David Mednikov

CS361 Summer 2018

Reflective Journal B

Wednesday, July 18, 2018

Today our team met to go over the requirements for homework assignment 3. In this assignment, our team was tasked with creating and analyzing the architecture of our product. As a team, we had to create a dataflow diagram showing a high-level architecture and a second dataflow diagram showing an alternate high-level architecture, identify two failure modes and draw a fault tree for each one, determine which architecture is more prone to failure, and based on that assessment, decompose one of the two architectures that we created a dataflow diagram for. In our decomposition, we identified two important elements of the selected architecture, and gave a lower-level dataflow diagram for each one. Then we validated the architecture that we selected for decomposition, walking through the use cases and describing how our selected architecture supports each case. Last, we explained the implications, specifically how we would revise our architecture based on the results of our validation, and how that would impact the quality attributes and potential for failure.

When going over various architectures as a group, we determined that the repository and blackboard architectures made the most sense to use. We liked the idea of the publisher/subscriber architecture as well, but we needed to incorporate a repository/database in some way, which is why we chose the blackboard architecture, as it provides a nice hybrid between the repository and publisher/subscriber architectures. After drawing our two dataflow diagrams, we selected Reliability, Efficiency, Usability, and Portability as our key quality attributes. As our program will be an app, we felt that an app these attributes were the most important. We want it to work seamlessly on multiple platforms, especially iOS and Android. We want it to be user-friendly so that first-time users do not have any issues understanding or navigating the app. And of course, we want to make

sure that the app is fast and always online. Since we will need to load map data, ping payment APIs, etc., efficiency and reliability are very important aspects of our design.

We determined that the repository architecture will be more reliable, as there are less connections that must be active for the app to work. However, less connections also means less reliability. So the blackboard architecture will be more efficient. We found that neither architecture made much of an impact on usability and portability.

In going over failure modes, we felt that that the likeliest failures would be related to our API connections. The APIs might be down, or we might send over data in a format that they do not expect, or some other issue may arise. Either way, the API connections create the most opportunity for failure. Our app will connect to a maps API (likely Google Maps) to retrieve map data and a payment processing API to process donations to charities, so these were the two failure modes that we decided on.

After examining the failure modes and how our two selected architectures would behave with them, we determined that the blackboard architecture would be more prone to failure. This is because there are more moving parts per transaction. Only one data transfer needs to fail for the entire transaction to fail. The failure could be between our server and the API, or the publisher and our server, or the server and our subscriber. Essentially, because there is more opportunity for error, the blackboard architecture is more prone to failure. This helped us determine that the repository architecture would be the better architecture for our app, as it will be more reliable and less prone to failure, despite it being less efficient than the blackboard architecture. We decided to go over this in the coming days.

This meeting showed me just how much work goes into designing a system architecture. There are many details involved with selecting the right architecture, and it really is essential to draw the architecture out in front of you and analyze how it will affect the key attributes that we found most important. Even then, that simply provides us with a higher-level overview of how our

system will be organized. We still need to figure out the nitty-gritty. Like most assignments in this class, this made me appreciate the amount of meticulous planning that goes into software design and development. It's not just about choosing a good architecture – it's about choosing the right one.

Friday, July 20, 2018

When decomposing the lower-level diagrams for our repository architecture, we decided that two important elements that we wanted to diagram were the client-side and server-side apps. This would allow us to get a detailed picture of precisely what components are necessary for the two sides of our app and have a better understanding of how they are all related.

After drawing our lower-level diagrams we went on to validating the repository architecture in terms of how it would work with the 3 use cases we defined earlier. We found that the repository architecture would work quite well for all 3 use cases, as they all involve sending data to a database and receiving data back. Thus, we did not determine that revisions would be too necessary, except for edge cases such as two users submitting a cleaned area at the same time, especially if they are working together. The solution we came up with for this would be to check the users' GPS coordinates, and if they were the same, our system will know that the two users worked together. It has not yet been determined what we will do in cases like that. We might give each user only half the typical rewards/badges for cleaning, or we might just give them the normal amount. We do not want to disincentivize users from cleaning, and people are likelier to spend a lot of time doing something with friends. So, it makes sense to give them the same amount of points as they normally would have earned had they worked on their own, but associate them both with the same task.

Another revision we proposed would be to have backup APIs for map data and payment processing. If the Google Maps API or payment processing API were to go down, our app would

essentially become unusable. So, we determined that it made sense to have backup map and payment processing APIs that our server could communicate with in the event that one of our primary APIs went down. We would only use the backup when the primary APIs were unavailable, but this is still something that ought to be developed before going live with the app.

