Project Title: TAPP

Team Goose

Team Members Names (Quest ID):

Anurag Joshi (a35joshi)
Gunin Khanna (g3khanna)
Daniel Weisberg (dweisber)
Xinjue Lu (x89lu)

Architecture Styles

The architectural style that was primarily used for this project was the Client Server architecture. This can be further broken down into two parts. Firstly, there was Client Server interaction between the user's phone and the Authentication and task retrieval component located in Google Firebase. Secondly, there was also Client Server interaction between an NFC tag and an Android phone.

Client Server Architecture between the Phone and Firebase

This part of the report primarily focuses on the four right-most vertical components in Figure 2. These include the Android OS, the TAPP Processor, User Authentication, Check User Data and the User's Database components. The components that would be on the client are User Authentication, Check User Data, the Centralized Controller Logic. The components which would be on the server is the Users and NFC ID Database which will be on Google Firebase. The connector between the client and server components happens through TCP/IP.

One of the functional properties described in our initial proposal was user and NFC tag registration and unregistration. This property said that users must be able to register or unregister themselves and NFC tags to our service through a secure and encrypted channel. To achieve these functional properties, we have a User Authentication component in our architecture diagram. This component links the TAPP Processor with the method call that checks for the user data in the user's database located in Firebase. If the user or NFC tag exists, then we authenticate them and allow the processor to perform the specific actions. The unregister functional property is satisfied too because users can either delete their accounts or delete an existing NFC tag. The functional property to display current NFCs, display tasks under each NFC is also fulfilled by pulling the required data from the Firebase database component of the architecture component diagram.

One non-functional property mentioned was security. It was mentioned that unauthorized users should not be able to trigger actions if they tap someone else's phone against an NFC tag. This is taken care of by the Android OS component of the architecture diagram. This component ensures that while the phone is locked, no unauthorized user will be able to open our application and perform unnecessary tasks. Even then, if an unauthorized user gets past this stage, the user authentication component in our architecture will prompt them to enter an email address and a password which they won't have access to. Furthermore, the project uses Firebase Authentication which encrypts personally identifiable information such as usernames, passwords, email addresses, phone numbers etc. using HTTPS and AES-256. AES-256 Encryption ensures that private data is always safe from eavesdroppers as there are 2^256 distinct possibilities. Breaking the AES-256 scheme is next to impossible. Firebase Authentication also logically isolates customer data to make things complicated for an attacker.

Another non-functional property mentioned was performance. Firebase, a Google product, is built using the NoSQL structure that uses JSON objects for its document storage. [3] This is demonstrated in Figure 1. That means Firebase is a non-relational database. Firebase doesn't have any concept of rows and tables. There are instead fields and subfields. Using JSON objects makes the application scalable and maintainable. This makes Firebase much faster than any other type of database. Clients who use the NoSQL structure prioritize quick and efficient data retrieval.

Furthermore, Firebase uses promises and callbacks. This means that our application would do something else while it waits for a response from Firebase. This ensures that we do not waste any time waiting for data from the database. However, it is also very crucial to properly structure the data object so that redundant calls are not made and each piece of information is returned only once with no duplication. We made sure that each piece of information is unique and connectors are

only used when required to link data existing in two separate JSON objects. Overall, it is safe to say that using Firebase through the Client-Server architecture will increase the application performance, maintainability and scalability.

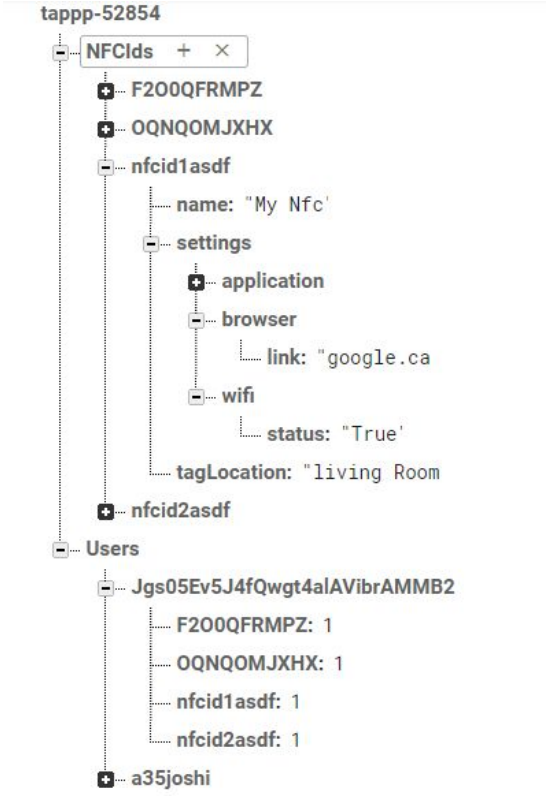

Figure 1: JSON style object storage in Firebase [3]

