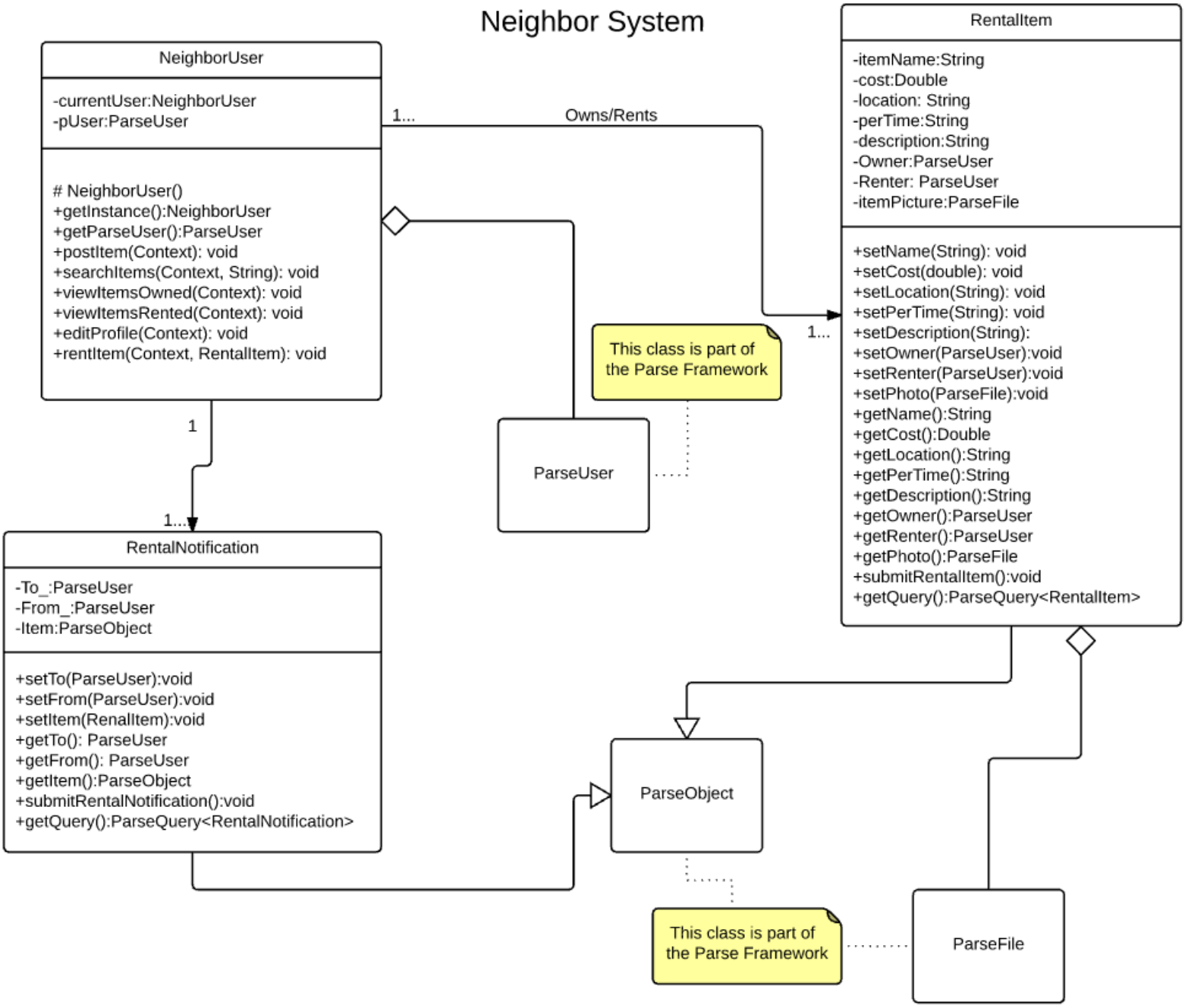
Neighbor

Project Part 4: Final Report

Brennan McConnell, Kade Cooper, Patrick Andresen, Neil Nistler

Overall, building Neighbor was a wonderful learning experience which resulted in a functional and polished Android application built around solid Object-Oriented Principles.

1.)



