ENGG*4450 - Large Scale Software Architecture

ASSIGNMENT 1 - REVERSE ENGINEERING

Instructor:
Dr. Petros Spachos

Teaching Assistant:

Group: 3

Pieter Jurgens Krige - ID: 1012072

Andrei Korcsak - ID:

Disha Singh Nath - ID: 0995702

Harshal Patel - ID: 1032961

Bilal Ayash - ID: 0988616

Neel Bhandari - ID: 1004853

Submission Date: Tuesday, October 6th, 2020

School of Engineering

University of Guelph

Contents

List of Figures

1 Introduction

UML, Unified Modelling Language, provides a clear visual representation of how the classes in a project are related to one another. Being able to see how the different components of the code are related gives a better idea of the code behaviour. In this report, the process of re-engineering an open-source signal messenger application was documented. This application is a cross-platform service for encrypted instant messaging and voice/video calling. The main objective of this process is to obtain useful information about the construction of the softare through UML diagrams of the source code.

2 Reverse Engineering Process

To get a deep understanding on how this application is built, divide and conquer approach was implemented. This approach recursively breaks down the code and represents its main components such as class name, sub classes, interfaces, extensions, attributes, operations, etc. in a more convenient and readable format using UML diagrams. This helps in communicating the whole idea in just a graphical format that requires minuscule knowledge of Java programming.

For reverse engineering purpose and class hierarchy lookup, Astah UML modelling software was used. It was chosen by the group and recommended by the TA as it is a tool which has a user friendly interface and proper specifications required for the assignment. It allows users to use different dependencies and provides proper navigation through classes, their operations and attributes. Additionally, Eclipse UML software was also used as it is a simple tool which helps bridge the gap between source code and and it's UML diagram.Another significant platform used was Github to explain and understand the code. Most open source project uses Github as it is effortless and painless to keep track of the code, variables, import classes, connections with other classes, etc.

Finally, after finishing individual tasks, the group came along to understand the higher-level diagram for the signal app. In order to do this, each member navigated to the chosen classes and analyzed the import statements to recognize how one class is connected to other classes in the same folder as well as other folders.

3 Signalapp Architecture

3.1 High Level Overview

The signal-App higherlevel diagram presented in figure 1 consists of the following packages:

- Messages , Notification, SMS, Contacts, Crypto, and Profiles

Through analysis of source code and package diagram presented in figure 1, The contacts package seems to be one of the main components in this application. The contacts package connects to all packages analyzed as it has the most dependencies and compositions compared to other packages analyzed. The Contacts package was utilized by the notification, sms, profiles, and the crypto packages. The notification, profiles, and sms packages depends on the contacts package. Input stream from user gets encrypted and sent to the contacts class. The Crypto package is an aggregation to the contacts, and profiles package. The profiles package depends on the contacts inputted by the user. The Notification and SMS packages depends on the contacts package as classes from the contacts package are utilized in both packages.

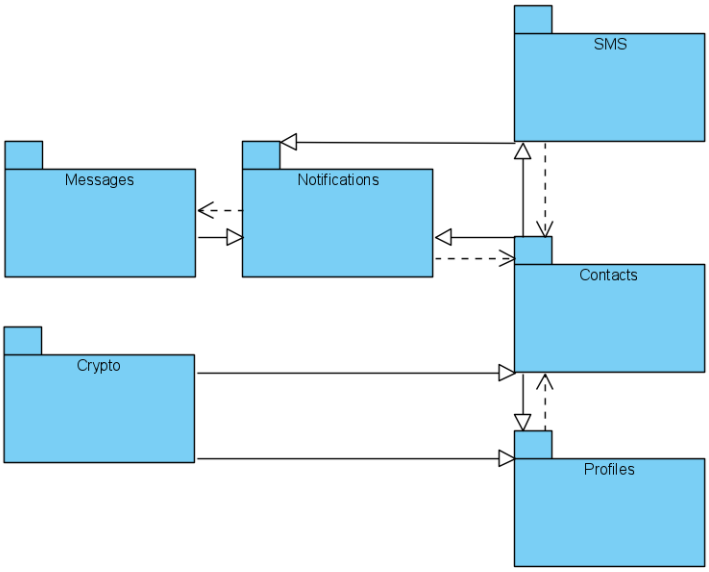

Figure 1: High Level Architecture

The figure above (Figure: 1) helps in understanding how classes connect and work together to provide the required output. In the sections below, packages functionality is highlighted and classes diagrams are presented.

3.2 Contacts

The Contacts folder of the Signal App consists of 31 Java class files, of which only five were modelled in UML Diagrams: ContactsAccessor.java, ContactChip.java, ContactsDatabase.java, SelectedContact.java, and SelectedContactSet.java. Astah UML was used to model these classes

The analysis was done by reading through the code provided in the Contacts folder and using the lecture slides and course resources to produce a diagram which encapsulated the information which I felt a person programming the class would need.

3.2.1 ContactAccessor.java

ContactAccessors' main purpose is to work with contact data which is on the phone already and parse it into something the app can work with. It can retrieve the contacts on the phone, parse the names and phone numbers, and add them into a database. It contains two inner classes, NumberData and ContactData which each implements Parcelable allowing them to be easily packaged for data transfer. ContactData creates parcels containing the ID, name, a list of type NumberData. The NumberData class parcels the phone number and type.

3.2.2 SelectedContact.java

SelectedContact is an object which consists of three attributes, a recipient id, a phone number, and a name. The methods in this class give the ability to: create instances of itself by giving a name or a number; return a recipient id or create one if not already present; and determine whether an existing contact matches another.

3.2.3 ContactChip.java

ContactChip extends the Android Chip class. It creates an element which allows information to be entered directly by the user, rather than parsing the data on the phone. It obtains the attributes need for a contact (id, name, phone number) from the user and creates an object of type SelectedContact. This methods also allows the setting an avatar image for the contact. It extends Chip, which is the standard method for getting user input in Android - our class, ContactChip, overwrites the Chip class's constructors.