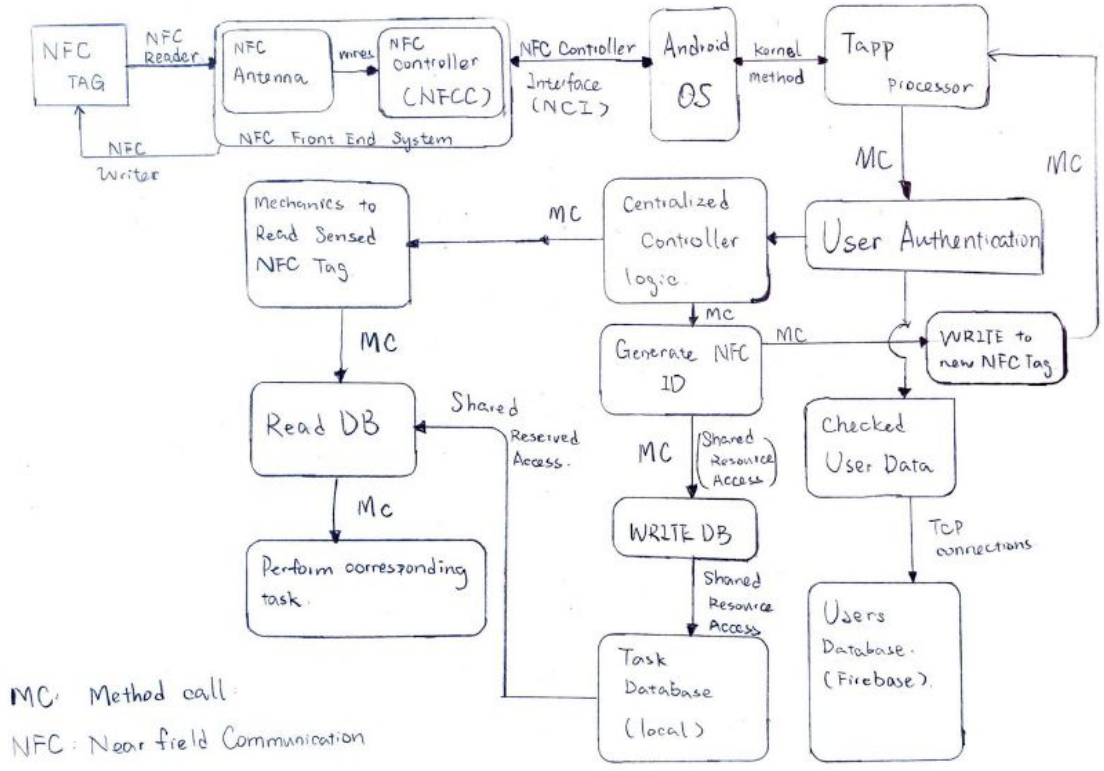

Figure 2: Client Server Architecture Diagram for TAPP

The User Authentication component in the architecture diagram in Figure 2 can be enhanced further as shown in Figure 3. There are two main classes that deal with the Client Server architecture portion of our project. They are the LoginActivity.java class and the SignupActivity.java class. They are represented as the signInUser and the signUpUser components in the given diagram respectively. In this case, the client is the Android device consisting of the two activities while the server is an instance of Firebase running on the Google Cloud Platform, as mentioned earlier.

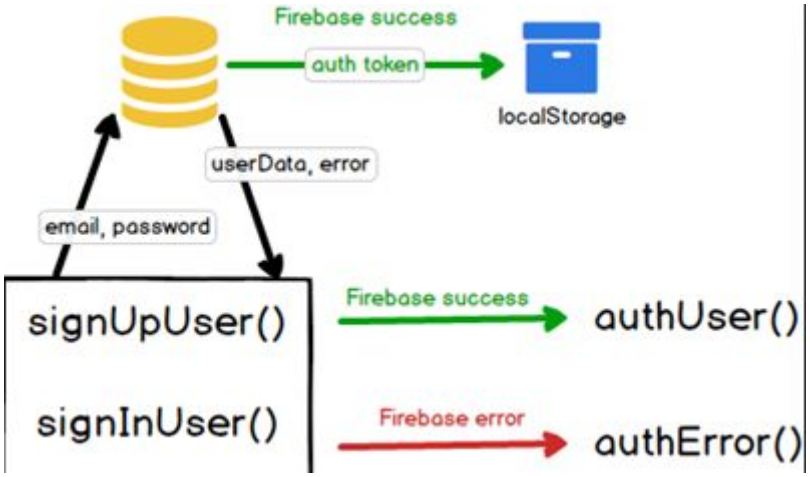

Figure 3: Firebase Authentication Architecture Diagram [3]

Client Server Architecture between the NFC Controller and Phone

The Client-Server architecture is adopted for all NFC communication handled by TAPP. This part of the report primarily focuses on NFC Tag, NFC Frontend system that includes the NFC Controller and the NFC antenna, Android OS, and the TAPP Processor components described in Figure 2. The client, in this case, can be recognized as the Android device that is running the TAPP application and the server can be recognized as the NFC Controller (Android.nfc) package that provides access to the NFC functionality, allowing applications to read NFC Data Exchange Format (NDEF) message in NFC tags.

