Monease

Final Reflection

Megan Brust
Matthew Knobloch
Jude Odafi
Steven Toub

Impressions

Overall

Taking a software product from its infancy stage of an idea through implementation was a good exercise in deconstructing everything that needs to be taken into account when building a piece of software. The steps that we went through, such as creating UML and architectural diagrams, writing peer reviews, and then finally coding what we designed allowed us to witness the whole design process. We are glad to have had this experience and will definitely take the lessons learned forward with us to our next job assignment, group project, or solo venture.

Team

The team made the project easier and more enjoyable. Working with three strangers, online, who only know that they are each taking the same graduate course, and coming together towards a common goal and actually implementing that goal is a tough task. It required a lot of good communication skills to make sure that we were all on the same page at each phase of the project.

Suggestions

We wish that there were more assignments with direct feedback related to the design of the software besides the second assignment. We feel that it would have been more beneficial for the final term project if we had done more design work and less of implementing the project.

Responsibilities and Contributions

We communicated and met as a team via email, messaging, and video chat. Steve and Megan were our engineers while Matt and Jude were the architects. Every team member contributed to the success of the project. During the initial design phase everyone brought their previous experience in software development and patterns that they had encountered before to the table. Once we started to get more detailed in the design, we divided up the architectural diagrams between the different members of the group.

For the development of the software we were able to divide it up into manageable portions that could be worked on independently and simultaneously. These portions were the main GUI, the graphs, reading a statement, reading from and writing to internal files and the data structure for storing the information. GitHub was an essential resource for us to make sure we were all on the same page and that we were able to test the code on our own, primarily using Eclipse and IntelliJ Idea to compile and run our code. Once the individual development was completed, we were able to combine all the pieces together seamlessly into our final product. For our final reflection document we each wrote our own reflection of the assignment and our software design individually. Once we all had our own thoughts we combined them together to get an overall reflection that expressed everyone's ideas, comments, and criticism of the software and design.

Lessons Learned

We learned that due to a short deadline, we had limits on how much we were able to design and implement. We did not get to implementing the estimates feature in our application (inferring from previous budgets what future budgets might look like). We also considered having Monease available as a web application, but decided that we did not have adequate time to build and test it thoroughly.

However, we learned we could save time in the development if we had a thorough, well-thought out design beforehand. Considering that we began with a simple piece of software and a solid design, the development of the software was not too challenging. If we had designed it differently it could have been much more challenging to develop.

Regardless of how well-thought-out your design is, it can be hard to stick to that design when you actually implement the software. In our experience we designed using the Model-View-Controller pattern. A key component to that design is having separate components for each of the three portions of the pattern. When we started to write code the separation between the three components began to blur together. We would have had to either redevelop the software with the initial design correctly or using what is already coded to act as the framework for the new design. This is especially difficult if it is a portion of code that gets used frequently.

Additionally, we found that completing code reviews before testing saves time overall. We found the statistics provided from IBM and Jones were true in our project, just to a lesser extent since it was a smaller project. This was true for us when we were making sure that some of the if statements always lead to a valid path. When populating the combobox with the days of the month whose transaction is being edited we had to make sure that not only will days always be

populated but the correct days will be populated. We were able to review the code and follow all possible paths to verify this before any testing took place saving us significant amounts of time when testing this portion of the code.

Something we were not anticipating but learned along the way is that we are going to miss things the first time around. When we were initially designing the software, we did not consider security at all. For many people this is considered a necessity for a piece of financial software. Had we thought about it initially, we would have made the time and design decisions to implement it. Since it was brought to our attention by our peers at a later stage in development, we collectively decided that we would rely on the security measures already put in place by a personal local system and wait to implement more rigorous security features into a future release.

Patterns and Vulnerabilities

For our project we mainly focused on using the Model-View-Controller (MVC) pattern. At the highest level, we designed our architecture to separate portions of the code for viewing, modifying data, and storing data. In the actual implementation, however, we combined a few of the portions together and were not as separate as they really should have been. For example, we combined multiple components of the software into the GUI. Not including the reading from and writing to files, much of what should have been the controller package got mixed in with the view package. It would have been better to separate soliciting information from the user using the GUI from writing that data back to our model, the month linked list. The different packages also were blurred together with the model and view packages. While we did have different classes and function for how we were storing the data (MonthClass, MonthList, TransactionObject and TransactionList) the location of the actual linked list was in the main GUI class. To truly have followed the MVC pattern, we would have separated the storage of the data to its own class and not in the GUIs. Upon reflection, we should have stuck to our design or considered reevaluating our design once we started to veer away from the initial design.

Looking at our software at a high level, we wanted it to be a stand alone piece of software that would help a user, anyone with multiple bank or credit card accounts, manage their current financial situation. The software would interact with your financial institutions, through reading statements, and present the data to you in a variety of ways like reports and graphs. We feel that the software that was implemented followed through with this intended high level design. As we dig a little deeper into the design we settled on an MVC design, as described above, with the three main containers. The view container was to be composed on the main GUI where the user can navigate through the options, input GUIs, and GUIs to display the data, in this case graphs. All of the GUIs besides the main GUI would be access from the main GUI and then interact with

the other containers of the design. Our model container was to be composed of two main components, our month linked list and our transaction linked list, with each month node in the list containing a transaction linked list. Each of the components in this container have classes for adding new items to the linked lists as well as retrieving items from the linked list. Finally our controller container was to be composed of a few components. They were to take the data that came from the view and write it back to the model. For us in our software, we have function instead of classes that took care of these actions.

The major vulnerability that we found in our software is the security of your financial information. As we previously mentioned, we initially did not design with security in mind. This is because we were focused on a simple, working solution that would run locally on an individual machine where internet access would not be required. However, when we considered that eventually the software was to be more closely integrated to a bank or made into a web application then we would have to have much more security implemented. We would also have to design for this much more as well.