Undercover Scrum Master

DANE WEBER, Excella

I hit a wall coaching a team that did not want to try popular Agile engineering techniques such as Test-Driven Development (TDD) and pair programming. I became a Scrum Master after working on the business analysis and account ownership side of software development and could not speak from personal experience about engineering practices. In order to get some first-hand experience and to gain a new perspective, I chose to spend a couple years as a software developer on a Scrum team. I learned more than I expected.

1. INTRODUCTION

Scrum Masters, Product Owners, and others who work closely with software development teams often seem unaware of the experience of the technology-focused people on the team. I became a Scrum Master for software development teams without ever having been a software developer myself. While not necessarily a problem, I wanted to coach my team on technical practices, but had little confidence because I could not draw from personal experience. In addition, while I had other roles prior to discovering the Agile world, I served as the Scrum Master every time I worked with an Agile team. I switched roles from Scrum Master to developer, hoping to gain first-hand experience with Agile engineering and technical practices, but ended up learning more important lessons about the work-life of software developers.

2. BACKGROUND

Coaching a team as a Scrum Master feels like a great fit for me and is something I believe in. While every Scrum Master I've met seems to have a different background, the role felt like it was tailored to me, drawing on my history and interests. When I discovered the role, I loved it and felt at home. While my perspective changed as a software developer, I brought the Scrum Master lens to the experience.

I'm drawn to understanding larger systems and how the elements fit in and influence each other. I studied philosophy in college and clinical psychology in grad school. Both of these, for me, were about understanding and wanting to make things better. I also enjoy creating games and game worlds where I like understanding how the rules balance and play off of each other, shaping the whole system.

I really want to help people live happier lives. I left clinical psychology not because of the work of therapy, which I loved, but because the subject matter of human dysfunction drained me. I landed in the world of bespoke software for the US government and enjoyed learning and assisting however I could. I happily helped with tech support, analysis, testing, account management, and project management. I enjoyed helping coworkers and the users of our software.

Wasteful and especially harmful processes or policies rile me up. Regardless of my role, I got involved in reworking or abolishing bad processes as well as bypassing bureaucracy and finding ways to talk directly with people. When I took Scrum Master training it just made sense and felt like fresh air. The ideas and material emboldened me to propose more dramatic changes. I became one of the champions for Agile approaches on my large project. While it was only one of my responsibilities, I took the role of Scrum Master seriously and later found an opportunity to serve as a Scrum Master full-time.

My job was made easier throughout because of my comfort with computers and the programming I did to automate my tasks. While I used my ability to code in my job for trouble-shooting, batch processing, and task automation, it was never among my explicit responsibilities. I understood and still believe that working on an enterprise code-base with a team of developers involves a lot more than just getting things to work.

My focus as a Scrum Master shifted over time while working with teams on the myUSCIS project. I spent increasing amounts of time on coaching the larger organization and wishing I could coach on technical practices. The Large-Scale Scrum (LeSS) framework has a chart of Scrum Master focus over time (Larman & Vodde) that rings true for me. Coaching the Product Owner and the development team start off as major focus

Author's address: Dane Weber, 2300 Wilson Blvd., Arlington, VA; email: dane.weber@excella.com, Twitter: @daneweber

areas but drop over time as they come to understand and embrace their roles. Coaching the larger organization takes significant effort up front to get permission and buy-in for the team. After clearing that hurdle, coaching the organization is a minimal focus, but grows over time since many of the impediments faced by the team are out of their control and the Scrum Master spends time addressing them. The other area of focus is coaching the team on technical practices, which starts out being the inherited practices, but over time should become one of the Scrum Master's primary focuses.

This matched my experience where, over time, my development team and Product Owner were performing quite well. The team integrated code frequently, running it through an automated test suite, and deploying it to test and production environments via automated pipelines. Product Backlog Items (PBIs) were split fairly small and the team delivered high quality work that provided significant value every iteration. I began spending most of my time addressing larger organizational matters for the sake of my team.

I did, however, see room for improvement in the team's technical practices. For example, reproducing and prioritizing escaped defects was time-consuming and frustrating, especially when a defect was a repeat (a regression). I observed the team writing tests at the end of a PBI's implementation, and also knew of examples where no new automated test was added when fixing a bug. In my mind, adopting the Test-Driven Development (TDD) approach would ensure higher-quality tests with greater coverage, both for regular development as well as when fixing escaped defects. My proposals to experiment with TDD fell flat for a number of reasons, including poor past experiences with it among my team. I had no history with TDD that I could use to motivate my team to try it.