One of the functional properties described in our initial proposal was the implementation of the NFC protocol i.e. being able to establish a connection with an NFC tag and being able to read and write from the NFC tag. Both read and write functionalities are described in detail below.
Primarily, the NfcAdapter class is used to get the default NFC adapter for the Android device (client) running TAPP. Reading NDEF data from an NFC tag is handled with the tag dispatch system. The tag dispatch system recognizes the discovered NFC tags and further categorizes the data to perform the desired task via intent filters.

Android-powered devices are actively looking for NFC tags when the screen is unlocked unless NFC is disabled in the device's Settings menu [2]. The tag dispatch system provided by Android analyzes scanned NFC tags, parses them, and tries to locate applications that are interested in the scanned data. The tag dispatch system does the following:

1. Parses the NFC tag to extract the MIME type or a URI that identifies the data payload.
2. Encapsulates the MIME type or URI and the payload into an intent.

3. Initiates an activity based on the intent.

Android supports both NDEF and non-NDEF standards for the NFC tags. It offers extensive support for NDEF format. However, non-NDEF format tags are supported by the Android .nfc.tech package thereby, fulfilling the non-functional property – Compatibility mentioned in the proposal document. NDEF data is encapsulated inside a NdefMessage that contains NdefRecords. To extract the MIME type or a URI that identifies the data payload in the tag, the android device (client) scans an NFC tag containing the NDEF formatted data and further reads the first NdefRecord in the NdefMessage. This is in accordance with the compatibility non-functional property mentioned in the initial proposal for this project.

The tag dispatch system uses the TNF and type fields to try to map a MIME type or URI to the NDEF message. If the system recognizes the same, it encapsulates that information inside of an ACTION_NDEF_DISCOVERED intent along with the actual payload. For non-NDEF formats, ACTION_TECH_DISCOVERED intent is used instead.

The tag dispatch system works as follows:

1. Initiate an Activity with the intent that was created by the tag dispatch system when parsing the NFC tag (either ACTION_NDEF_DISCOVERED or ACTION_TECH_DISCOVERED).
2. Tries all possible intents until an application filters for the intent.
3. If no applications filter for any of the intent, do nothing.

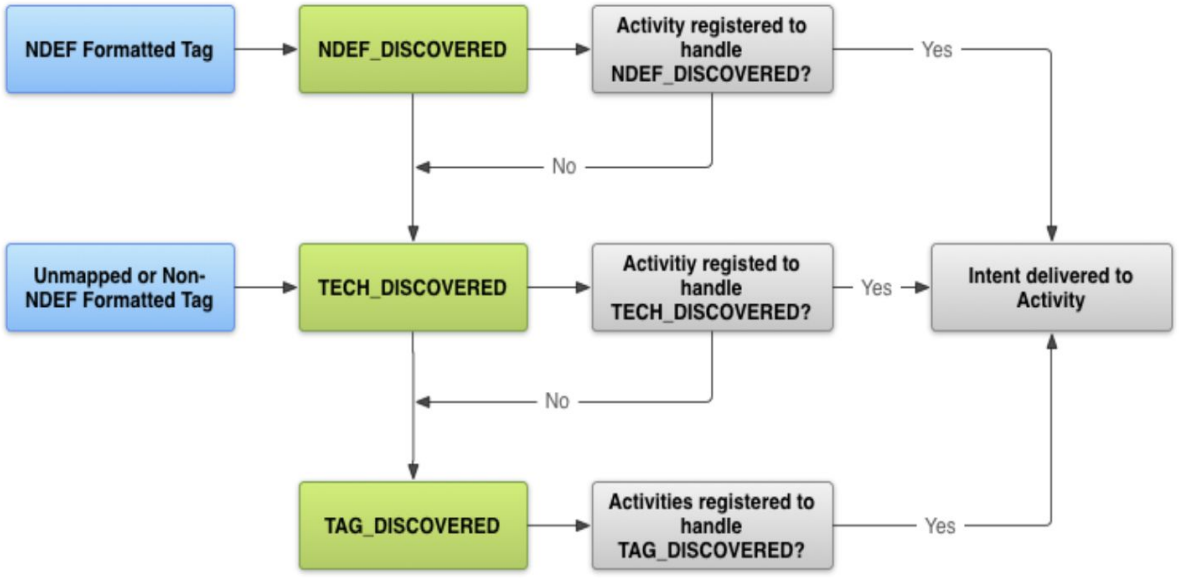

Figure 4: Tag Dispatch System Architecture Component Diagram

The definition of the strategy pattern states that it is a behavioral design pattern that allows us to select a specific algorithm at runtime based on the needs of the client. In our project, we had to modify this definition since we wanted our code to run a specific collection of algorithms instead of a single algorithm. NFCTriggerActivity.java gets triggered every time someone taps their phone against