3.2.4 SelectedContactSet.java

The SelectedContactSet class creates a list object of type SelectedContact. It provides limited but useful functionality like adding, clearing, and removing contacts from the list. The SelectedContactSet only allows for one instance of a SelectedContact object to be added into its list, therefore it has a multiplicity of one while the SelectedContact has a 0 to many multiplicity in this relationship.

3.2.5 ContactsDatabase.java

ContactDatabase is the class which creates a database of contacts which are able to be interacted with through the Signal app. The methods in this class allow for interaction with the database and its contents. This database does not contain just the basic user data, like in SelectedContact, but has columns for other attributes which help

with higher level app usability. The methods use the contactID (unique to every contact) in order to know where the action should be performed. This class also contains two sub-classes: SystemContactInfo and SignalContact. The SystemContactInfo is which is used to obtain the baseline contact information needed (ie. number, name, id) and then uses its methods to create a SignalContact which is then pushed into the database. The reason a SignalContact created is to store the extra attributes such as voice call support and display name.

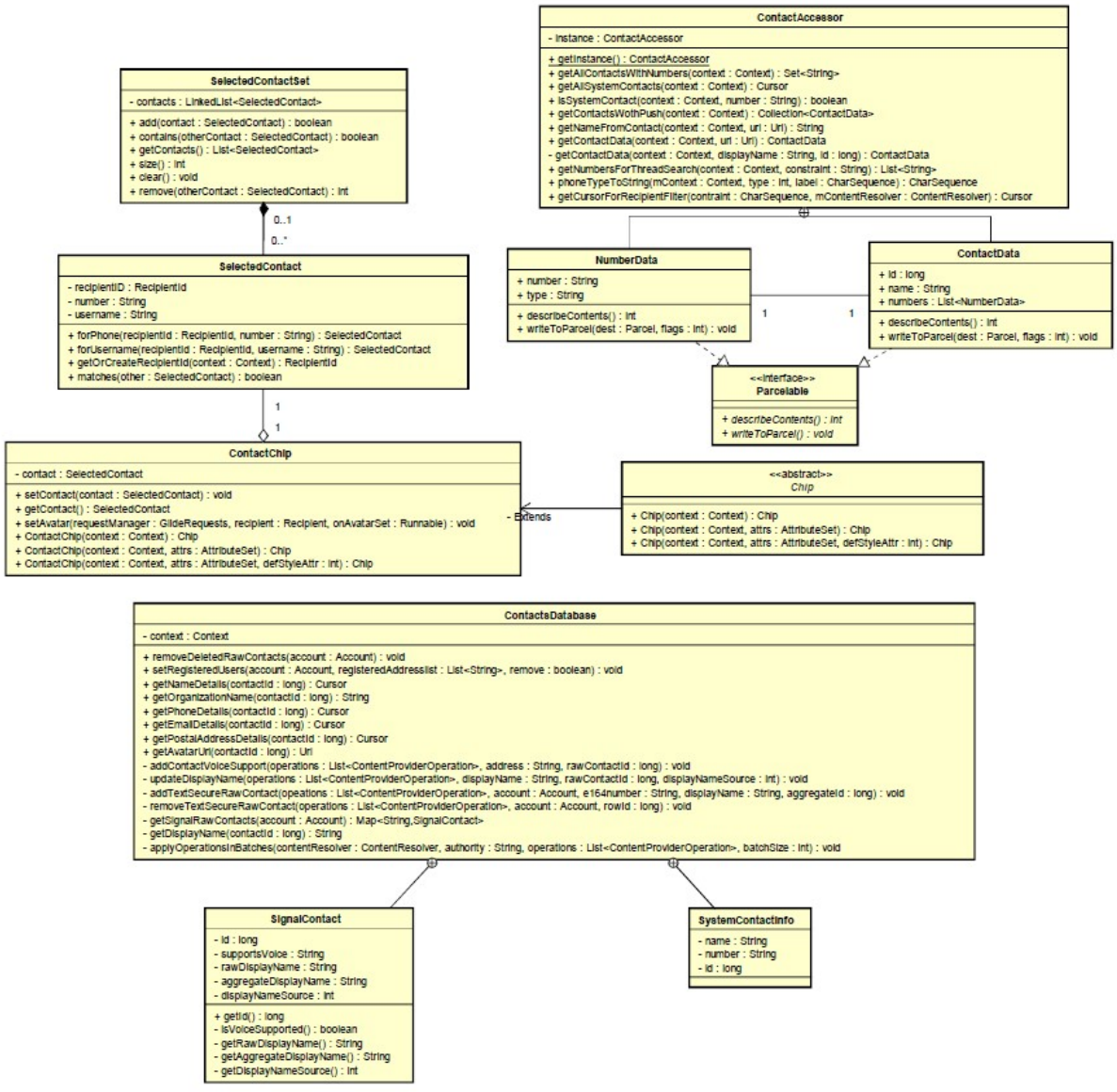

Figure 2: ContactAccessor.java

3.3 Profiles

The Profiles folder of the Signal App consists of 6 class files. The five files modelled in the UML diagram are; AvatarHelper.java, GroupShareProfileView.java, ProfileMediaConstraints.java, ProfileName.java and SystemProfileUtil.java.

3.3.1 AvatarHelper.java

The main purpose of the AvatarHelper is to aid the Avatar class. It is able to get an iterable set of avatars for a directory, delete an avatar, get the stream for an avatar, set the stream as an avatar, gets an output stream and many more. AvatarHelper has a composite relationship with Avatar, this object has 3 attributes; inputStream, filename and length. A composite relationship exists between Avatar and AvatarHelper as without Avatar there is no need for an AvatarHelper. AvatarHelper also has an aggregation relationship with AttachmentSecrets as AvatarHelper has an AttachmentSecrets object in the method getAvatar.