Similarly, while the team was good at dividing a PBI into tasks, these tasks were usually worked by a single person, and any unforeseen absence would put the task and likely the PBI on hold. While "pairing" (referring to "pair programming") was welcome among the team, it usually meant dividing up tasks or consulting with another team member. As a Scrum Master, I imagined that pair programming (with one computer and two developers switching off frequently) would reduce our work in progress, increase resiliency to absences, improve code quality, and possibly even increase the rate of PBI completion. This was an unpopular suggestion met with skepticism and objections from the team's past experience.

I felt frustrated by my inability to champion these technical practices and wanted to learn first-hand if these and other Agile engineering practices could be successful. With the support of my firm, I took a month to ramp up on the Ruby programming language and the Rails framework in order to join a different project as a full-time software developer.

I joined a project modernizing the E-Verify system for the federal government. The team had been quite successful over the prior two years, and this success led to requests to modernize more and bigger parts of the E-Verify system. The project added staff in order to meet these increasing requests and split into two teams. I joined just as the project grew again and was reorganized into three teams. The structure followed the LeSS model with a single Product Owner and product backlog across the teams, but each team had its own Scrum Master and sprint backlog. Each team was cross-functional and expected to be able to address any story added to the sprint. The team divisions had been decided before I joined, but I joined the project before the new division took effect, so I was a founding member of the team.

In addition to a Scrum Master, the team I joined included a business analyst with extensive domain knowledge, a fully-remote DevOps specialist, a fully-remote senior developer, and three other on-site developers, making a team of eight. Project tenure ranged with half of the team having been on the project over a year and the other half having joined in the preceding months during the expansion.

3. UNDERCOVER SCRUM MASTER ON AN AGILE DEVELOPMENT TEAM

My new team spent some time forming and establishing norms, including choosing a name ("Bits, Please!") and team values: Thorough, Joyful, Helpful, Trustworthy, & Respectful. Because this was an ongoing project, however, we started day one with a set of items in our team's sprint backlog and dove directly into the work. While I focus here on things that were problematic in one way or another, the team was exemplary with excellent value-delivery and technical practices. My overall experience was truly wonderful, and I was impressed with everyone I worked with on the project.

3.1 Cognitive Load

It struck me early on and throughout my two years just how much there is to know and master as a developer and team member. This creates a cognitive load that plays a significant role in affecting attitudes and motivations in ways that were not obvious to me as a Scrum Master, but which will be a major consideration when I coach teams in the future. The sheer number of different tools, technologies, systems, and processes that were required to do my job required significant mental energy. This cognitive load was tiring, even though I enjoyed the work. Every tool and technology I interacted with had its own interface, rules, and quirks. While these may make sense and become familiar over time, they require memory and thinking to apply.

Just to get to the point of making my first change to the codebase required layers upon layers of tools and technology. I was issued an unfamiliar Macintosh for the project with a VPN setup that required more effort than it should have. Cloning the code repository required a GitHub account and using Bash and Git. I chose to use VS Code for editing code, which worked well for me but still requires learning its commands and features.

To run the containerized application required using Docker, Make, and Rake, as well as initializing and running containerized Redis, Oracle, and Postgres databases. Reading and changing the application code involved Ruby, Rails, Javascript, React, Redux, CSS, Sass, HTML, Sequel, and even SQL. The automated tests used various testing libraries and frameworks, including Cucumber, JMeter, Rswag, RSpec, Minitest, RuboCop, ESLint, Jest, Codecept, and Cypress.

Once the code change was made and tested locally using a browser, Swagger, Insomnia, or Curl, a pull request had to be created and reviewed, which was coordinated in Slack. After approval and merging, Jenkins jobs would build a new image, run tests, and deploy to various Amazon Web Services (AWS) infrastructure. Monitoring and troubleshooting the test and production environments involved CloudWatch, Splunk, New Relic, and Kibana tools, each with their own take on query syntax.