an NFC tag. In our project, the code to perform each action and the associated The above-described reading mechanism is implemented in NFCTriggerActivity.java and NFCManager.java.

In addition to being to read data from the NFC tag, in order to support the functional property of NFC protocol, the application must also be able to write data to an NFC tag. The message could be an NFC_ID to identify each card uniquely, thereby, ensuring to perform tasks only associated to the specific NFC tag given the NFC_ID and satisfying the non-functional property of dependability. The writing mechanism described below is implemented in NFCWriteActivity.java andNFCManager.java.

The NfcAdapter is initialized in the onCreate method (NFCWriteActivity.java) and further enables the pendingIntent to run in the foreground of created activity via the onResume method.

The onNewIntent method is overridden, which is activated when the NFC tag is tapped against the device, given the intent filter that is set. The custom writeTag method (NFCManager.java) takes the tag, and the message to be written as the parameters and uses NdefFromatable to format the message to be written and writes to the tag using the writeNdefMessage method.

Design Patterns:

The design pattern portion of our system primarily consists of two main behavioral design patterns. They are the Strategy design pattern and the Command design pattern.

Strategy Design Pattern

The strategy pattern is a behavioral design pattern that allows us to select a specific algorithm at runtime based on the needs of the client. In our project, we had to modify this definition since we wanted our code to run a specific collection of algorithms instead of a single algorithm. The NFCTriggerActivity gets triggered when a user taps their phone against an NFC tag. When this happens, the code built into this class reads the content of the ID attached to this NFC tag. The code then gets a list of actions registered for this NFC tag from the database. In our project, the code to perform each action and the associated algorithms are implemented as separate independent classes or strategies.

There are multiple reasons why we decided to use the strategy design pattern. Strategy pattern makes the task of adding extra features and functionalities to existing code easy. In our case, we currently support a certain set of actions that get triggered when the user taps their phone onto an NFC tag. In the future, if any other developer wishes to add more features to this project, then the strategy pattern makes this task very easy. There would be almost minimal to no modification done to the existing code. Another key aspect of the strategy design pattern is that the code picks a set of algorithms at runtime. This is very crucial for us because we don't know which NFC tag would the user tap next and therefore it is almost impossible to predict the next set of actions to be performed. This problem is automatically taken care of at runtime by the strategy pattern. Another reason why we decided on using the strategy pattern is that it promotes the reuse of code and prohibits code duplication. For instance, one of the actions we support is turning on WiFi. In our implementation, we had to write the logic to perform this action only once instead of duplicating it for every NFC tag. Strategy pattern is used when we have one object and we want to perform different actions using this object. In our case, the Firebase instance returns us a single data object and based on the values given in this object, we perform different algorithms at runtime. This suggests that the Strategy pattern would be a decent fit for this project.

We had considered using other design patterns as well to accomplish this task. One of them was the visitor pattern. One key drawback of using the visitor pattern was that the visitor must be aware of all the hosts and must provide operations for each host. That means, we would need to know what set of actions are going to be performed by an NFC tag even before we approach it. This involves additional code and maintenance logic.

Any typical implementation of the Strategy pattern consists of a Context object, a Strategy interface and multiple Concrete Strategy classes. In our case, we have a ContextObject.java class that is called by the client (NFCTriggerActivity) when the user taps their phone against an NFC tag. The context object provides a link between the settings object returned from Firebase and the concrete strategy classes. The strategy interface (Strategy.java) provides a link between the Context object and a concrete strategy class. The concrete strategy class (for instance, wifiToggling.java) provides an actual implementation to the algorithm. These algorithms are very likely to differ from one another in a distinct way. In our case, the algorithms basically have the source code to implement various

actions such as opening a browser or turning on Wi-Fi etc. This is demonstrated in Figure 5 as a UML structure.