Last, since all of our data processing will be done by one main server, we felt that it would help our efficiency if we were to have multiple servers to meet the traffic demands of loading map data, sending photos, processing payments, etc. For this reason, we liked the idea of using Amazon Web Services (AWS) to bring on additional virtual servers as a backup to our primary server. We would only use AWS if our primary server was unable to handle the load currently being placed on it, so we would still be able to enjoy the reliability of a single-repository architecture while also having a safety net for when our server cannot perform efficiently.

This part of the assignment helped me recognize the amount of details that go into designing the lower-level aspects of an architecture. Also, it showed me that even when a stage of the design process is "complete", it is not actually complete. After decomposing the repository architecture, we determined several revisions and improvements that could be made to increase reliability and efficiency. These changes to our architecture are quite fundamental in design, so implementing these would require us going back and making many changes to our previous work. But, that is part of the process of software design and development. New problems and ideas will come up, and they will come up later than we'd want them too. That's part of the fun though!

Sunday, July 22, 2018

This week my group designed the system architecture for our app. In doing this, we learned a lot about the value of thoroughly planning and designing a system architecture early on in the stages of software development. There are many moving pieces in a program and it is much easier to build something from the ground up when you have a solid, fully detailed plan. Similar to how

buildings aren't designed as they are built, software isn't designed before it is built. Having a blueprint is essential to build a system that will be effective, efficient and usable.

There are many advantages to having documents for the requirements and design of a software system because, as mentioned earlier, there are so many moving parts. Building a big project without a solid plan is undoubtedly going to lead to issues. Something will be implemented wrong, something else might break existing functionality, the system might not be designed for future development, deadlines will be missed, etc. Having everything written down in clear language will also make sure that there isn't any confusion due to interpretation. Essentially, it makes the development process smoother in every way. There is no shortage of advantages.

The larger the project, the more reason there is to document everything in advance. If there are many software engineers that will be working on it, we need to ensure that they do not have their own interpretation of the tasks. If the system is complex with many components and a lot of communication between them, we want to make sure that the components are all developed so that they will work nicely alongside one another without cluttering up the project. Essentially, anything that makes a project more difficult, whether it be due to complexity, size, functionality, etc., will benefit from having clearly documented requirements and design.

The main circumstance where it would be acceptable to not document everything is when time is a constraining factor. If there is not a lot of time for a project, then it is difficult to devote so much overhead to writing detailed documentation and design. If there is a feature or system that needs to be built in a few weeks for example, we may not be able to afford the luxury of painstakingly going over all of the details before development begins. In cases like that, it is better to have a clear idea of what needs to be built, and then assign that out. That is part of how Agile works. If something breaks, go back and fix it during the next sprint!

Wednesday, July 25, 2018

Today our group went over the requirements for homework assignment 4. The task this week was to develop an implementation of our system based on the architecture that we selected in assignment 3. We selected the repository architecture, so we will be going off of that.

The deliverables this week included a UML class diagram, which we were able to create by tweaking our class diagram from the first assignment. We went over the packaging and one of our team members specifically analyzed the cohesion and coupling aspects of it and went over that with us. As a group we felt that incremental development made more sense for our project since there were many smaller components that could be developed individually and work on top of each other. Since the customer had laid out clear requirements up front, it wouldn't make sense for us to develop in cycles. We don't anticipate new features or cycles until the app goes live, and there is a lot of development to do until we get there.

When going over the design patterns that would be useful in implementing the system, we decided on the Builder, Adapter, Façade, Memento, and Observer design patterns. Different design patterns would be useful for different components, so this exercise was a great way for us to understand that there can be many different ways that components are built, even though in the end they all get tied into one unified system.

We selected Use Case 1 from HW2 for the updated sequence diagram. Now that our system is more defined at many levels, we can make the sequence diagram more detailed and actually show how our classes and components interact with one another.

We found that the interfaces we need are one to connect our server to the Google Maps API, as well as another interface to connect our server to a payment processing API. Obviously, the user interface in the app is something else that will require a clearly defined and developed interface. When thinking about exceptions that are likely to occur, we all agreed that the interfaces with other APIs are the likeliest parts of the system to have issues. The APIs might be down, they might not

parse our data correctly, etc. I used to work in software integrations so I'm well aware just how many issues can arise from data transfer. In addition to API issues, however, we may also have issues with our database. We would set up our database so that any time the server attempts to write to the database, if there are any issues, the entire transaction will be rolled back. This will protect the integrity of our data and ensure that we do not have corrupted records.