3.3.2 GroupShareProfileView.java

The GroupShareProfileView extends the FrameLayout, this class is used to initialize the user interface when in a group. The user interface will display the profile name and photo to all the members of the group. It also has an aggregation relationship with View as well as Recipient since GroupShareProfileView has both View and Recipient as it attributes.

3.3.3 ProfileMediaConstraints.java

The ProfileMediaConstraints extends MediaConstraints, the purpose of this class is to get media data such as; the max image width and height, max gif size, max video size, max audio size and max document size.

3.3.4 ProfileName.java

ProfileName implements Parcelable, the main purpose of the class is to interact with the profile name. This class has methods to format the profile first and last name into a full name , get the first and last name from the profile full name as well as many other helper (setter and getter) functions .This class also has an object called creator, this object overrides the ProfileName constructor and is able to create a ProfileName object with the input of a Parcel. Creator also has a method that is used to generate an array of ProfileName's.

3.3.5 SystemsProfileUtil.java

SystemsProfileUtil is composed mainly of 2 classes; getSystemProfileAvatar and getSystemProfileName. The getSystemProfileAvatar method has 2 parameters, one context type and the other a MediaConstraints type. The main purpose of getSystemProfileAvatar is to get the Avatar that the cursor is pointing at, this is done through the use of BitmapUtil. The getSystemProfileName method has one parameter and it is of type context. The main purpose of this method is to get the profile name of the avatar that the cursor is pointing to. The main purpose of the SystemsProfileUtil is to get the Avatar and the associated name that the cursor is pointing to.

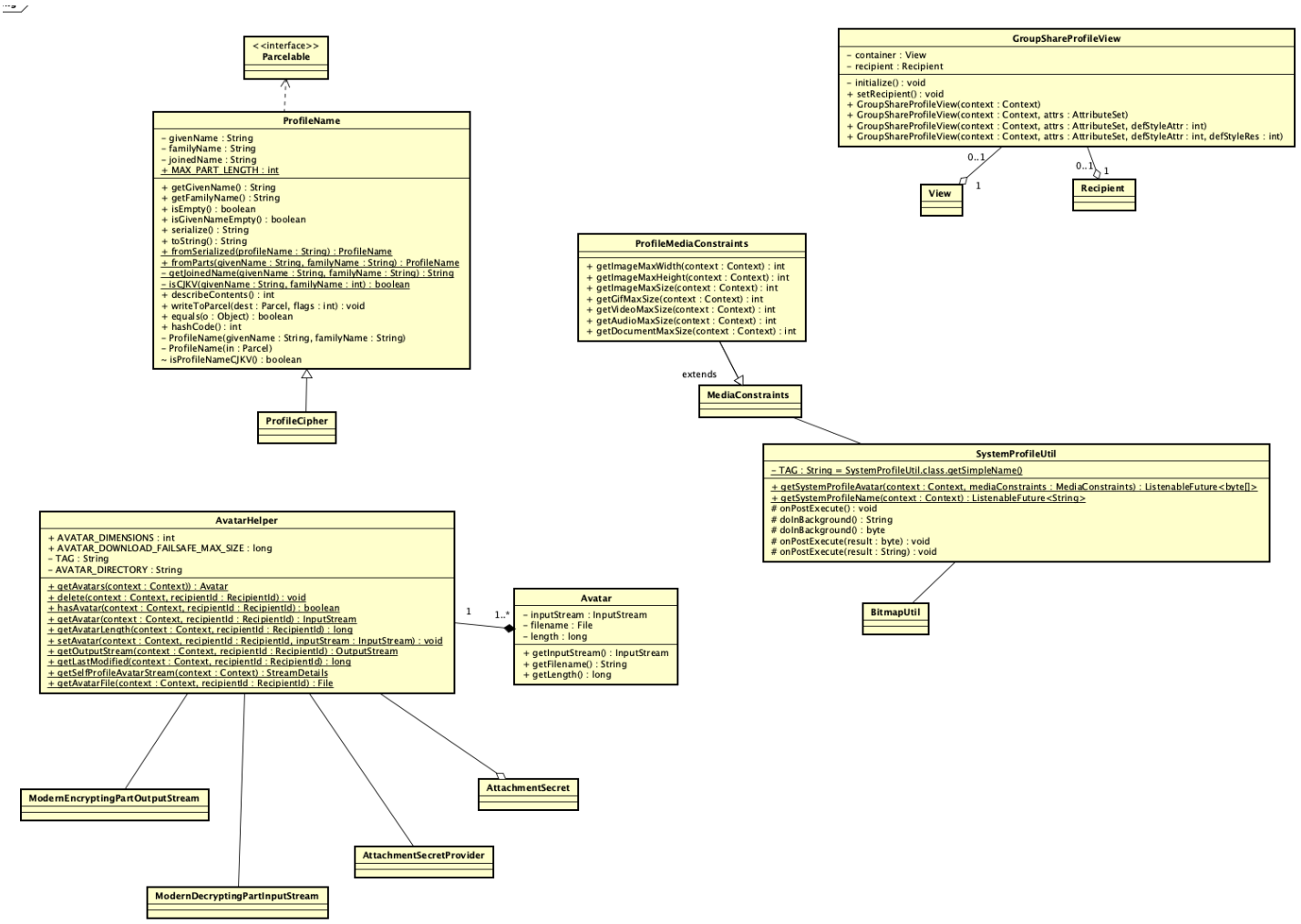

Figure 3: Profile UML

3.4 Messages

The messages folder consists of 6 different java files. These files are BackgroundMessageRetriever.java, IncomingMessageRetriever.java, IncomingMessageProcessor.java, MessageRetrievalStrategy.java, RestStrategy.java, and WebsocketStrategy.java. All the classes from these files have been modelled.