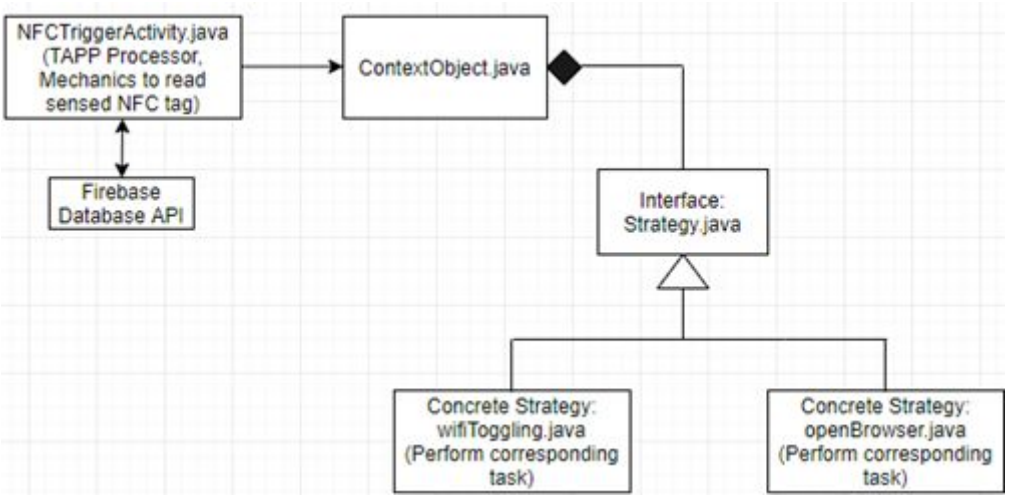

Figure 5: Class diagram for our project that shows the interaction between various classes and external APIs when the user taps against an NFC tag.

Figure 6 demonstrates the interaction between various classes and external APIs when the user opens our application. The Firebase API classes such as the Database UI API, Authentication API and the Database API reside on Google's servers. The NFCAdapterAPI resides in the NFC Controller in the NFC tag, which is also treated as a server from an architectural viewpoint. All the other classes reside on the user or the client's phone. A mapping between the design level entities such as classes and the components described in the architecture diagram is given through parentheses in Figure 6.
The physical location of the classes documented in the class diagram in Figure 6 will be the end user(client)'s device. Our system uses the Firebase API to read and write user data to the database.

