Naiem Gafar
CSCI 370: Software Engineering
Dr. John Svadlenka
13 December 2020

Project Post-Mortem

The final deployment of Sprout Exchange is an accumulation of brainstorming, planning, and development that was extended across almost four months. Throughout this entire process, as a software engineer, there were several decisions made early on that proved to be impactful on the overall outcome of the project. Likewise, whereas the plan-based waterfall approach employed for this project ensured its constant progression, concerning several aspects, it proved to be a hindrance. Yet, there were several features of the architecture, namely its proximity to the Django framework upon which the system was constructed, that simplified a lot of the design planning and accelerated later development.

As with any long-term software engineering project, several takeaways may be used for future improvement. First and foremost, it was realized from a very early point in the process that the scope of the project would have to be considered given time and labor constraints. Since it had to be completed by the end of the semester, and given that this was an individual project, it was clear that priorities had to be set. Whereas my requirements document lists almost 20 varying requirements, in reality, these could be divided into three general categories in order of their urgency. Requirements that involved the security of users, such as that concerning user authentication, have always been a top priority. Likewise, the main functionality that would enable gardens to be created, listed, and invested by other patrons would be a part of the base layer. Other functionality, such as posts, commenting, and secondary sales were nice to have, but not a must. During the planning and implementation phases, I, unfortunately, failed to focus my priorities on only a few requirements, instead attempting to cover everything. With this, the scope of the project inevitably grew to a point where features were being planned that did not contribute to the overall system, while other significant functionality had still not yet been finished. Perhaps I should not have overcommitted from the beginning.

Nonetheless, this project has expanded my knowledge about the model-view-controller pattern. As it pertains to the overall architecture of the system, I believe that the framework that I employed worked well. For a system such as this that requires user authentication and security, the added benefit of using Django is that it comes with built-in login, logout, accounts, and security features that I would have otherwise had to implement myself. The reuse of existing functionality through third-party libraries also allowed me to focus less of my attention on things that were not core functionality. For instance, when it comes to address validation, I had to ensure that users were entering valid addresses. Instead of doing that myself, or even relying on some external service to call upon, I instead used a python library to validate

the address as users were typing it in. Likewise, when it comes to making the web pages mobile-friendly, using the bootstrap API does this automatically. Thus, although I do not have a mobile application, I am confident that my website will work well for users on mobile devices.

Having a well thought out database design before implementation also proved to be quite valuable because once the data was decided upon, the methods that needed to be created were just altering these members. The design choice to incorporate CRUD operations across the entire system, although it may seem like an elaborate feature now, will prove to be valuable in the future when I am ready to turn this system into a mobile application. Using the URL structure and these CRUD methods, I can easily communicate with the database, grant certain users certain access to certain information, and eventually be able to develop a REST API.

However, there are several aspects of the project that did not go as planned. For instance, whereas most of my requirements were clear and meaningful, those concerning the handling of payments were rather ambiguous. Thus, I had a very difficult time implementing or fitting payments into the overall architecture and design because it had not been clearly defined from the beginning. Perhaps this means that the currently deployed version will be more for user testing, and not actually in a deployed environment where real customers will use it. Moreover, in focusing on the overall design, architecture, control of data, and incorporating the business logic, I severely overlooked the user interface design. Thus, many of the pages are not as pretty as they should be and there may be a lot of rough edges when it comes to visual aesthetics. Likewise, the "planning" aspect of the website also had quite ambiguous requirements, and thus, there is not really as elaborate a planning page as I would have liked. Going back, I feel that I should revisit all of these "short-fallen" features, generate a set of new requirements, and go back through the SDLC activities for them.

Overall, when it comes to the development of a website, especially for a transaction-based system, the waterfall approach that was utilized does not work as well as is expected. For instance, committing to a certain set of requirements at an early stage means that later on when I came up with changes, it was harder for the process to adapt to them. In my opinion, an agile methodology would have worked better in this scenario. In that manner, I would have worked on the business functionality first, without regard for the secondary features. Likewise, when developing a website, the documentation sometimes got in the way. When you have a methodology that requires detailed documentation, it is not as easy to just pivot and do something else. When it comes to websites, there is a lot more flexibility for these quick changes, and so a methodology that embraces change would have been beneficial.