Agile Discovery - Aiden Burgess

SOFTENG 306 Refactoring Project

For the second half of SOFTENG 306, I lead a team of seven in a refactoring project. This was my first experience leading a software project of such a large team. In my personal work I prefer having a backlog of items, which can be pulled from as tasks I am currently working on are completed. This follows a Kanban style of work, and so I suggested for our team to use Trello to track the backlog and tasks. Tasks were not assigned (assuming self-assignment) and there weren't clear deadlines for tasks.

After a few weeks, we found that the team completed less work than expected, so we were behind in terms of progress, and there was an unequal workload amongst team members. Some people have a tendency to procrastinate, so this style of Agile did not work well for them. Also there was not much accountability as we held meetings infrequently, so people didn't realise that others were getting work done. As the project deadlines approached I recognised these issues and changed added clear deadlines and assigned people with specific tasks. After these changes, the team worked extremely hard over several days to complete all the tasks.

Daily standups, which is a practice of both Scrum and Kanban, were not feasible in this project and generally in a university environment. Students have different schedules, and are not working full time on the project assigned to the team. This makes it impossible for all team members to meet daily with substantial updates. Instead, this practice can be adapted to be held a few times a week, to allow for semi-regular updates. These semi-regular standups would be useful in motivating the team and informing everyone of the project progression.

Through this experience, I learned the Scrum value of adaptability and the Agile value of responding to change. If we had followed the sprint retrospective the problems that were found would have been discovered sooner, and there wouldn't have been an unsustainable workload near the end of the project. I think sprint retrospectives are more important earlier in a project to set good practices and reflect early on productivity. I also should have fostered an environment to support my teammates and helped motivate them instead of taking such a hands-off approach. Instead of interacting directly with my teammates, I trusted the processes and tools too much. The development at the end of the project was not sustainable, so did not follow Agile principles.

I had to make a difficult decision during this project. There was a big PR that we discovered was going to be too complex to complete in time for the project deadline, so I responded to this change by shifting our focus onto smaller and easier refactors, rather than trying to stick to our original plan. In our team's retrospective we all agreed that this was the correct move, and it reinforces the Agile value of preferring responsiveness over a plan. An improvement to this situation would have been following the XP principle of incremental changes; and to have split this large PR smaller, more manageable PRs. This way, some of the progress made during the PR would have been integrated, instead of the entire PR being discarded.

Amazon Internship

Over the last summer break, I worked at Amazon in an Agile team. Initially, my team was following a strict version of Scrum. There were daily standups combined with operations discussion which amounted to over one hour each day. Our team identified major issues in a retrospective. Estimates were not accurate and we couldn't perceive a way to make them more accurate. The amount of time spent in meetings was too excessive. Therefore, we decided a change was needed and considered Scrum, Kanban, and a mix of Kanban and Scum. Finally we decided to use a mix of Kanban and Scrum, reducing the amount of time in meetings, and keeping the overall goals clear through a Kanban board. There would still be sprints, but there would be continuous flow of tasks from the backlog to the work in progress.

I also experienced many XP practices, such as pair programming/debugging with my mentor, testing and continuous integration, collective ownership, and an "on-site customer". Almost every line of code I wrote had to be tested, with many types of tests. The team also actively practiced collective ownership. As all the code had to be reviewed before being merged, no one person was responsible for any issues, the entire team took responsibility. We also received most of our tasks and issues directly from customers, they would send tickets with issues or potential features, and we would respond to each person individually and in a timely fashion. These customer tickets made up most of our product backlog.

There was an internal anonymous survey to measure how the team was feeling about our work and projects. My manager noticed that satisfaction had been decreasing slightly over a month or so, and scheduled a meeting for the team to air any concerns. Here, everyone spoke openly about issues such as excessive amounts of time in meetings, stress, and operational issues.

In a University context, it is important to adjust the teams behaviour as the project progresses. To achieve the team needs to actively value openness and courage to bring light to any issues and solve them together. It is unlikely to have the same amount of collaboration with customers as in my internship, as most university projects do not have a dedicated person to act as the role of the customer constantly involved to provide new scope. As I learned at Amazon, this can also be seen as a benefit, as customers have infinite wants, and the scope of projects for university should have limitations to be fairly assessed.

Collective code ownership (XP) in industry allows each team member to make changes to any part of a code base. This may not work as well in a university context, as there are strong time constraints. Under these constraints the time needed to switch to a different module - reading documentation and understanding code - may be inefficient. I prefer the system of weak code ownership where modules are assigned to owners, but team members can modify other modules if they feel the need to do so. This means that there is always at least one person who has deep knowledge of some part of the codebase. I have experienced this system in a university team project where some team members remained on the same area of code, while others switched between the frontend and backend. I found it helpful to understand the code base while switching by talking with the existing members on the codebase.