The UML diagrams in this section were created using a reverse engineering tool to automatically generate the java code from the git repository into UML diagrams. The tool used was a plugin for Eclipse called ObjectAid. The diagrams were then manually designed using Astah by using the generated model as a guide. Details that were missing from the auto-generated UML model were added after analysing the code in every file.

3.4.1 BackgroundMessageRetriever.java

The BackgroundMessageRetriever.java file contains a class used for retrieving messages provided by the MessageRetrievalStrategy class while the app is running. The class is composed of methods which determines if the message retrieval fails or passes. If fails, messages should get rescheduled. It also detects whether there is a need of executing a message fetch or if the websocket will take care of it.

3.4.2 IncomingMessageRetriever.java

This file mostly contains classes with functions used for overriding. The IncomingMessageObserver class is connected directly with the MessageRetrievalThread. The MessageRetrievalThread class extends to the Thread class and implements the Thread.UncaughtExceptionHandler. The ForegroundService class extends as well to a class called Service but it does not have any relationship with the other classes.

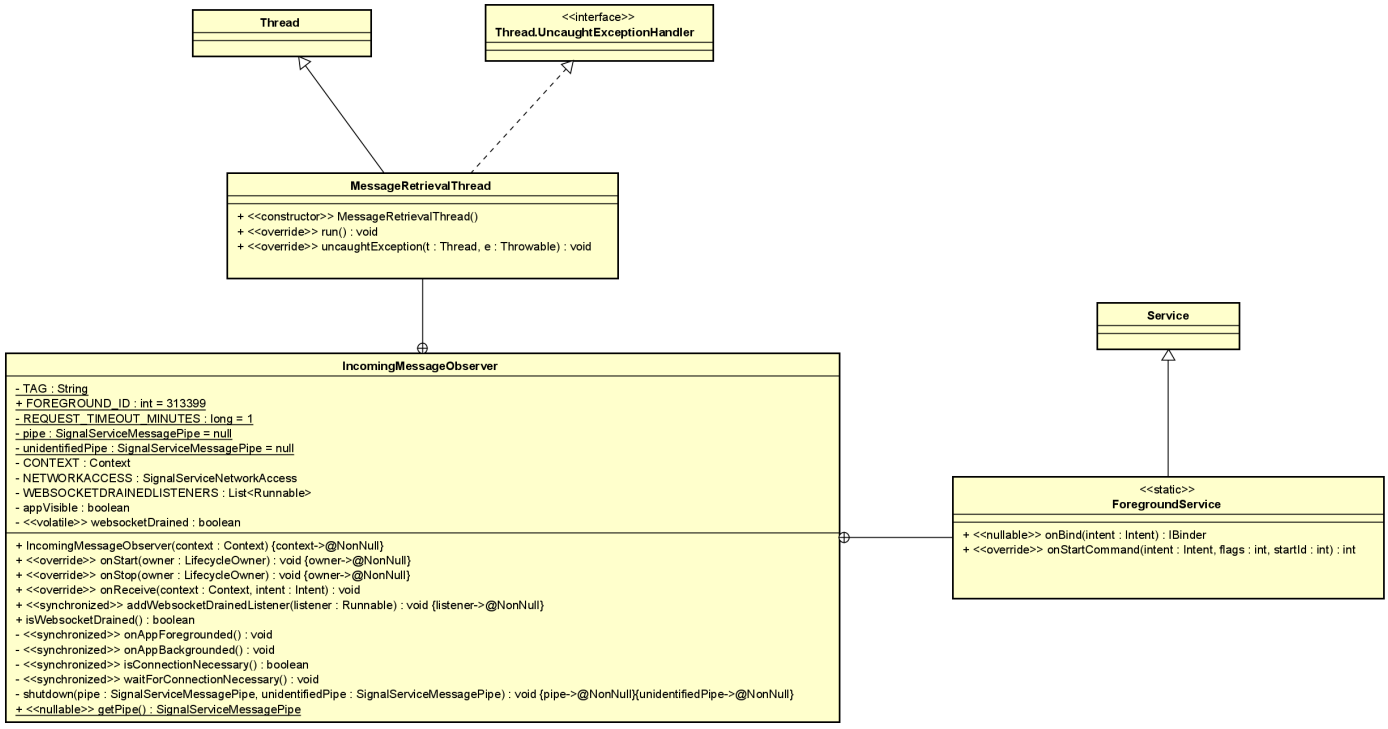

Figure 4: Messages UML Diagram Part 1

3.4.3 IncomingMessageProcessor.java

There are two main classes in this file, where the IncomingMessageProcessor contains an inner class called Processor that implements the Closeable class. These two classes depend on each other where IncomingMessageProcessor cannot exist without a Processor class to provide the incoming messages. An instance of a Processor is required to allow the process messages in a thread the safe way. The IncomingMessageProcessor is the entry point of all envelopes that have been retrieved. Envelopes must be processed here to guarantee proper ordering.

