ISAD157 – Facebook Database Coursework Report

Introduction:

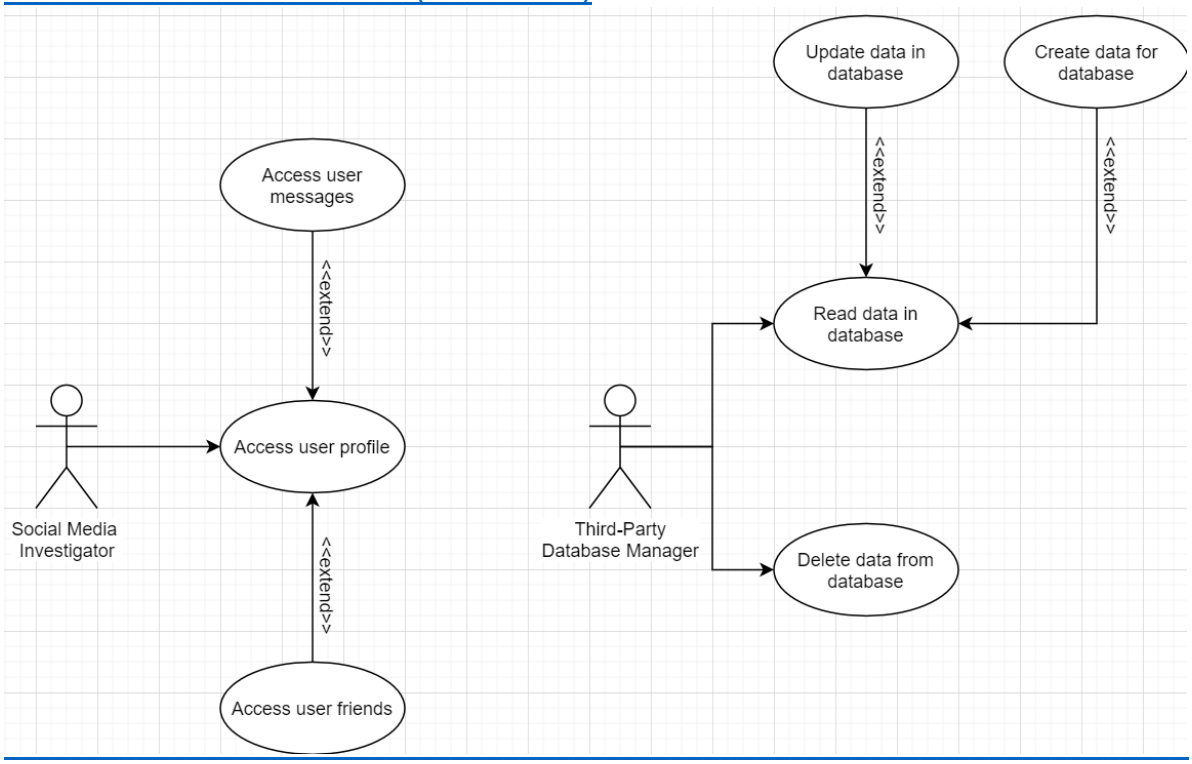
For this coursework project, it was decided that the database would be built for an investigator working for law enforcement. This investigator would have basic access to the database, being able to view any of the necessary information but not edit it in any way. The scenario in mind was that the investigator would have been given access to a particular sample in a third-party's database that had been identified within the entire database for potentially suspicious users and/or messages – finding this information was a part of the case to which the investigator had been assigned. The investigator was to analyse the users, their information, and their messages to find anything that would aid their investigation. The second (and only other) user for this database (sample) is the third-party's database manager themselves. This person would have higher-level access than the investigator; they would have the additional capability to create, read the information about, update the information about, and delete users. Though the manager's capabilities were not represented within the database itself, they were shown through the UML diagrams.