In addition to the above, various parts of the application required understanding HTTP, OAuth2, NGINX, regular expressions, Flipper, Graphviz, Markdown, AsciiDoc, Sidekiq, Elasticsearch, and more. Troubleshooting might require SSH. The AWS infrastructure included ECS, SQS, S3, and more, each of which might be involved in a change, as well as the government's networking and the Akamai CDN.

If that's not already overwhelming enough, in addition to Slack, communication occurred via Outlook and various different video conferencing solutions. While we preferred to define and visualize work on physical boards, we also tracked it in JIRA. Trello was used for tracking other tasks and improvement efforts. Incidents and paging were done through VictorOps in addition to Service Now. These tools were each used in particular ways based on the project's and teams' working agreements, implicit understanding, culture, and schedule of events.

While each of these tools and technologies solve problems and made our lives better, they took mental energy to use well. None of the above, however, provided any business value themselves. What provided that value were the applications themselves with their business rules and connections to partner systems. Understanding the behavior of these systems and the needs they were built to address was more important than anything else in actually delivering value to the government and their stakeholders.

All of the above is to say that I had very little energy left to spend on many of the things that mattered far more to me as a Scrum Master. While I still cared about making our daily stand-up effective, what was included in the Definition of Done, the format of user stories, and how we were improving flow, I would frequently reach a point where addressing these things felt draining and tiresome.

As a Scrum Master, I had been frustrated when my team would express disinterest or just ask me to make a decision for them, saying that they would be fine with whatever I decided. Other suggestions for new and different ways of doing things would be shot down or fizzle because the team thought it would take too much effort. One particularly confusing piece of feedback I had received as a Scrum Master was that my retrospective event formats were too varied and that the team would appreciate a simple or repeated format. This surprised me because I prided myself on the effort I put into bringing variety and novelty into retrospectives and imagined that anything less would become boring and repetitive.

My experience as a software developer with a heavy cognitive load puts these experiences in a new light. I now see that learning a new retrospective format, trying a new process, and even changing a team norm can feel like having to learn one thing too many. Since the business needs, application, and underlying technology all seem essential to accomplishing the job, anything that seems negotiable is the natural target for freeing up mental resources. When I put effort into various experiments and process improvements, the experience was more taxing and less rewarding than when I did so as a full-time Scrum Master. For example, one experiment I'm proud of was introducing an Apoptotic Review, named after programmed cell death, where we focused on ending experiments and processes. Even though I liked the activity and wanted it to continue, I was eager to hand off responsibility for it. I can only imagine that I would have invested more time and effort into shepherding and iterating on the activity if I were serving as a Scrum Master and did not already feel over-burdened.

Taking cognitive load into account can provide a new perspective and inspire strategies for successful improvement efforts. Regardless of intention, I observed several things that helped me as a member of an Agile team.

Written references that appeared alongside an activity, as opposed to hidden in a document repository or even wiki, made it far easier to follow through on something we agreed to as a team. For example, we created a checklist that automatically appeared in every pull request, reminding us of various criteria for work being "done," including tests, feature toggles, communication with other teams, and other good practices. As our daily stand-ups evolved, we wrote up the format on a poster that hung on the wall where we would gather. Similar posters were created for recurring meetings to remind us of the format or agenda. When we experimented with a Kanban flow, the column policies on the physical board were helpful, while the reference wiki we used after switching to JIRA was largely ignored.

The project culture was strongly egalitarian, and our Agile coach and Scrum Masters usually sought consensus or at least democratic support for changes or even ongoing efforts. While this was important and far better than my past experiences of dictatorial and highly directive management, it also meant that they were often hesitant about reminding the team of our norms or holding us accountable, fearful of nagging or being bossy. I really appreciated the times they did simply remind me of how to do something, such as what our policy is for moving cards to various places on the board. In fact, I would have been happy to have received even more direction and help remaining disciplined. My forgetfulness about a given norm or process was not motivated by opposition to it, but simply not prioritizing it above all the other things I was trying to keep straight in my head.

One thing this experience also confirmed for me was that it can be okay to spearhead an improvement or even to just do something without a lengthy discussion with the whole team. While this takes prudence and willingness to take feedback, I was always happy as a team member to learn that someone had figured out a better way to do something without having to be personally involved in the decision. I might not agree that the new approach was better, but trying it out without a major investment of my time was preferable to having to discuss its merits at length.