The following features were implemented:

- Signing Up
- Signing In
- Signing Out
- A home-screen for a user which displays information relating to them
  - Alerts (such as another user renting one of the items you own) are displayed on this home screen.
- Posting an Item
- Searching for an Item
- Renting an Item
- The user can view Items Owned
- The user can view Items Rented
- The user can edit their Profile.
- The user can view other Neighbor Users' Profiles

Certain features / stretch goals we were unable to implement simply due to time constraints. Some features that are missing are the ability for more in-app negotiation over renting an item. For example, specifying a security deposit stipulation when posting an item or allowing for more of a negotiation about price between two users were a couple stretch goals of ours that did not get implemented. Additionally, I wanted to use the Context IO API to be able to send an e-mail from one user to another when the first user rented an item owned by the second user. And finally, the largest stretch goal by far that did not get implemented was a way to be able to pay via PayPal or Venmo within the app. Ultimately however, we feel that we built an outstanding application that certainly lives up to our expectations. It is fully functional and propers as would be expected. With a little more work and polish it may just be ready to publish to the app store!

This final report would not be complete without touching on how doing the initial class digram did indeed help prior to the actual coding. It is much easier to translate ideas into code when you have a specific blueprint you are following. If we had simply dived into the code and then started building our idea as we went, there would have been many disjointed ideas and refactoring several times would be needed. By putting in effort during the planning stages (the analysis and design steps), time was actually saved later. From the initial class diagram (and other planning) it was very easy to simply map our diagrams and ideas into functioning code. By considering various solutions to the problem we set out to fix before actually implementing any of the ideas we gained a solid understanding of where the difficulties would lie. We were able to define clear project goals with straightforward solutions and implementations via during the analysis and design phase. I certainly have been guilty of jumping straight into coding the solution on certain projects in the past, and while they work, the code ends up looking completely disastrous which is very error-prone and it would be hard to either add features or change features. By spending time carefully analyzing the different use cases for the system and how we planned to implement them, we were able to write clean and concise code that still maintains the ability to be changed or added to.

Jumping write into the coding portion would be similar building a house without a blueprint and without planning about what parts you might need. Clearly it is of utmost importance to plan, and for software engineering, that occurs during the Analysis and Design stages of the SDLC. We were able to point out flaws that might occur early (trouble-spots), discuss the requirements of the project, and consider what the user would deem most important early on in these stages. Ultimately, these stages are not only incredibly beneficial to any Software Project but completely necessary.

2.)

We did make use of design patterns in our code and additionally have an area where it may have been helpful to make use of one additional design pattern. Lastly, we used an ORM (which you mentioned is an interesting aspect of our project as well as pertains to the class since we did discuss the Hibernate ORM).

Our system has a NeighborUser class which is basically - as you might suspect - a class for users which sign up. The necessity for a singleton became evident as only one NeighborUser may ever be logged into the application (per phone) at a time. It would not make sense to have multiple different users logged in at the same time on the same phone whereby posting and renting items would break the application. When a NeighborUser is using the app, if an item is posted, it is obviously an item belonging to the NeighborUser currently signed in. Much like the president analogy used in class, only one NeighborUser may exist at a time, and if one user logs out (on a particular device) and another user logs in, there still only exists a single NeighborUser.

Another design pattern which we made use of was a facade. We have NeighborUser which has an attribute ParseUser (which is a class provided by the Parse framework). This ParseUser class has a ton of complex functionality. Thereby, the ParseUser is a private attribute of NeighborUser - which provides an interface to ParseUser. The interface provided only contains some of the methods which are application needs rather than all of the methods that a ParseUser actually has. NeighborUser contains simplified functions which can be called by our client code and thus the complexity about the ParseUser class is concealed from the client. A simple example is NeighborUser has a method saveProfileChanges() which conceals the process of a ParseUser connecting to the database, notifying the database of the changes made, and submitting a background request for saving the changes (accomplished via the ORM). All this complexity is hidden and an interface is provided only for the methods which the client may need to use.