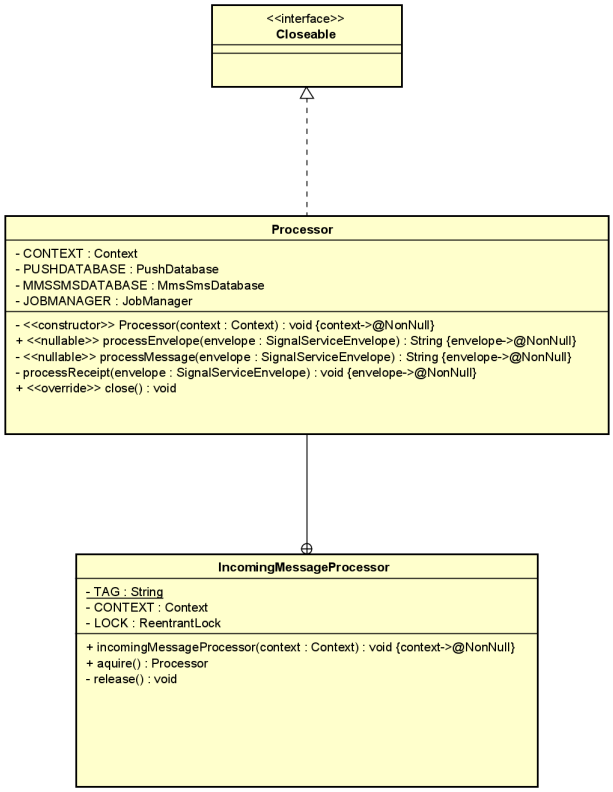

Figure 5: Messages UML Diagram Part 2

3.4.4 MessageRetrievalStrategy.java

MessageRetrievalStrategy is the most important class within the messages folder since it builds upon many other classes. Both the WebsocketStrategy and RestStrategy classes are extending from MessageRetrievalStrategy and it also contains an inner class called QueueFindingJobListener. The class is responsible for fetching and processing batches of messages. It contains functions for fetching and processing any pending messages. Messages are blocked until stored and processed, not just retrieved.

3.4.5 RestStrategy.java

This file only contains the RestStrategy class which extends on MessageRetrievalStrategy. It is used for retrieving messages over the REST endpoint. It creates a new instance of QueueFindingJobListener which means that the two classes communicate to each other.

3.4.6 WebsocketStrategy.java

The WebsocketStrategy class extends from MessageRetrievalStrategy. It contains an override function called execute(), a constructor, and an exception handling function. An instance of this class is being called by the QueueFindingJobListener as well which implies that there is a relationship between the two classes.

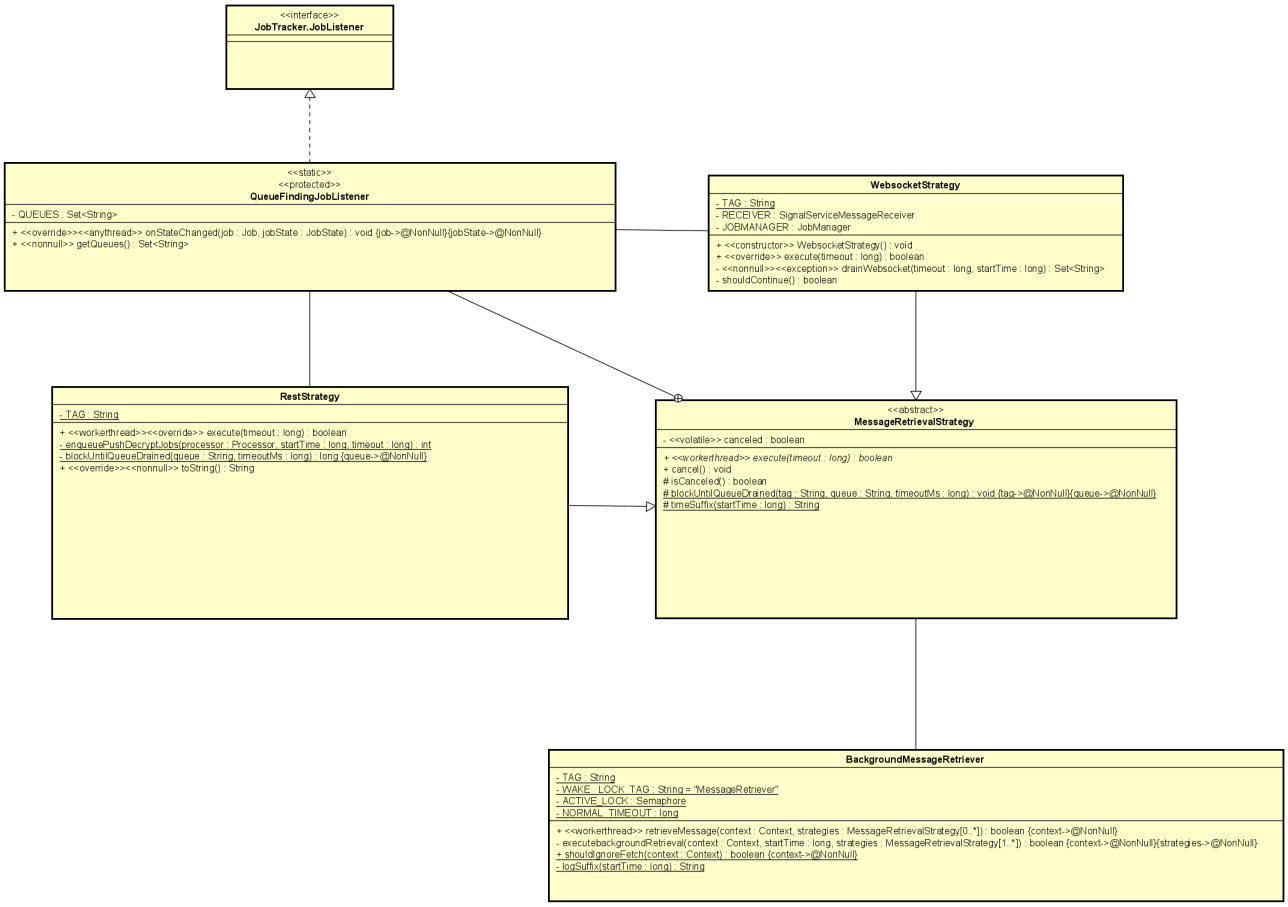

Figure 6: Messages UML Diagram Part 3

3.5 SMS

The basic purpose of the sms folder is to handle incoming and outgoing messages. In addition to that, it also encrypts, stores identity, verifies identity of the incoming and outgoing messages. Finally, it can handle group messages, media messages and can perform same operations such as encrypting, storing and verifying incoming identity, outgoing identity, etc.