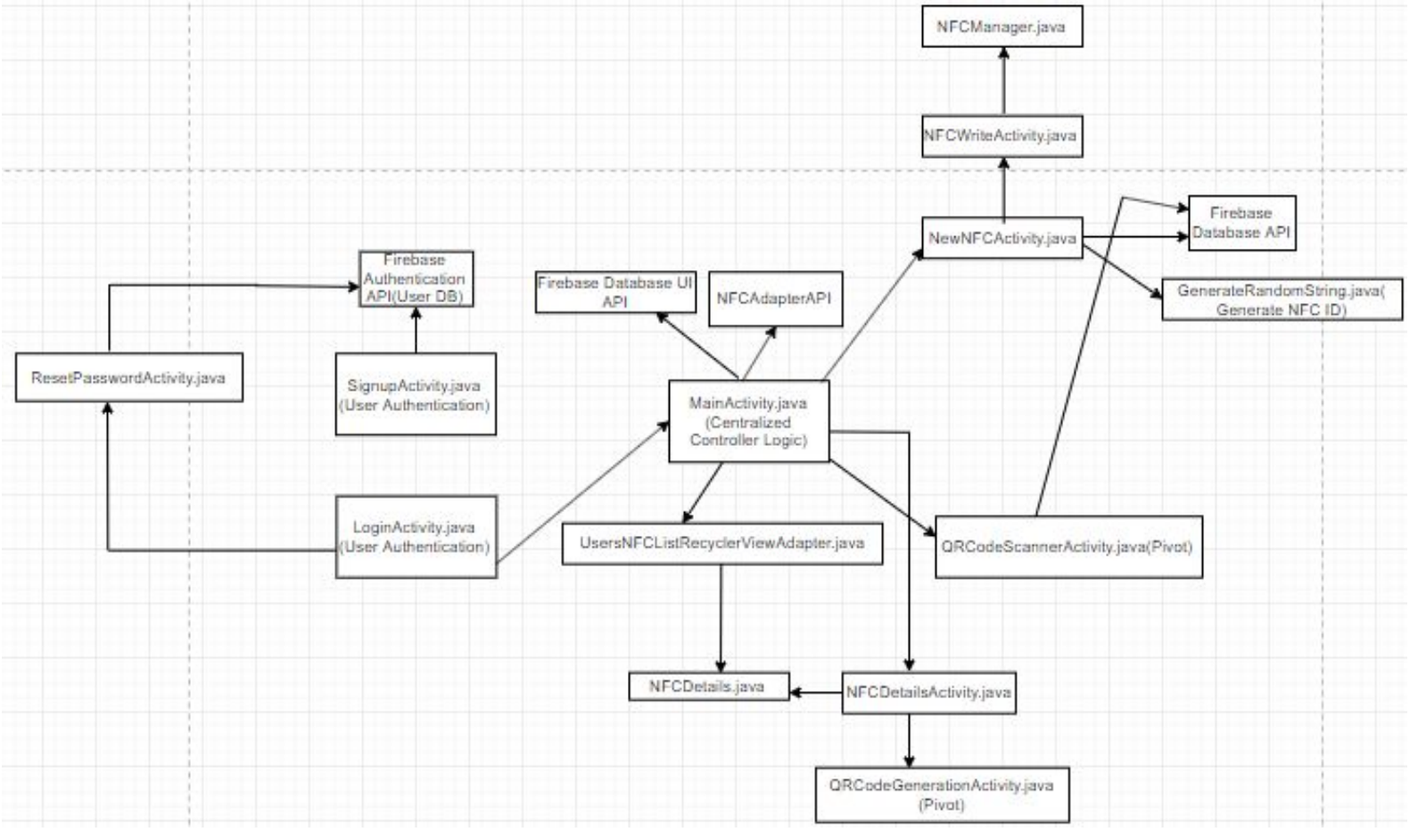

Figure 6: Class diagram for our project that shows the interaction between various classes and external APIs when the user opens our application.

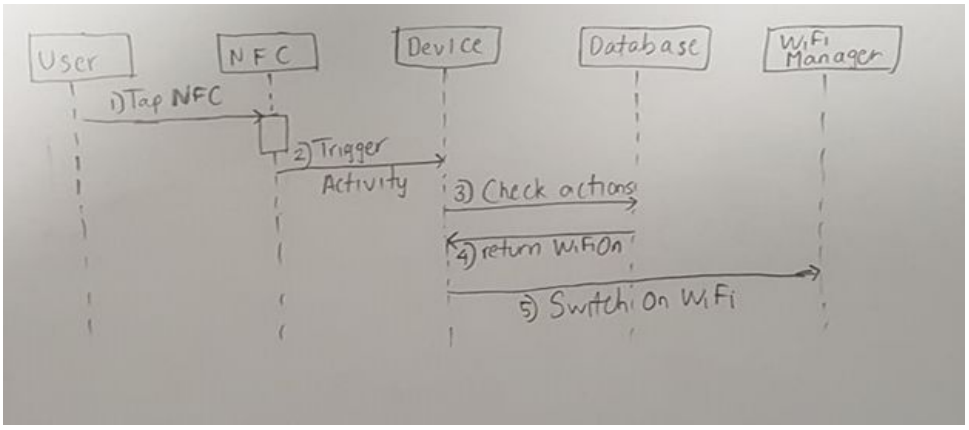

Figure 7: Sequence diagram to demonstrate system behavior if the user taps on an NFC tag that turns on WiFi.

The strategy pattern is well known for its ability to maximize cohesion and minimize coupling between the concrete strategy classes and the client object which is linked to the class diagram via the context object. This is because the client is only coupled to an abstraction or an interface and not a specific realization of the abstraction (the algorithm). This ensures that there is a minimal dependency between the client(NFCTriggerActivity) and the concrete strategy classes. This is similar to “abstract coupling” which is a more generic form of referring to minimize coupling. Any changes in the NFCTriggerActivity class won’t impact the concrete strategy class and vice versa. So, in the future, let’s say that the code to switch on Wi-Fi has a different syntax than what is currently used. In that case, all we would need to change is the concrete strategy class(wifiToggling.java) and we would not even have to touch the NFCTriggerActivity class. The strategy pattern also achieves high cohesion because it promotes coding via an interface. This forces us to break large classes into smaller classes and subsystems. The strategy pattern is also in association with the open/close principle which states that our project will always be open for extension to new algorithms but closed for modification to the existing classes and interfaces. Since this project has the potential to implement additional functionality in the future, all we would need to do is add more and more tags to the Firebase settings object and create corresponding context objects and concrete strategy classes. The downside of doing this is that each new algorithm requires us to create many new classes, and this could exponentially increase the number of objects defined in our code. This will negatively impact our project’s performance(a critical NFP) and would increment the application’s memory usage. Another disadvantage would be that the clients would need prior knowledge of what all algorithms or functionalities are being currently supported. The client must know whether we have the concrete strategy class to turn on Bluetooth or not.

One way our system could evolve in the future is that there could be a change in the number of parameters required to turn on Wi-Fi. This could be to avoid a security-related pitfall or a change in Android’s general guidelines. In any case, our system should be able to handle situations fairly well, thanks to the strategy design pattern. This is because the only part we would need to modify is the concrete strategy class associated with this algorithm and nothing else.