Evaluation:

UML Diagrams:

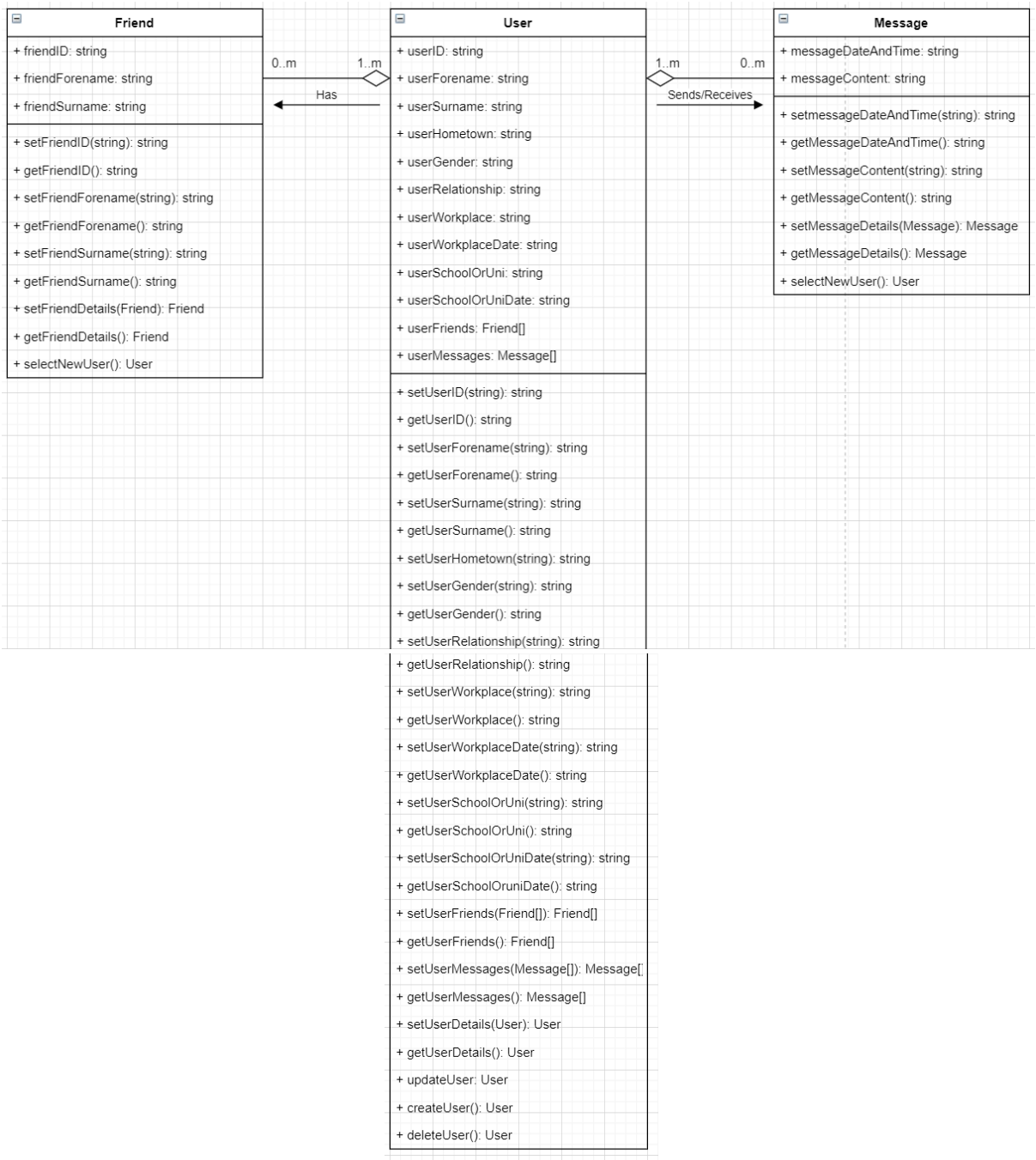
The first diagram to be created (and the first to be finished) was the Use Case Diagram. It is simple enough, containing two separate diagrams: one for each user (the 'Social Media Investigator' and 'Third-Party Database Manager', abbreviated to SMI and TPDM respectively for conciseness and simplicity). For the TPDM, only the create, read, update, and delete (CRUD) functions were included; the assumption made here was that the TPDM would be capable of the same use cases as the SMI, and that they needn't have been repeated.

