.

5. Mistake #4: Not Working in a Team Room

Since we were rarely pairing, I allowed Team 1 to work from home before 9:00 and after 2:00 each day. I found that developers were much less productive and less likely to ask for help when they were out of the team room.

Full-time pairing largely solved this problem with Team 2. However, the snowstorms that pummeled Denver this winter forced us to work from home a few times. Even once we solved the problem of how to pair remotely, pairs were still less productive because they couldn’t easily communicate with the rest of the team. Had we worked remotely for more than a few days, we would have delivered much less business value for the customer’s budget.

On both teams, I observed that the more senior developers were more likely to reach out for help when working apart from the rest of the team. Junior developers would “go dark” and waste time on issues that would easily have been resolved with communication. In the team room, however, team members of all levels communicated freely.

6. Mistake #5: Letting Team Members Specialize

One developer on Team 1 specialized as part of his career; he was a database developer. As a result, SQL work became a bottleneck and subtle SQL errors slipped into production. In the middle of the project, he went on vacation, and it became clear to the rest of the team that database knowledge needed to be shared across the team. After that, other developers made an effort to assign themselves SQL tasks. But the dedicated database developer still ended up with 90% of the database work, including the most complex stored procedures.

Another developer on Team 1 specialized on a particular part of the product. She struggled with the first file upload feature she selected, eventually making it work before the end of the iteration. To become better at this, she assigned herself every file upload task for the rest of the project. Only later, in production, did we discover that the file upload features were extremely poorly designed and error prone outside of the limited test data she had used.

Team 2 showed early signs of this kind of specialization. But thanks to pairing and frequent pair switching, the team recognized that this was a
problem and stopped it in an early iteration retrospective. We intentionally created a culture that allowed developers to assign themselves tasks in areas they don’t yet know well. SQL and AJAX were particular challenges. The few of us who had strong skills in these areas would leave those tasks for other developers to assign to themselves. Then we would pair with them and let them “drive” in the pair as much as possible. Some team members were always better at SQL or AJAX, but by the end of the project, every team member could work successfully on any part of the application.

7. Mistake #6: Using Long Iterations
Team 1 used three to five week iterations, varying the length around holidays, for a total of five iterations in the project. By the fifth iteration, team members had become better at estimating; they understood the Planning Poker process, and they were more willing and able to defend their estimates. As the end of the project approached, they communicated more readily and articularly in all team and customer interactions.

In the gap between the two projects, I coached several other teams. Because the coaching engagements were only two to four weeks long, I had the teams use one week iterations so that they could practice all the parts of an iteration multiple times in my presence. I was pleased with how quickly these teams improved.

Having become comfortable with one week iterations through that coaching, I used them with Team 2. By the end of the fourth week, Team 2 had developed the comfort with the process and the rich communication that took almost six months to develop in Team 1. With weekly retrospectives, they were able to shape the process sooner and had more of a sense of ownership over the way we worked. This created a desire to improve their own work and the work of the team that I didn’t see with Team 1.

Based on these experiences, I’ve concluded that the number of iterations a team has completed is a better predictor of their effectiveness with XP than simply calendar time with XP. One week iterations allow a team to develop this experience in a short time.

8. Mistake #7: Not Including Junior Developers
Given these risks, I still experience a strong temptation to staff an agile project with an all-senior team when possible. I avoided junior teams until I had no choice and was assigned one. Succumbing to that temptation, though, means missing out on the benefits of having junior team members.

At Avanade, most projects are not agile and many developers are recent college graduates. Including junior developers on agile projects allows us to expose them to activities they ordinarily wouldn’t see until several years into their career. Unlike many of their peers, junior developers on agile teams participate in design and architecture, estimate work, and meet with customers.

After seeing unexpected growth in the junior members of Team 2, I conducted an informal study of our summer 2006 college hires to see what effect agile vs. non-agile work had on learning. I asked them to rate their skills in various technologies and non-technical areas like estimating before they joined Avanade, immediately after they completed our six week college-hire class, and as of the time they completed the survey. The responses showed much faster learning among agile team members, especially in the non-technical skills. For the long term success of our organization, it seems, nothing builds our capability faster than staffing junior developers on agile projects.

There are benefits to the projects themselves, as well. Junior developers bring new perspectives, untainted by preconceived notions of “how things are done.” Short iterations with retrospectives can harness those new ideas to improve the process. This lack of experience also tends to mean a willingness to try new things, such as pairing and TDD. I believe that one of the reasons these practices were more readily adopted on Team 2 was that the junior team members have no previous project experience to tell them that pairing and TDD were strange. Their eagerness to adopt the new practices rubbed off on the more senior team members.
9. Conclusion
In many ways, the mistakes I highlight that make XP problematic with junior developers are really mistakes of partially adopting XP. The XP practices work best together. This becomes more visible when the team doesn’t have the experience to compensate for the lack of one practice or another.

In retrospect, I would apply the lessons I’ve learned with junior teams to all my teams in the future. By drawing on their experience, the more senior teams I’ve worked with were able to avoid the pitfalls associated with not fully adopting XP. But had they fully adopted XP, including such practices as pairing, they might have been even more effective.

10. References