3.5.1 MessageSender.java

MessageSender class is the building block of sms folder as it connects all the other classes in the same folder.It imports various classes from the standard 'android', 'androidx' package and from internal 'org.thoughtcrime.securesms' folders. This class does the various operations. It controls outgoing text messages, media messages and group messages and uses simple method called 'send' to send the outgoing text message. It also uses try-catch block to send any media messages and re-sends it if there is an exception. In-fact most methods try-catch block to test and execute if an error occurs in try block.

MessageSender class has a nested class called 'PreUploadResult' which implements interface 'Parcelable' that contains the abstract methods. This class holds some final variables, methods to assign and return those final variables in unModifiableList format (@NonNull) and few override methods (@Override) to override the methods in the super-class. Finally, it has enum method type which has a constant called 'INSTANCE' whose sole purpose is to instigate undertaking of the events that follows. One interesting thing to note is that all the methods in message sender are of static type. This methods can exist independently of any instances created for the class.

3.5.2 IncomingTextMessage.java

IncomingTextMessage class imports similar but fewer classes as messageSender. It implements 'Parcelable' to access the interface methods and has various public methods to set and get the sender's as well as receiver's information such as identity, device ID, protocols, subscription IDs, etc. It has a final variable 'GroupID' which is Nullable. This could help in detecting method calls that can return null and also variables that can be null.
The IncomingTextMessage class has a few Boolean methods to check some basic information i.e. is there any reply path, is message secure, is identity updated, is identity verified, is it unidentified, is it a group message, etc. At the end, it has an override method which writes all the information to 'Parcel' which is an interface . It does not explicitly connects to messageSender class but it connects through the interface Parcelable.

3.5.3 IncomingIdentityUpdateMessage.java

IncomingIdentityUpdateMessage class extends IncomingTextMessage. It has 2 methods to update the incoming identity and check if it is updated. First method only uses 'super' keyword to call its superclass (IncomingTextMessage) methods. This would eradicate the confusion between superclasses and subclasses that have methods with the same name. Second method is of 'boolean' type to return true if the identity is updated. This method is overridden so that this class can implement IncomingTextMessage (parent class) methods.

3.5.4 IncomingIdentityVerifiedMessage.java

IncomingIdentityVerifiedMessage extends IncomingTextMessage. It has exactly 2 methods same as the class IncomingIdentityUpdateMessage but this time verifies the identity instead of updating it. The first method would use 'super' keyword to call methods in parent class (IncomingTextMessage) to verify identity and the second method is of 'boolean' type which checks if the identity is verified and return true if it is. This method is also overridden.

3.5.5 IncomingIdentityDefaultMessage.java

IncomingIdentityDefaultMessage also extends IncomingTextMessage. It also has 2 methods similar to above classes named: IncomingIdentityDefaultMessage which uses 'super' keyword to connect to the IncomingTextMessage (Parent) class and overridden method isIdentityDefault of boolean to check if it is default or not.