The project had a great culture of experimentation. With non-trivial experiments, we found that we needed a "lead scientist" to own the experiment. While the entire project supported experimentation and thus supported each experiment attempted (even if we might be individually skeptical that the experiment was a better way of doing things), many experiments would be tried only in part or not at all. This was simply because many would forget they were supposed to be doing things differently, such as tracking the source of reported bugs or throwing a party when 100 PBIs were completed. The experiment owner would remind people to follow the protocol of the experiment and would make decisions in the moment on how to handle situations that were ambiguous in the original experiment definition, such as how to categorize a given bug or whether abandoned PBIs counted toward the goal. Successful experimentation, meaning that we tried an experiment and learned from it, was strongly tied to the amount of effort put in by the experiment owner.

3.2 Implicitly in a Hurry

Beyond having so much to learn, there was pressure to hurry. I observed this pressure while I was a Scrum Master, and I had addressed it multiple times, emphasizing that the default assumption should be that we do our work well unless there is an explicit decision to take on technical debt now for an urgent need. I had always assumed that it was the Product Owner or people in similar roles who were pressuring the team to cut corners and hurry up when I was not around. I did not realize that I was part of the problem.

A classic experience in software development is Product Owners and other stakeholders asking how long something will take and expressing their eagerness to see it implemented. This implies that sooner is better without explicitly requesting compromises. The project's product management team made a few explicit decisions to rush efforts, but in general were respectful of the time it took to accomplish things well. To my surprise, the pressure to hurry rarely came from them.

The Scrum Masters also supported us in doing good work and certainly did not ask us to rush or cut corners, but the metrics they tracked told another story. Cycle time and sprint velocity imply that faster is better, especially because we would talk about "what happened" to work with a lengthy cycle time or to sprints with lower velocity. Metrics about code quality were given much less attention.

The focus on getting more done sooner without a counter-balance certainly implied that we should sacrifice what we could to get features done, but what really surprised me was that the most explicit pressure to hurry came from my fellow software developers. When implementing a piece of new functionality, I would come across code that should be rewritten to improve the codebase and make future changes easier. This code might even have a comment left behind by a previous developer saying that the code should be cleaned up. I would usually ask a fellow developer for help or advice on how to rewrite the piece of code. Several times I was advised to leave the code alone and consider the backlog item complete, even though the person agreed that the code was bad and should be improved. This advice was given along with justifications about the refactoring taking "too long" or comments about nearing the end of the sprint and needing to get all backlog items "done" by the end.

My interpretation of this pressure to rush is that the team incurred "unauthorized technical debt." The concept of technical debt is that something is implemented more like a prototype now with the understanding that the code will have to be rewritten later, making the overall effort greater. This can be a good business decision in some cases where early revenue, acquiring customers, meeting a compliance deadline, or other payoff is worth the extra effort. There was no such business decision in the cases where I was advised to skip improving a piece of code.

One thing that helped counteract this attitude was the explicit adoption of Thoroughness as a team value and regular reinforcement of our team values at retrospectives where we shared examples where we had fallen short or exemplified our values. When someone on the team identified important improvements to make to the test suites, the naming of classes and methods, or some other aspect of code organization, a team member would on occasion humorously tease them that our team is thorough and look at them skeptically to see if they were going to make the improvements or skip them. Since most of us preferred to improve the codebase anyway, this was usually all the encouragement it took to follow through.

Another practice that addressed some of the technical debt was adopting a "20% rule" where 20% of developers' time was set aside to work on whatever part of the system we wanted. Permission to pay down technical debt was all that we needed to dive in. We were happy to be given the chance and feeling of permission to take time to make improvements that probably should have been done much earlier.

We made a major cutover less than a year into my time on the project. Because of all the stakeholders involved, the product management team made an explicit decision to rush and release something that was not ready. The production issues that followed on this release led to an emphasis on improving the quality assurance practices across the project. Among the things tried during this time was an assessment of the riskiness of each codebase as well as the state of various risk-mitigations, including automated testing, telemetry, and built-in resilience. These scores were visualized, and attention was brought to risky codebases with poor mitigation. Attention to these metrics served as a counter-balance to the cycle time and sprint velocity metrics.