Command Design Pattern

The command pattern is a behavioral design pattern, which is related to the interaction and responsibility of objects. In the behavior design pattern, the interaction between objects should be in such a way that they can easily talk to each other and even then, they should be loosely coupled. In the Command design pattern, an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the purpose that owns the method and values for the method parameters. It encapsulates a request under an object as a command and passes it to invoker object. The Invoker object looks for an appropriate purpose which can handle this command and gives the command to the corresponding object. Next, that object executes the given command through a process known as an action or a transaction.

There are four terms that are always associated with the command pattern. They are command, receiver, invoker and client. A command object knows about the receiver and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The receiver object to execute these methods is also stored in the command object. The receiver then does the work when the execute() method in the command is called. An invoker object knows how to execute an army, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command; it knows only about the command interface. Invoker object(s), command objects and receiver objects are held by a client object. The client decides which receiver objects it assigns to the command objects, and which commands it attaches to the invoker. The client chooses which commands to execute at which point. To execute a command, it passes the command object to the invoker object.

The reason why we decided to select the Command design pattern is that the UI component of TAPP itself is a command pattern. The button on the UI is the invoker, the onClick method is the command interface, and the Concrete Command implementation varies with the kinds of requests we send to the receiver. The receiver in the application is the Google Android package. It receives our commands as a receiver. Figure 8 is the UML class diagram of the command design pattern, which shows what it is mentioned above. In this diagram, the Invoker class doesn't implement a request directly. Instead, the invoker refers to the command interface to perform a request (command.Execute()), which makes the invoker independent of how the application is performed. The Command class implements the Command interface by performing an action on a receiver (receiver. action1()).

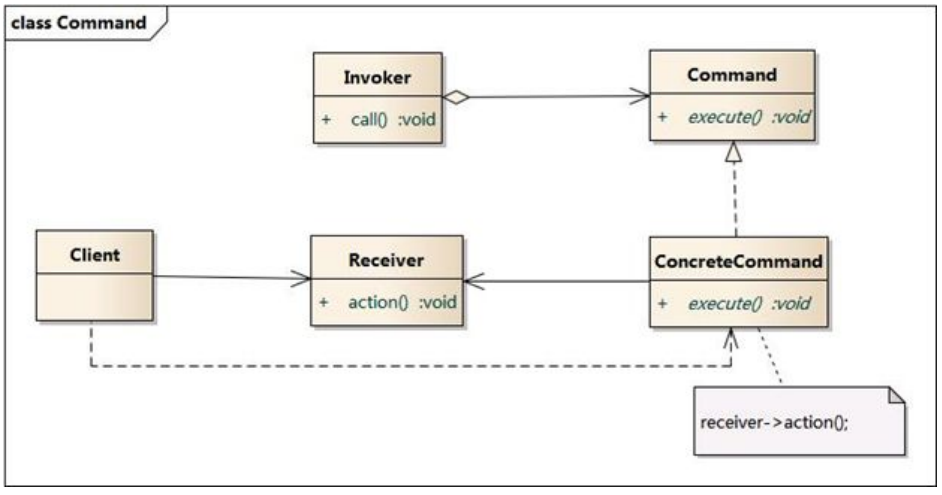

Figure 8: UML Structure for the Command Pattern

The Sequence of the Command design Pattern shows the run-time interactions: The Invoker object calls execute on a command object. The command calls an action on a Receiver object, which then performs the request. We take our application TAPP as an example to illustrate this. In TAPP, the procedure of accessing the Google Android package is an example of the command pattern. The user uses the application to click a button on the TAPP UI. After this, the event listener of the specific button will decide which specific command should be operated based on the user (client) 's behavior. After this, the Google Android package receives requests, and therefore, in this manner, they can support the request to do whichever operation the user wants.