Figure 7: SMS UML Diagram Part 1

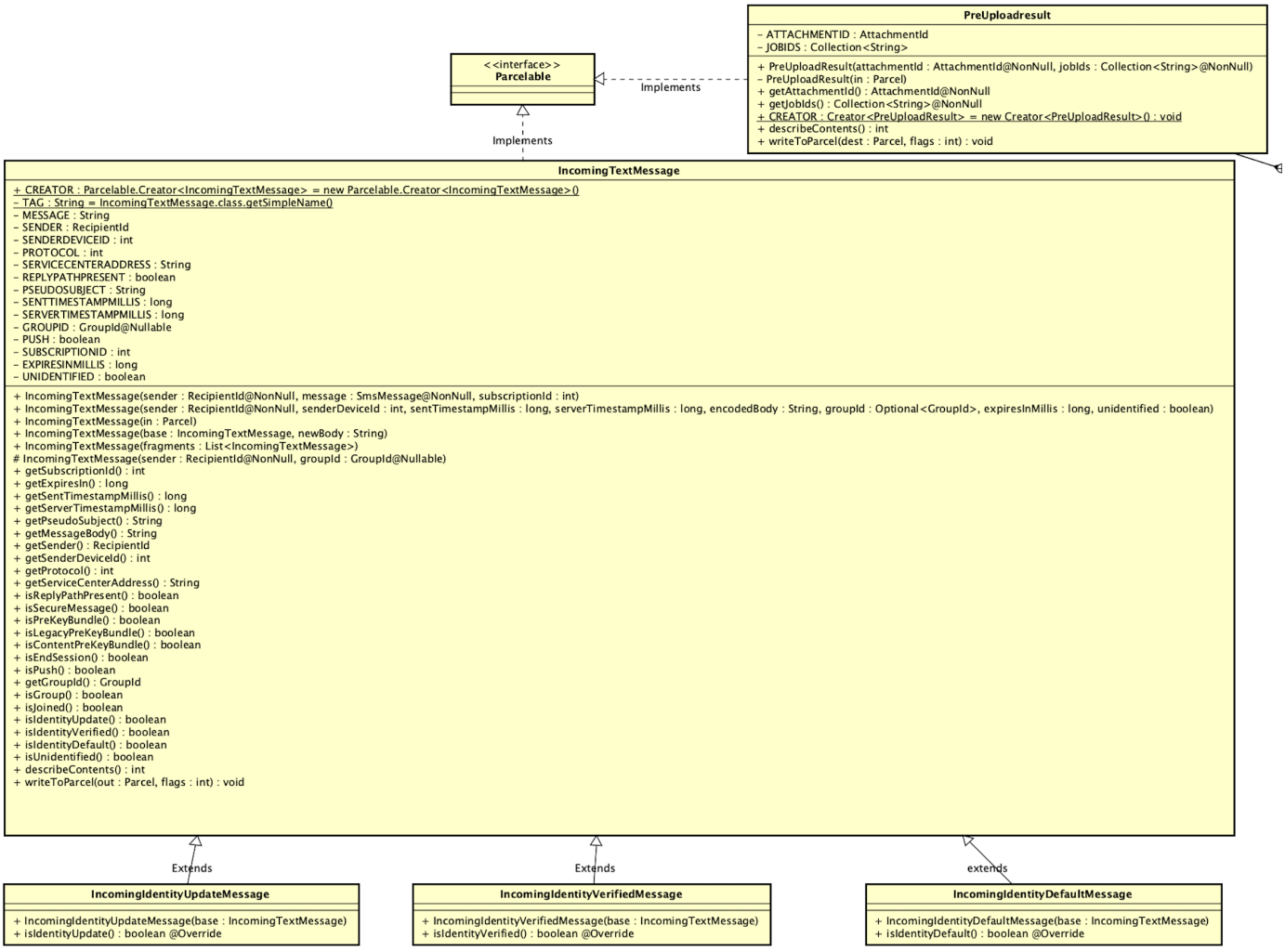

Figure 8: SMS UML Diagram Part 2

3.6 Crypto Application

To reverse engineer the crypto folder, different tools such as eclipse, Visual Studio Code, and Visual Paradigm were used. All code in the crypto file was analyzed to help understand how the crypto package work. Not all folders where documented to keep irrelevant information minimized. This package is responsible for the security aspect of the application. From Analysis of files, this application uses different encryption algorithms. This includes AES, SHA256, and SHA1 (256 Bit Encryption). These algorithms are trusted by developers to ensure security is not breached. This package would be a difficult package to implement changes to as the Encryption and decryption process already uses advanced algorithms.

3.6.1 SignalProtocolStoreImpl.java

This class is part of the storage package in the crypto package. This class is important as it stores encryption and decryption keys to read data in storage. Through passing context into this class a new preKey, Signed prekey, identity key, and Session store variables are created and utilized throughout this class. It mostly overrides functions used in the storage package.

3.6.2 MasterCipher.java

This class gets utilized in the AsymmetricMasterCipher class, identityKeyUtil class, and the MasterSecretUtil class. The main function of this class is to handle encryption for local storage. The protocol format includes a 16 byte random IV, AES-CBC, and a HMAC-SHA1. This protocol uses a key to encrypt and decrypt messages in local storage. The public class AsymmetricMasterCipher utilizes this class when encrypting and decrypting bytes. In the class IdentityKeyUtil, the masterCipher class is used to identify key pairs. This includes the private key by using the decryptkey function in the class. This class is also utilized in the AsymetricMasterSecret class to create a new masterCipher key if masterSecret is not empty. This class is used to decrypt the djbPrivate bytes which is retrieved from context.

3.6.3 ProfileKeyUtil.java

This class provides the the profile key and status of the profile key to other classes. This class was used in the UnidentifiedAccessUtil class. The provided profile key is used to derive the access key which is later used to decrypt specific context.

3.6.4 AttachmentSecretProvider.java

Attachment Secret Provider class is a provider responsible for creating or retrieving the attachmentSecret model. AttachmentSecret class is associated with this class with a multiplicity of 0..1. The attachment secret class contains the cipher ,Mac, and modern key which is utilized in the sub classes ByteArraySerializer and ByteArrayDeserializer. These associations allow the application to serialize attachments and store in local storage.

3.6.5 AsymmetricMasterCipher.java

This class is responsible for encrypting local data. In the case where TextSecure receives an sms, but the user's local encryption passphrase is not cashed this class is used. This could be caused due to a timeout, or passphrase has not been inputted yet. If this case occurs, a public key of a local key-pair is accessed. Using symmetric encryption (When user inputs the passphrase), the private key of the local key-pair is accessed, message is decrypted, and replaced into the DB with symmetric encryption. In this class, an ephemeral key-pair is first generated, ECDH is performed with the public key of the local durable key-pair, KMD with the ECDH is then performed with the result to obtain a master secret. The master key is then used to encrypt the message.

3.6.6 KeyStoreHelper.java

This class is utilized in the Attachment Secret Provider class and the database secret provider class. This class provides functions that help in extracting or generating the keyStore and the secret key. This class also contains sub-classes that serialize and de-serialize context before and after storing in database and retrieving information from database.

3.6.7 ClassicDecryptingPartInputStream.java

This class is responsible for decrypting the input stream. if error occurs during this process, this class handles exceptions. This class is associated with Cipher input stream wrapper class

3.6.8 Tools Used

- Visual Studio Code: This was a personal choice. This IDE was used to read the code and search for terms to understand how classes connect in the package.
- Eclipse JAVA: This IDE was used to create the class diagrams. This IDE was used instead of ASTAH because project was build on Eclipse. This allowed to ensure data presented in the report is accurate.
- Visual Paradigm: This tool was used to validate the class diagram as it provides an automated class diagram of all files. This