Compared to last week, this week was a bit easier. The concepts, other than the building patterns, were not extremely new to us, since a lot of it was built on previous work in HW2 and HW3. Like usual, everyone in the group was able to meet and came prepared.

Sunday, July 29, 2018

This week further cemented in my mind the importance of documentation for requirements and design. Things that are not explicitly written down will be forgotten or misinterpreted, fact. Documentation is essential to prevent those from happening, because people forgetting or misinterpreting requirements can doom a project from the start.

Practicing strict discipline in the development process makes a big difference in developing reliable and trustworthy software because the software itself will be a result of the effort and attention to detail that was put into it. As the old saying goes, "you only get out what you put in". A software system that is built without putting the effort into clearly defining the many components ahead of time is bound to have issues come up.

I believe that for developing a base platform, having clearly defined requirements and design is essential. There are so many pieces that need to be implemented, and designed so that they can be modified in the future, that simply winging it will be inadequate and lead to plenty of mistakes. However, when developing new features and functionalities, I feel that using the agile philosophy makes sense. Features need to go out frequently and not take ages to develop. If one feature has issues, we can simply roll it back and fix it. There will not be a bunch of development on

top of feature that makes it difficult to undo. Documentation for new features does not need to be as detailed as the documentation for the foundation of our project. Essentially, I believe that original requirements and design out to be meticulously documented, but when moving onto the maintenance and further development stages of the SDLC, that is not as important. Instead, getting features out on time and without issues should become the new primary focus.

One thing organizations can do about lacking documentation when using agile methods is set aside time when developers can focus on documentation, or require a page write-up about new features, or perhaps even create a new team that works alongside the development team and writes documentation on whatever the developers are working on. Personally, I like the documentation team approach the most. Some people enjoy writing technical documentation and others enjoy writing code. Letting individuals do what they know best is the smart way to run a business.

Wednesday, August 1, 2018

Today, our group met with the customer to go over what we have worked on so far. Before we actually met with the customer, as a group we came up with user stories and shared them on a Google document. During our meeting with the customer, he told us how he felt about our current progress and asked us some questions about the documents that we have submitted so far. We were also able to split our user stories into 3 groups of those to implement this coming week, those to implement the next week, and those that will not be implemented at all.

Along with the customer, the team determined that we would implement 3 user stories this week: “Exchange points for currency to donate to a charity”, “Register for the app”, and “Report a user”. I would create a data sequence UML diagram for the first, Alex would create one for the second, and Kevin would create one for the third.

Next week, we will implement 4 user stories: “Verify that a user cleaned an area”, “Earn badges/medals”, “Comment on a user’s activity”, and “Submit a photo when marking an area as

cleaned". I would write the plans for the first one, Alex would write the plans for the second one, Hitesh would write the plans for the third one, and Kevin would write the plans for the fourth one.

That left 4 user stories that we would not implement at all: "Mark an area as cleaned", "Donate to a charity", "View badges and progress towards badges", and "Challenge another user". After the meeting we had a good idea of what everyone's job was for the week, so we went off to work on our separate tasks.

Sunday, August 5, 2018

Today I compiled the work that we split out in the meeting into the final format. When going over the assignment, I understood just how many moving parts there are to a user interface, and why all of the user stories must be defined. We want the user experience to be as detailed as possible before we even start development. In doing this task I was able to think about some of the key questions for the week.

It is essential for each development team to have a customer rep participating in at least scheduling the work because the customer is paying for our time and we want to make sure that we develop the system in the order that the customer would be ok with. In addition it makes sense to have the customer present in general. In our meeting, the customer had a few questions and clarifications that he brought up. As business partners we always want to ensure that any sort of miscommunication is avoided, so having the customer here allows us to be explicit about everything that is not immediately clear.

Weekly work meetings make the project more efficient because you essentially briefly go over what you're done and assign out tasks for the next week. In my opinion using a divide and conquer technique is effective for completing a lot of work, so it works well for our group. I find that we are able to get the work done quite efficiently by just giving individual people different responsibilities.

Group members know what everyone else is doing because we go over this in the meeting. By being explicit and doling out tasks fairly and appropriately, we are able to avoid having group members having overlapping responsibilities, that way work is not done twice.

When we have system-wide requirements for a project with many smaller parts, we will come up with a way to develop and implement the system-wide requirements first. All other components will need to work with respect to the global requirements, so it makes sense to plan out the global ones first. Once the global requirements are clearly defined and implemented, it is easier to include their functionality into the smaller components. While we clearly cannot define every single global requirement before other development begins, we can maximize efficiency by defining as many as possible at the beginning.