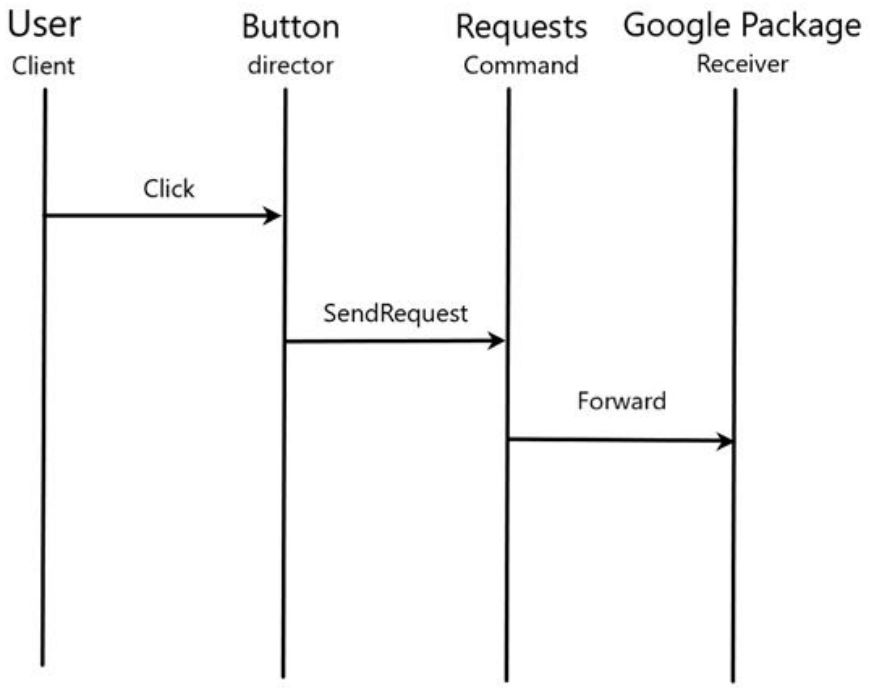

Figure 9: Sequence diagram displaying what happens when a user clicks on a button on the screen.

Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or patterns. Moreover, in this project, it reduces coupling between the invoker and receiver of a command.

Compared to other behavior design patterns, the command pattern increases the number of classes for each command. If the number of classes are too big, the project will be difficult to manage and it may reduce the application performance and speed.

Mapping Team Members to Architectural components/Design level classes.

Anurag Joshi was mostly responsible for coming up with a partial list of supported actions or tasks that can be performed when the user taps their phone against an NFC tag. He was also responsible for coming up with a database schema to store these actions and tasks under an NFC ID and to link each NFC ID with a user. This had to be modified after the pivots were released to significantly ease coding for the same. He was also responsible for making a link between the Firebase database and the application so that the code knows where to search once it senses that an NFC has been tapped. Lastly, he also implemented the strategy pattern, as outlined in this report to pick up an algorithm from a set of algorithms at runtime. The classes modified for the same are NFCTriggerActivity, ContextObject, Strategy Interface, some Concrete Strategy classes and MainActivity. These architecture components are TAPP Processor, Centralized Controller Logic, Tasks Database and Performing corresponding tasks. He is working on the login functionality through Firebase Authentication at the time of writing this document.

Xinjue Lu was responsible for the UI design and UI Implementation. The design procedure for me is straightforward as follows. Firstly, Xinjue did background for the NFC usage, how to make full use of the NFC functions. After obtaining the background, he did a contextual inquiry, asked several android users, what is the highest using tasks they do when using mobile phone. The following process is to draw the interaction Map, he listed down the structure and detailed functions of the application with the help of his teammates. After that, Xinjue started to draw the wireframe by using pencil project to determine the services, UI and interaction. The last step is to draw the High-Fidelity Prototype on Adobe XD based on the wireframe. To implement the project, Xinjue decided to make several icons at the beginning and then generated several pages such as activity_main.xml for the log in and activity_nfc_list.xml for presenting the UI shows on front-end. Moreover, he confirmed that the command pattern, as mentioned previously is been followed as each button as an invoker, request as a command.

Daniel Weisberg was responsible for creating the NFC reader(Read sensed NFC tag) and writer(Generate NFC ID component) as well as the QR Code reader and generator. He created the intent filters for both the reading and writing to the NFC. When the device was put next to an NFC, he triggered the reading activity to parse and record the NFC ID (class: NFCTriggerActivity). For the writer, he created a custom activity that wrote to the NFC whenever the phone was tapped (class: NFCWriteActivity). For the QR generator part from the pivot, he decided to make a simple activity that would simply use a library to generate a QR bitmap from the provided string. This image would then be shown within the activity in order to be read (class: QRCodeGenerationActivity). For the QR reader, he made a decision to use a live camera feed as opposed to taking a single picture and looking for a QR in it in order to have a better user experience (class: QRCodeScannerActivity). The QR reader was able to quickly scan QR codes and record the value it reads into the firebase database.

Gunin Khanna was responsible for implementing some of the functionalities that are to be triggered when a user taps the device against a registered NFC Tag. To be consistent with the strategy design pattern, He implemented multiple Concrete Strategy classes to perform the respective tasks associated with each NFC ID and in turn modified/consumed NFCTriggerActivity, ContextObject, and Strategy Interface. He also implemented multiple activities (activity_nfc_list, activity_new_tag, etc.) to facilitate the user workflow, this included changes to the respective XML files for each view to make it responsive across devices with different aspect ratios. He also worked on integrating these UI elements with the core logic of the application through various OnClickListener events. He is currently working on implementing various UI elements to support the application and development of more functionalities that can be triggered.

Referred Links:

1- https://developer.android.com/reference/android/nfc/package-summary
2- https://developer.android.com/guide/topics/connectivity/nfc/nfc
3- https://firebase.google.com/docs/database/