Project Reflection

It's difficult, when beginning something as open ended as a self-motivated group project, to decide on a software development methodology; particularly when you are learning about them simultaneous to those decisions being made. You are, in effect, making decisions that will have deep ramifications months from that decision, and it can be very difficult to uproot and regrow your project if you decide that things are not working as intended. In looking back on the decisions we made, we feel that this project went well – that we followed a model that allowed us to course-correct when things went in the wrong direction, but mostly produced good code and a robust project which we could be proud of.

While the methodology we chose doesn't quite fit perfectly into the labels that we learnt about, what we did was much closer to agile software development than it was to waterfall, and the benefits of that became clear very quickly. With one or two meetings a week, we could stop ourselves from heading down a misguided path, while feedback from the other group members lead us to design better classes and subsystems. This also kept us focused on the project – even though we all had four other classes, we consistently had new code every week, and we feel that these frequent meetings created an incentive to remain up-to-date in terms of the workload each member was responsible for.

Perhaps the most important aspect to ensuring that we created correct code in the first attempt was designing unit tests *before* writing our code. This was emphasized in class, and, though not always the obvious thing to do when you are designing a subsystem, it's clear to us in retrospect that unit testing was integral to keeping our project's timeline moving along smoothly. We would generally write unit tests before adding a feature, retaining them as regression tests, and making sure that we then had integration tests ready when additional functionality was added. We also began to make use of Travis about halfway through the project, which was a great way to track any problems our commits might have made. We also paired it with a slackbot, so we could see who made the commit, and whether it had passed or failed. While not every skill we acquired in this project will necessarily be taken to our future careers, the ability to write appropriate tests will certainly be of use to us in the future.

The above points are, of course, important parts of good software engineering in a group setting, but the most difficult and important aspect is the splitting of tasks and the management of group dynamics. This is perhaps the only aspect of the project in which we have some mixed feelings. Unfortunately, there was a tendency for members to fall into the roles that fit their previous work experience, their strengths as programmers, and their own personal interests. This is particularly difficult to avoid when you have 12 weeks to produce a working product. Nevertheless, there was a tendency for people to fall into particular roles, to be very knowledgeable about that role, but be somewhat blind to the other parts of the project. Some of the ways we might combat this in the future is to take even better advantage of GitHub's wiki features, to provide presentations to other members about subsystems that we have completed, and to have role switching at set intervals.

Software engineering is an extremely difficult discipline to teach to students, because it relies more on principles than a set of skills. Of course, there are many concepts that will inevitably of use to software engineers – object-oriented design, concurrency, user experience, development methodologies etc. But there are certain intangibles which can only be learnt by being thrust into a

situation which simulates the real thing (or just the real thing). We feel that, in constructing Silver Screen, we got a real, substantial taste of what actual software engineering looks like; what it feels like, and we know that we will use, re-use and hone these skills in the future.