Developers may make unwarranted assumptions that they should hurry and make compromises, and these assumptions are confirmed by Scrum Masters, Product Owners, and others focusing primarily on cycle time and sprint velocity. These metrics and conversations imply that developers should hurry up. When building a prototype or proof-of-concept for a start-up, such compromises may be sound business decisions, but in other cases, such as long-lived federal IT systems, these compromises are almost guaranteed to cost more in the long run. Metrics, conversations, and celebrations of improving the quality and maintainability of the system are important to offset this pressure to hurry.

3.3 Feedback loops and planning activities

Regardless of the amount to learn or any feelings of being rushed, I found writing code to be very rewarding. Planning activities, however, became some of my least favorite meetings, even though I had cared deeply about them as a Scrum Master. I had observed this in the teams I worked with as a Scrum Master, but I did not expect to change my attitude so dramatically.

I could see the direct effect of code I wrote with immediate results. Coaching a team, however, had long and often indirect feedback loops. While some work was certainly frustrating or difficult, focusing on a problem, solving it, and seeing the effects of my work was gratifying and fun. The outcome also provided a positive and rapid feedback loop with relatively little ambiguity. In contrast, my previous work as a Scrum Master involved a slower feedback loop with less obvious results. My efforts as a Scrum Master were directed toward helping my team become more effective, but even where I was successful, the changes usually took place over the course of weeks and took observation and judgment to see. Backlog refinement and sprint planning have visible results that I appreciated as a Scrum Master, but they paled next to the results of writing code.

Few developers on the team, if any, were excited about participating in backlog refinement. As a Scrum Master, I saw backlog refinement as important for everyone, and I imagined that developers would find it rewarding because it gave them a voice in guiding the direction of future work. It definitely frustrated me and my teammates to look at a PBI for the first time in the sprint backlog and to disagree with the direction or implied implementation. In these situations, I would wish that I had been present for the refinement conversations before the sprint, but when the next refinement conversation come up, I was not any more eager to join.

My feelings as a development team member were at odds with how I felt as a Scrum Master, and also at odds with my intellectual understanding of the importance of planning activities. This made the most sense to me when I compared how quickly and frequently my coding efforts paid off with the delayed pay-off of backlog refinement. The experience was a case of immediate vs. delayed gratification. Neither I nor my teammates objected to these activities, but we might not show up to them if we were invited to attend only voluntarily. I volunteered more than the average, probably in part because I came into the team already placing a high value on backlog refinement, but refinement conversations did tend to lean more heavily on a subset of developers.

We had other "chores" that were a shared responsibility which were less fun that writing code. We experimented with a points system inspired by the Harry Potter "house points" approach where various chores were rewarded in the hope that a broader population would volunteer for various activities. While I believe such a system could work, the incentives and rules ended up dis-incentivizing a majority of the team. Points were awarded for a delineated list, including acting as the scribe for a meeting, as well as ad-hoc awards by the project leads. There was a prize for the team with the most points at the end of each month, but if a team had a clear lead, the other two teams figured it was not worth their effort to come from behind, and then even the leading team figured that since they were winning, there was not much reason to volunteer for more.

Finally, the project successfully involved the full population of team members by implementing various rotations for these chore-like activities. We had to swap shifts on occasion, but simply taking turns ensured that we all took part. Because the chore was shared evenly, nobody felt justified in skipping out of less-fun meetings. For some chores, such as being on-call for production support, we held a snake draft to claim shifts for a month, allowing people to pick days that worked better for their schedules.

I intend to take this insight into future facilitation roles: a lack of enthusiasm may simply mean that the given activity feels like a chore to someone, even though they understand and agree with its importance. The lack of enthusiasm does not necessarily indicate disagreement. Sometimes we just have to do our chores.

3.4 Test-Driven Development (TDD)

Test-Driven Development was a key reason that I embarked on this journey. I wanted to understand why my teams resisted TDD and whether it was as amazing as some proponents claim. When the testing infrastructure was in place and well-done, the rapid visual feedback loop was gratifying and motivating. In contrast, when working in areas of the code where the tests were sparse, finicky, or riddled with dependencies, the work was onerous, and I simply wanted to move on to something else. Good testing infrastructure required a group effort, which could be hard to make time for since it is invisible to most stakeholders.

