Reflections on TDD, Pair Programming, and Refactoring

I felt that the communication aspect between myself and my partner was generally positive during the first session. We first chatted about what we felt we could commit to in the time given. I found having to agree with someone how complex story was before starting forced me to think hard about why ranking I gave it. I found I had to defend why I thought one story would be harder/easier than another and on some points had my mind changed.

During the development stage of the session I first assumed the position of the navigator. I think this was helpful to my partner who wasn't as strong at coding as myself. They were mostly happy to be guided by what I suggested but would question when I said something they didn't understand or disagreed. We'd then have a conversation about it to decided on the best course of action. This was extremely useful as there where occasions in which the navigator would be caught out and issues were often caught early. Having two pairs of eyes during testing and development was also useful to catch issues earlier. It also help when trying to think of all the test cases for a particular scenario. There was not a single time that both of us thought of every possible test case.

I liked the "red-green-refactor" approach taken during the sessions, but I found that I felt it difficult to adhere to. One reason was probably due to habit. Another was because of the triviality of the code we were writing. It was fairly obvious what the design should be without strictly adhering to the method. I'm keen to try the approach in a more practical setting, potentially a forthcoming assignment. I think sticking to the method requires a bit of practice to become habit and is more obvious useful when developing something in the real world.

One of the more challenging things for me during the refactoring exercise was to split the code from one big loop into several smaller loops. It seems very counter intuitive and inefficient to use three loops when you could just use one. I understand this is because actually this is probably not a case where you're going to see bottlenecks, but after you've been trained to think about how to intelligently structure you're code for so long it becomes difficult not to prematurely optimise. I think that refactoring code so that it becomes less efficient but more readable is a skill that needs to be acquired by practice.

I think that TDD, pair programming, and refactoring can be used to deliver a good design but that these are not the only requirements to success. I believe that while they are certainly enabling practices that can help a developer achieve a good design, the ultimate deciding factor for success is the commitment of the developer to following the practices. These closely practices is what separates the agile agile process from ad-hoc hacking. However if a developer can stick to these principles I think they will provide a powerful yet lightweight and flexible way of working.

My pair found that writing tests first produced a greater satisfaction that writing them after producing the production code. Writing tests first forced us to think of all of the different test scenarios up front, which meant we thought more about what the production code was going to have to do. It also had a the bonus of not having to write the test as an afterthought. As soon as we correctly wrote the production code we get all green checkboxes and we're immediately onto the next story. I felt that doing the dull bit up front (testing) makes it easier to mark a story as "done-done".

Having a decent set of test coverage before attempting to change something, either to add new code or to refactor, was very useful. I found that this gave my pair a confidence boost that we wouldn't break something when making changes. We even managed to catch ourselves from breaking the code even during the basic exercise in the workshop.

An additional point that occurred to me during the workshops is how do we use these practices to handle difficult test cases? One recent example that comes to mind is in a WebGL project. There are some cases which are obviously testable (does this model have n vertices?) and some that are not (is this being Sphere drawn correctly?). I seems that the red-green-refactor work-flow breaks down when you no long have decent test coverage for the red and green steps.