Moti Begna
CSCI 5801
5/2/2019

Final Assignment

Part 1: Post Mortem

At the beginning of the semester, there was no doubt in my mind that the class would require a fair amount of work to be put in in order to meet the requirements of the course. The extent of that work, however, was not something that I could have predicted. When it came to the various stages of development, the man-hours spent discussing and formulating each document such as the initial User Requirements and SRS took anywhere between 5-10 hours of individual input for every step of the document as a whole. In many cases, it was not the work that was actually put out that took the most amount of time, but rather the time spent thinking about what could be added and/or mentioned. Specifically, I recall spending a lot of time conceptualizing what my UML diagram would look like and how various classes would end up relating to one another. In general, I was unaware as to how much of a role documentation would have in the class.

The specific area of development that stressed the most amount of attention to detail had to have been documenting the specific requirements for each user after being given a vague outline of the tasks that the users could accomplish in the system. Ensuring that individual requirements were not only clear and concise, but also kept consistent in comparison to other requirements was a task that couldn’t simply be met by a single person. While I believe that the work that I put in was adequate, it didn’t take long for me to understand that having a team that you can rely on to give new perspectives and insightful ideas is an integral part in the development of a system. It was when we worked on the assignments together that we were able to locate contradictory statements, unnecessarily detailed segments, and underdeveloped ideas. For example, when working on how a student would interact with the TA application, we spent a majority of the time bogged down on the details of the GUI such as whether or not we would employ lists, selections, or text modules for various questions. It wasn’t until one of our teammates pointed out that we where dealing too much with the UI layer that we reevaluated our process.

When it came to the requirements document as a whole however, it didn’t take long for some elements of the document to become invalid. In fact, once we started thinking about the design of the system, we quickly realized that we were missing important elements of the system, and that some elements that were combined needed to be placed separately to meet OO principles. While nothing major went wrong when building the requirements document, when it came to missing elements, we needed to introduce new classes that handled different routines. One instance of this involved the creation of an Announcement class that would be instantiated by the Admin class if a request existed for it. With our initial class diagram, all subroutines that related to the administrator were kept in one class. It wasn’t until we started the design process (assignments 5-7) that we realized this would be a problem if we wanted to ensure that classes exhibited high-cohesion. Thus, we also separated out how faculty recommendations would be submitted by creating a Recommendation class.

Another element of the requirements document that we realized was missing was how we would keep track of the students that decided to not accept a TA position after being sent an offer. While a test case for that instance was mentioned, no steps beyond a simple response was regarded within the rest of the document. This proved to be an issue once we started on the design process, since gathering and retrieving information from the central database became a key aspect of the system as a whole. In our design, we decided upon the idea that once a TA rejects their offer, they will simply be marked with their respective user identity, and that administrators would be able to see this response and act accordingly in hiring a new individual for the position. In essence, it wasn't until we truly thought about the design of the system as a whole that we discovered that we unknowingly ignored important interactions. And while our solution was simple, we were able to address it by the 6th assignment.

Once we got to the design stage, issues such as the aforementioned continued. We had to no longer thing about the system as a set of abstract user tasks and cases, but rather using specific method calls, routines, and subroutines. The root-cause of our issues during the design process was a lack of complete understanding as to how the system would be interacted with. Particularly, we had issues pinpointing the interactions between the BLM, UI, and Database layers. When it came to the UI/BLM, we had difficulties deciding what class should first be called once a request has been sent to the system from the UI. We spent a lot of time thinking about how requests would actually be sent, how our system would be able to handle and translate them, and even if we should be thinking about the specifics of any of that at all. We came to the conclusion that a parent User class would be the first handler of requests, passing requests to its children classes that represent the various users of the system. This solution was not perfect by any means, and in fact did not address much of the issues we had with how requests are handled. The BLM/Database layer interactions didn't cause many issues, but it did force us to reevaluate how information would be gathered and retrieved. Instead of all of the data being sent to the central data unit, we introduced an intermediary data store in the BLM layer that specifically pertained to holding TA information, such as applications and faculty recommendations.

Looking back at our work during the design process, there are a couple of changes I believe would have helped in properly implementing the system. Firstly, as mentioned before, we did not have a clear understanding of how requests would be handled. While we had a firm grasp on how the various classes within the BLM would interact, we should have specified how requests would be transposed into the layer. This clarity would be essential, seeing has how the entirety of the system hinges on the correctness of request handling. Secondly, I believe that some of the methods and classes that we introduced were underdeveloped when it came to their descriptions. For example, when it comes to creating an Announcement object and dispatching it to other users of the system, the specificity of how it would be sent we not mentioned, but rather that it "would" be sent after its instantiation. Class interactions are integral in ensuring that not only would those implementing the system be able to understand how various classes would communicate with each other, but also to be able to allow them to think about how they could integrate certain design principles—such as loose coupling—between classes. Finally, I believe that we did not take alternative cases, such as rejection of an offer, into consideration as much as we should have. If we did, we could have added more to the system, such as a class that would handle how rejections are recorded, and properly deal with them so that responsibility would not have to solely be on the administrator. As we mentioned earlier in the class, users only want to reach their goal when using the system. If there's anyway that we can mitigate the process towards that goal, we should spend time finding a way to do so.

Part 2: Design

A key part of the system that I believe would have been drastically more efficient if a design pattern was used is how students are viewed by the system and other users, such as administrators and payroll staff. In particular, I believe that the decorator pattern would be a good design pattern to include in this area. By utilizing the decorator pattern, the system would be able to identify the role of a student using the system. Use of this pattern would allow the system to add new functionality to the students based on their role without altering the original structure of how students are viewed in the system. This would allow us to maintain the principle of flexibility, utilizing the "closed to change, open to extension" ideology of design. Specific roles that a student could be decorated with include a previous TA role, an Incoming TA role, and/or a Current TA role. This would lessen our use of flags that indicated whether or not a student is or is not any of the previously mentioned roles, thus requiring less conditionals to be used in identifying the various aspects of the system that the student could access.

As mentioned, the alternative would be to use flags as we have done in our current system. By doing this, it would require that other classes have some degree of access to the student class (if the attributes were not made private, and getters/setters were not included). More conditions also mean more tests that would have to be run later down the line, and if we were to focus on condition coverage testing this would prove to be an incredibly arduous task. The decorator pattern would eliminate many of these issues by maintaining the state of the student object, and simply wrapping a role around it that would have its own access to routines in the system.

When thinking about the implementation of this design, there are multiple instances in the system that a student might be decorated. For example, once a student logs into the system, the system may check if they have been in a previous TA position before, thus giving them a new role. This role might allow the student to bypass certain aspects of the application, where the system would, instead of requiring them to give certain answers, prefill answers using information from their previous application. Another instance of decoration could occur when a student accepts an offer. Once the system adds a new role to the student, the student would be given access to view the appointment details, and maybe even be given access to view the other incoming TA's in their specific class that they would be working with. Overall, this design pattern would not only ease system routines, but also give users a quicker and better route to their goals.