Automated testing was important to everyone, especially because every code commit to the master branch was automatically deployed to the production environment if it passed all automated tests. I tried TDD on my own and with teammates several times. It was often a great experience. In addition to the enjoyable feedback loop of making code changes and seeing the effect, I also had the gratifying reassurance that I had written a failable test that now passed because of my code. This amplified my enjoyment of coding. Even when I was confident that the code worked as intended, a simple test could uncover bad assumptions. For example, I added a one-time event as a "holiday," and I had clearly specified it as occurring in its specific year. The test for the holiday worked, but it was easy enough to add another test to prove that it was a one-time thing. Instead, that test uncovered logic that ignored years and only matched on month and day, which would have made the event into an annual holiday. This seemingly unnecessary and simple test caught a defect that might have survived for years.

This rapid feedback loop was easy when the existing code was already well-tested. The other tests in the neighborhood provided examples to follow and served as encouragement to maintain the test suite. Building out such a good automated test suite took time and dedication. Where tests were better, it was easier to keep them in good shape and contribute further. Well-configured testing frameworks and custom test support also encouraged me to write tests. For example, we used Rswag to test API calls against their schema definitions and, because the schema was comprehensive and avoided repetition, it was natural to update the schema when making changes.

Where tests were poor, however, it was easier to leave them alone and contribute to the problem. Fighting with testing frameworks and mock objects was not a rewarding experience because they are tertiary to the production code changes. Even more demotivating, however, was working on existing code with few or poor-quality tests. In these cases, it was tempting to leave behind only a minimum of new tests. The lack of good tests was often a sign that the code in question was not designed to be easily testable and had strange dependencies. In one example, we used the VCR library for recording and replaying web requests, but instead of using it as intended, we built up a pattern of hand-editing the recorded responses. While this library would ideally make mock web requests stable and quick, we took shortcuts early on that snowballed into a fragile suite that could not be re-recorded automatically, thus making any future changes a frustrating exercise in guessing at how the recordings were created.

Testing first, which is essential to TDD, amplified any difficulties with the existing code and tests. It was easier to figure out after the fact how to test a new class or method than it was to start that way. When I was a Scrum Master my advice was to write tests and improve the testing infrastructure as part of each PBI. While this is sound advice to refactor as you go, a codebase can rapidly deteriorate unless the entire set of developers remain disciplined. A codebase that already has problematic testing, such as some parts of my project's codebase, can really benefit from some dedicated and concerted efforts to improve the tests and testing infrastructure. Without that concerted effort, individual developers can only do so much and may become overwhelmed by the enormity of the cleanup effort.

3.5 Pair Programming

The culture across the teams was fun and very collaborative. Even so, we did not practice the kind of pair programming that I have read about. We frequently swarmed and pair-programmed on PBIs. All of the developers would gather for 10 to 15 minutes for impromptu "Code & Coffee" sessions where the person who called the session shared something they had found or written. When someone needed help, they could type a Slack command that triggered our "andon cord," alerting their team, turning on warning lights in the team room, and getting the team's immediate attention and assistance. When we "swarmed" or "pair programmed," however, each person had their own computer. Switching drivers required the time-cost of committing, pushing, and pulling code. Because of this transaction cost we would often keep one person driving for over an hour. When the pair or another member of the swarm saw a specific change to make or command to add, they would dictate the text or give explicit instructions to the driver. These changes would have been far faster if they had switched drivers for a minute, which is part of the promise of the "two developers, keyboards, and mice with one computer" model.

One thing that made it uncomfortable to switch drivers on the same computer was that we each used the editor of our choice with multiple customizations. Typing and editing on another developer's machine felt alien, slow, and prone to unintended consequences such as when using a habitual shortcut that works differently. We were also not in the habit of plugging in keyboards or mice to our laptops, so setting up two sets of peripherals would have been a strange thing to see. Without the duplicated peripherals, however, there was the friction of being polite about taking over someone's computer. That is, when seeing something that needs to be corrected, it feels rude to push someone aside in order to type on their computer. Only when I felt frustrated by not seeing what I was being asked to do did I think to slide my laptop over and offer it to my teammate.

I still hope to find opportunities to pair with duplicate peripherals, the "real" way. I have not lost faith in pair programming, although I do appreciate other forms of close collaboration. What does strike me, though, is that pair programming is not a natural result of close collaboration. It requires setting up the right environment (the desk, computer, peripherals, chairs, editor, bindings, etc.) and then agreeing to and practicing habits of passing off control of the computer. This takes commitment and practice, since it is not a natural evolution.

