# 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, so I suggested that our team use Trello to track the backlog and tasks. Tasks were not assigned (assuming self-assignment), and there weren't clear deadlines for assignments.

After a few weeks, we found that the team completed less work than expected, so we were behind in progress, and there was an unequal workload amongst team members. Some people tend 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, added clear deadlines, and assigned specific tasks. After these changes, the team worked extremely hard over several days to complete all the tasks.

Daily standups, 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 help motivate the team and inform 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 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 critical early in a project to set good practices and reflect 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. At the end of the project, the development was not sustainable, so it did not follow Agile principles.

I had to make a difficult decision during this project. We discovered a big PR would 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 sticking to our original plan. In our team's retrospective, we all agreed that this was the correct move, reinforcing 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 splitting this significant PR into smaller, more manageable PRs. This way, the progress of the PR would have been integrated into the codebase instead of discarding the entire PR.

## 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 significant issues in a retrospective. Estimates were not accurate, and we couldn't perceive a way to make them more real. 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 time in meetings, and keeping the overall goals clear through a Kanban board. There would still be sprints, but there would be a 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 practised 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 problems 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 anonymous internal 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 and scheduled a meeting for the team to air any concerns. Here, everyone spoke openly about excessive amounts of time in meetings, stress, and operational issues.

In a University context, it is essential to adjust the team's 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 a benefit, as customers have infinite wants. The scope of projects for a university should have limitations to be reasonably assessed.

I have not practised pair programming or team programming in a university project. Some of the upsides could be knowledge sharing, where the strengths of both team members can be shared. A simple example could be one person knowing a key binding and sharing that so that the other member becomes more productive. A downside is the time commitment and scheduling issues. Both people need to be present and actively participating, so one member's time that could have been used to work on the codebase is spent watching and talking. However, this drawback can be offset depending on how much value they gain from the experience. Therefore, I think the most effective matching for pair programming would be individuals who have very different strengths and weaknesses, as they gain the most from working together.

Collective code ownership (XP) in the industry allows each team member to make changes to any part of a codebase. This may not work as well in a university context, as there are substantial 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 code area while others switched between the front and backend. I found it helpful to understand the codebase while switching between areas by talking with the existing members on the codebase.