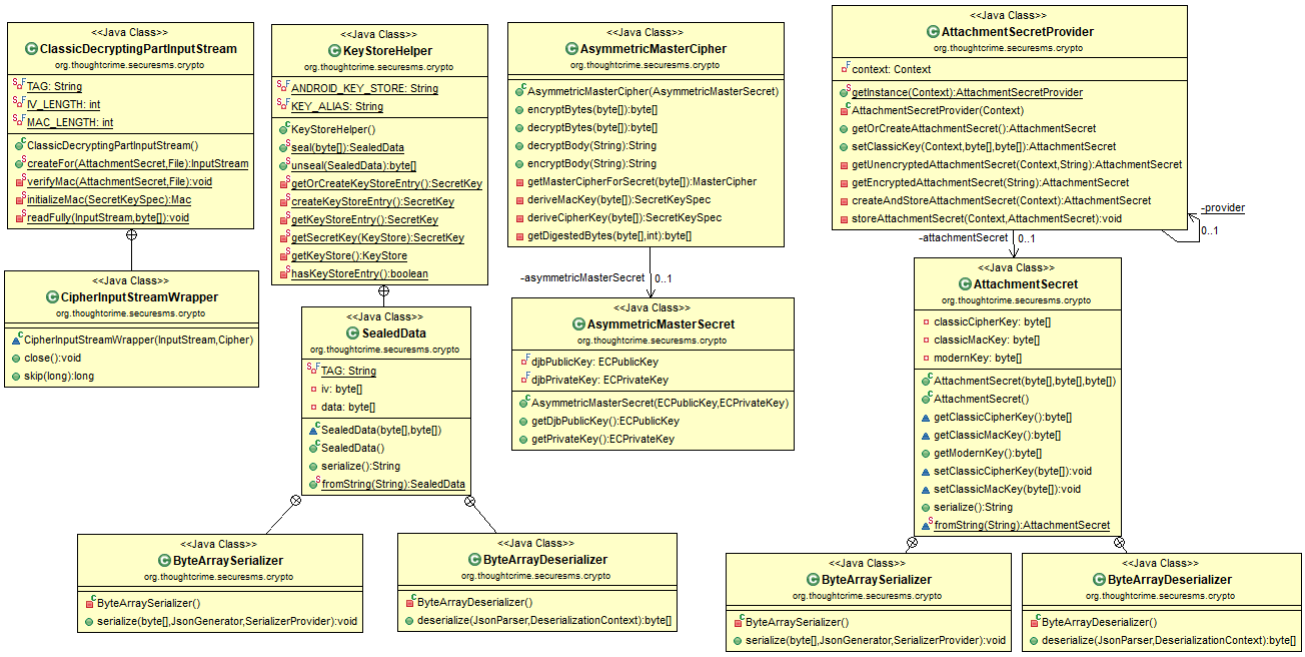

Figure 9: Crypto UML Diagram Part 1

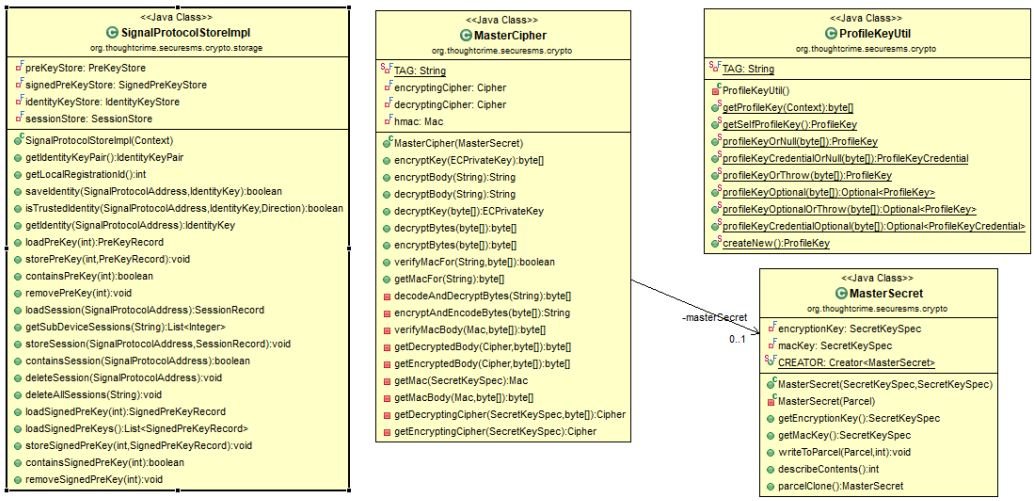

Figure 10: Crypto UML Diagram Part 2

3.7 Notifications

The notification package has 20 Java classes from which 5 are chosen for the UML diagram. It's basic functionality is to manage the notification being sent to the user by receiving information from other packages in the application as well as from classes inside the package

3.7.1 MessageNotifier.java

MessageNotifier is an interface which contains the class ReminderReciever that extends BroadcastReciever. It's main purpose is to create the blueprint of the methods that will be implemented by the classes using MessageNotifier as an interface.

3.7.2 DeafultMessageNotifier.java

DeafultMessageNotifier class handles the posting of system notifications for new messages. It implements the interface MessageNotifier and overrides most of its methods to introduce specific functions such as show the user the thread, timestamp, failed message delivery or update the notifications. It also has it's own methods that add more functionality, one such is isDisplayingSummaryNotification method that verifies if notification summary is being displayed by returning 'boolean' value true for proper implementation. This class has two inner classes, CancelableExecutor which is used by it's field and DelayedNotification that implements Runnable interface.

3.7.3 OptimizedMessageNotifier.java

OptimizedMessageNotifier is also a class that implements MessageNotifier. It contains two fields, wrapped and limiter to use a leaky-bucket strategy to limit notification updates. To accomplish this, it overrides the methods from the given interface.

3.7.4 MarkReadReceiver.java and DeleteNotificationReceiver.java

Similar to other Reciever classes, MarkReadReciever class and DeleteNotificationReciever class both extend BroadcastReciever. The objective of MarkReadReciever is to schedule the deletion of sms and mms notification as they have been read. It Uses class ExpirationInfo from database.java and schedule the deletion. The DeleteNotificationReciever class is responsible for actual deletion of the notifications.

Figure 11: Notification UML Diagram

4 Conclusion

While analyzing the Signal messaging application, UML diagrams have shown to be a great tool to visually present the complex relations and dependencies between packages as well as their classes. By using the recommended reverse engineering process, it was possible to break down the code into smaller sections and understand the individual functioning of classes and their interaction with each other on the deeper level of the code. This technique has helped strengthen the grasp on the overall structure of the application and as a result, in the future will enable individuals to catch bugs and faults in the design more efficiently.