4. WHAT I LEARNED

I did not have the experience I wanted with the two key practices that motivated me to become a software developer: TDD and pair programming. While I still see their potential value, I have a much better appreciation for why taking them on is a challenge for so many developers and teams. I still hope to do each of them more in order to fully experience them. When coaching a team to try TDD, pair programming, or something else that requires discipline and is not a natural evolution, I intend to advocate for a commitment to stick with the practice for an extended period of time before rejecting it. As an illustration: riding a bike is not rewarding at first, wobbling and tipping over, but after sufficient practice it is a fun and efficient way to cover ground. Rejecting bike-riding while it is still slow and painful is not a fair assessment. On the other hand, few would learn to ride a bike if they could not see others enjoying it. Simply having an intention to try the practice is unlikely to result in much.

The stereotypical complaint about "too many meetings" resonates with me far more than it did, given that many meetings are part of slower and less-rewarding feedback loops. Worthwhile activities are worthwhile, however, even if they are not enjoyable. A lack of enthusiasm for an activity is not implicitly an objection to it. The "tragedy of the commons," where shared responsibilities for long-term success are neglected, seems to apply to software development teams. My limited experience indicates that the effect is greater with larger teams, which also just makes sense to me because as more people share responsibility, each individual has a smaller share. I expect to take extra care to ensure that teams understand the value of the various activities they support, as well as improving the quality and rapidity of the feedback loops involved. Assigning ownership, rotating or otherwise, is now a tool I have gained for ensuring that shared responsibilities are addressed well.

Even unintentionally, Scrum Masters, Product Owners, and others may imply that a software development team should rush and cut corners. I believe that it is important to counterbalance that inference in order to maintain and improve the codebase in order to avoid catastrophe. On the one hand, software developers should be professionals, acting in the best interests of their clients and stakeholders: we should just do the right thing. On the other hand, if you nag your surgeon enough to hurry up, he might be tempted to skip the hand-washing and use half as many stitches. Just as surgeries-per-day should not be the primary metric of success for hospitals, making features-per-iteration your primary, much less only, metric can incentivize unmaintainable code. Explicit support for improving and maintaining the codebase is important from all concerned. Looking at metrics or other assessments of code quality can be an important counterbalance to velocity/cycle-time metrics. While end users may not be impressed by re-writes and refactoring, the team and relevant stakeholders should celebrate these improvements.

Team members who work with numerous tools and technologies may have a heavy cognitive load which discourages them from adopting new processes and practices. Appreciating this will change how I view improvement efforts and lack of follow-through. Easing the burden improves the chances that a new approach will stick or that an experiment will be carried out. It also helps in other areas where people are expected to remember the agreed-upon way to do something. Checklists, templates, and other write-ups are helpful, but mostly when they appear in the space in which you already are, rather than in a library that must be checked. Given our human nature, timely hints and reminders are invaluable. In addition to providing these aids, I also plan to sometimes simply take action and encourage others to do the same: the lack of appetite to discuss improvements or experiments does not mean that people are opposed to the improvements or experiments themselves.

5. ACKNOWLEDGEMENTS

Every member of the "Bits, Please!" team has been a joy to work with, helping me grow as a software developer and teammate, and putting up with me patiently: Matt Acors, Mark Brown, Jay Danielian, Kaleb Dufera, Hugh Gardiner, Cameron Ivey, Mark Koenig, Angela Lu, Griffin Norris, Iksu Oh, Jen Pengelly, Chris Petro, and Siva Sivakumar. This experience report would have made far less sense without terrific feedback from Candase Hokanson, Mindy Bohannon, and Mark Koenig. Mark Grove, Mark Heppler, and Mark Koenig each encouraged me to share what I had learned from my time as a developer. Julie Wyman suggested the "Undercover Scrum Master" title for the first time I spoke about my experience. There would be no experience report if not for support for my mission from my employer, Excella. Tony Solomita and Fadi Stephan were both instrumental in arranging for my transition and I am indebted to them. No one believed in me more or sacrificed more for this than my wife, Jeanne Weber, whose support was essential. Thank you!

REFERENCES

Larman, C., & Vodde, B., "Scrum Master focus" within the Large-Scale Scrum (LeSS) framework,
https://less.works/less/structure/scrummaster.html#ScrumMasterfocus