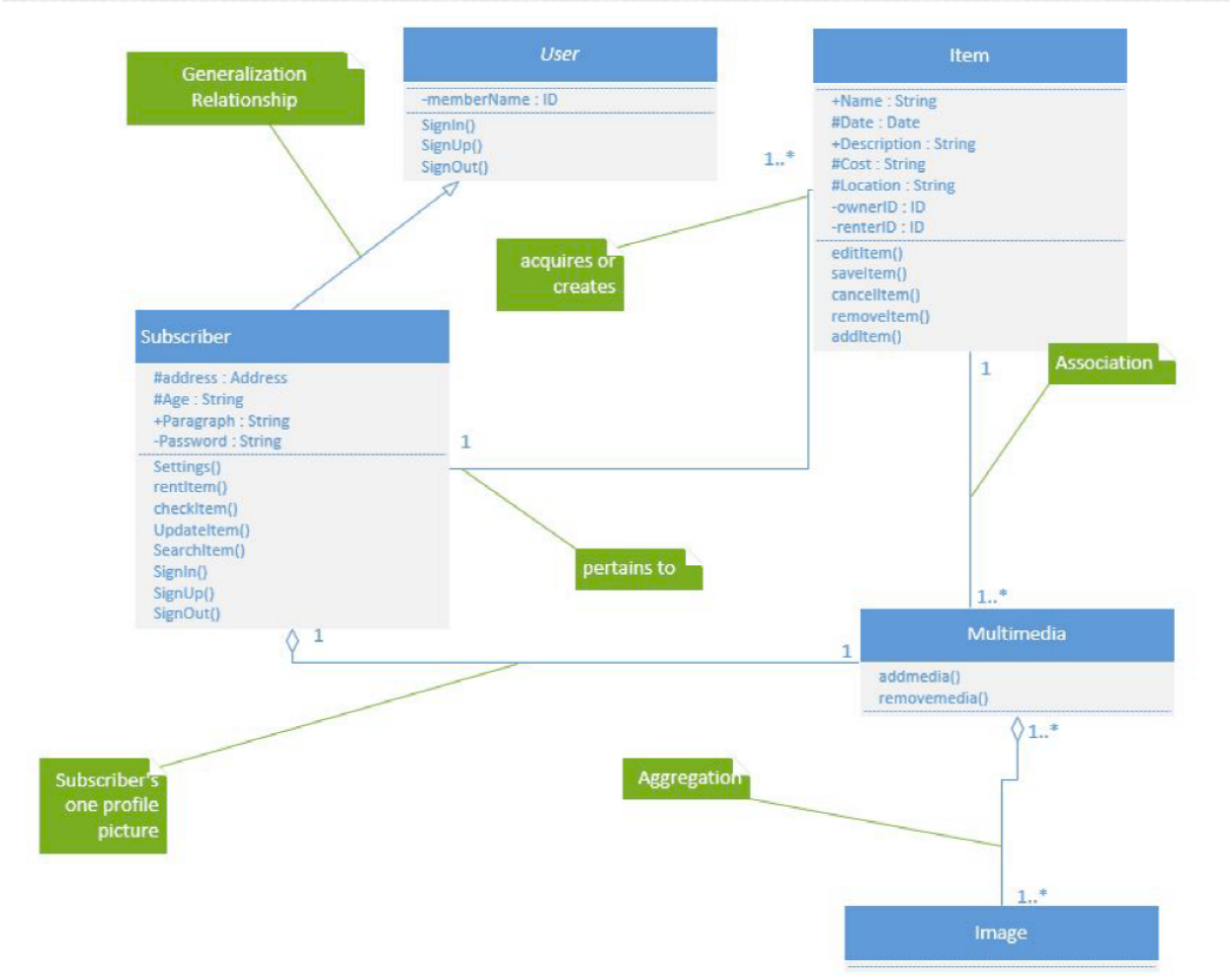
One pattern which we could have used but did not is the Builder pattern. Our RentalItem class has many attributes which are other classes and is thus a fairly complex aggregation. Had it been any more complex, we definitely would have felt the need to used the Builder pattern but with the current implementation we were simply not quite to the point of needing it. The finer control over the construction process of our RentalItems that the Builder pattern would provide would remove the painstaking

process of a complex constructor alongside multiple use of multiple setter functions. It would have been more proper to simply use the Builder Pattern to construct this RentalItem object and have the client then be able to grab the object knowing fully well that it is a complete object and no errors will occur when making use of the object in the future.

Lastly, I would like to discuss the Parse ORM. In class we discussed Hibernate and implemented a simple and straightforward hibernate example for our quiz. With this android project we absolutely needed to use a database and did not want to handle all of that complexity ourselves. As mentioned throughout this document we resorted to Parse which provides cloud-based backend services for our application. Of these services, there exists an ORM which allows us to very easily map our classes to database tables and our instantiations (objects) to rows inside said tables. Although the syntax is different than Hibernate, the ideas and fundamentals are identical. We build a class which is to represent a new table (and if that table happens to not exist, it will be created for you upon executing the code), and use a series of Parse 'set' methods which act similar to Hibernates 'Id or Column' syntax. Then when the code is ran, whenever an object such as this is instantiated, these attributes are stored server-side and saved to our cloud-based database. This was a wonderful feature that made our lives as programmers much easier. Ultimately, we felt that we made great use of design patterns and OO concepts where it was necessary, and that this learning experience will definitely have us considering design patterns the next time we approach a problem.

3.)

(Initial class diagram from project part 2)



We did indeed still need to tweak our initial class diagram based off of our own feedback and vision for the project. Apart from a revamp of names across the system (which was changed to add more clarity and descriptiveness to Classes and Methods), we also made some changes due to the fact that we switched platforms. Initially we planned on developing a web application using a JS framework. In the end, we went ahead and developed and Android application. Alongside the Android Framework we used Parse (which provides a backend cloud-based services alongside its own ORM framework). Some of our classes now extend 'ParseObject' which is how the Parse ORM works. When you extend a class as a 'ParseObject', this allows it to be saved to our backend in the database and provides the respective and necessary functionality which is already built-in and handled server side by the Parse Framework. Additionally we were able to

remove 2 of our planned classes because Parse now had attributes for ParseObjects which would replace that functionality. For example, we had a multimedia class planned initially which was designed to hold pictures. Parse provides a ParseFile (which implements Serializable and therefore can be used to store our pictures) which can actually store the byte array and can be an attribute to a ParseObject. Naturally this added flexibility reduced the necessity for our Multimedia class. Additionally, some other things changed such as certain methods were removed because they were stretch goals which were not completed. Many attributes were changed to private (this was simply poor planning/ignorance when completing our initial class diagram) and thus was corrected quickly when we began implementing the idea. We can conclude that many changes to our class diagram were needed due to the addition of us using Parse as a backend service.

Another aspect of our final class diagram compared to our initial class diagram that when we completed Project Part 2 we had not yet learned about Design patterns and thus had not yet considered any of them in our solutions. This, combined with all of us being complete novices when it came to UML, made some major changes to the Class Diagram quite necessary. It was obvious that many mistakes existed in our first diagram and the whole idea of analysis and design was very new to all of us and it definitely came with a learning curve. In hindsight, and for the future, it will be easier to consider where errors/difficulties in our design might exist and thus apply the design patterns during these analysis and design stages. Additionally, we would all hope to spend more time painstakingly analyzing the work that was done during these stages to make sure they are completely correct and error-free so that corrections are not necessary during the coding implementation.

In the end, a class diagram can indeed make life much easier… if done right.

4.)

It needs not be stated that we all obviously learned many hard-skills about analysis and design such as all of the UML notation. But apart from learning UML, we learned that the analysis and design steps in the SDLC (Software Development Life Cycle) are crucial elements to having a clear direction and solid understanding of a system. By going through the Analysis portion with: Use Cases, Requirements, and Activity Diagrams we created a good, general framework which helped us better understand the vision of the system and aid in user satisfaction for our target audience. In addition, by laying out the Sequence, Class, and State Machine diagrams in the Design process, the separate parts of our system became connected in front of us to form the functional design of our system. This further aided with how the final system would function and how the users would interact with the system. Furthermore, the planning stages (Project Part 2) provided one of the biggest learning experiences of this class (perhaps the biggest). We obviously needed to implement many UML diagrams but what we learned was the necessity for detail and the utmost importance of avoiding contradicting diagrams and making sure that all diagrams represent the same system which is

functioning in the same way. Good planning in the analysis and design steps saves a vast amount of time later. As is the motto, “measure twice, cut once”. This is so very true when developing a software system. With proper UML diagrams in place, it is pretty straightforward to map the diagrams into properly functioning code and a usable system. One difficulty we had during this previous stage was being sure there were no contradictions throughout the design. In the future, we would desire to spend even more time on the analysis and design steps so as to fully prepare for building the system and make sure that the vision of the system is exactly what it needs to be. Thus, the UML diagrams made everyone’s job straightforward when it came time to code. And later, the only thing left to discuss was the UI designs for the system and what would look polished to the end user. Consulting the designs is always necessary when you forget what needs to be accomplished / how you intended to accomplish it. Overall, the main thing we learned is that there exists a high upfront cost in the Analysis and Design stages, but this time is most certainly well spent.