20.04.02 Facebook Use Cases (Coursework)



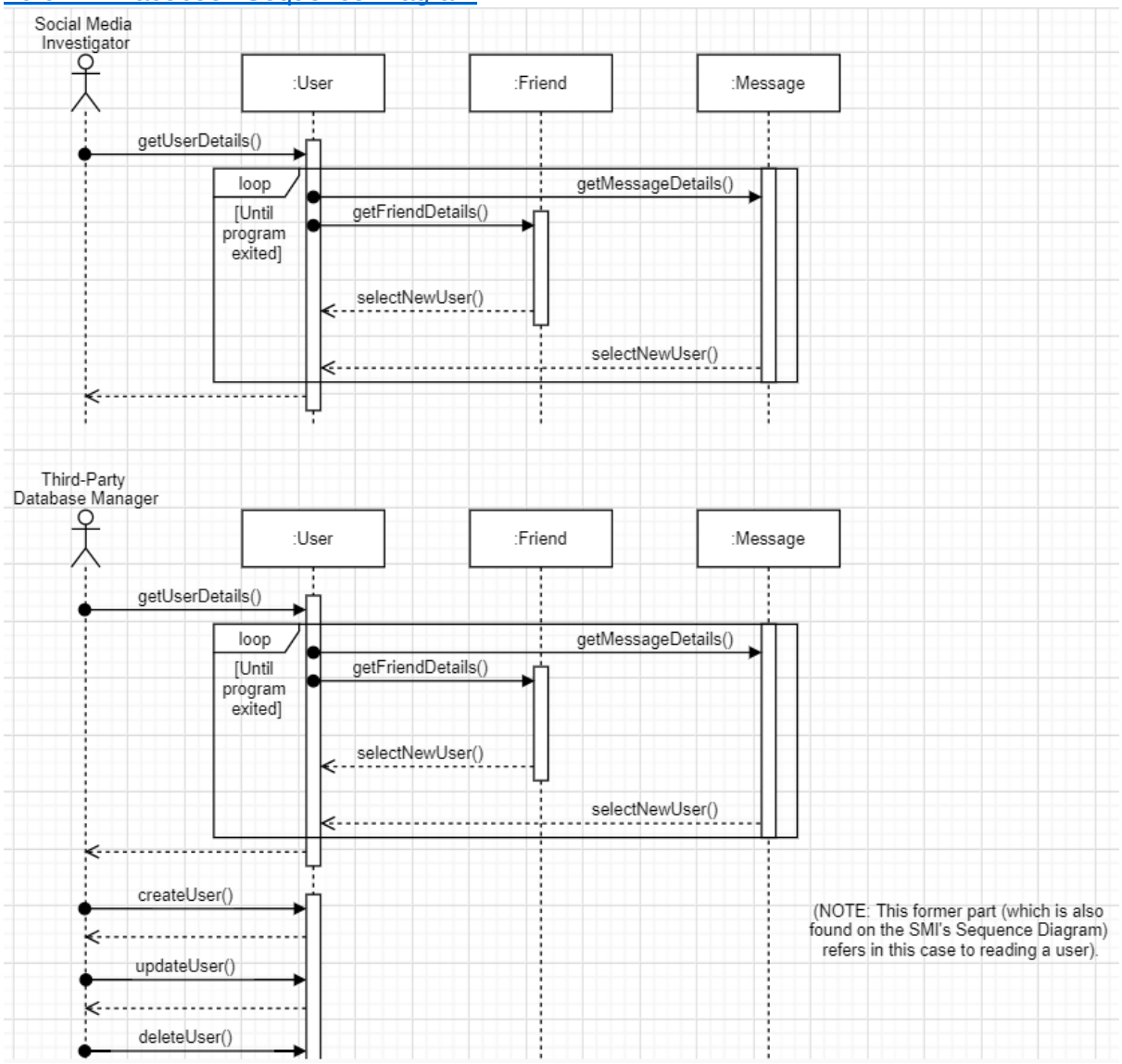
The Class Diagram was included in the number of UML diagrams created, but it should be noted that, with the functionality of the database as it is, it was decided that the classes were not a necessary inclusion in the C# code itself. The database works completely correctly without them. Regardless, a diagram to represent the three main classes (User, Message, and Friend), their variables, and their methods was created. A method named “selectNewUser()” was created for both Friend and Message such that viewing one of these could link back to the User. However, it must be admitted that no such functionality was put into the database because it couldn’t be figured out exactly how to do so; instead, the database simply displays all data with a data grid for each table.

20.04.05 Facebook Class Diagram (Coursework)



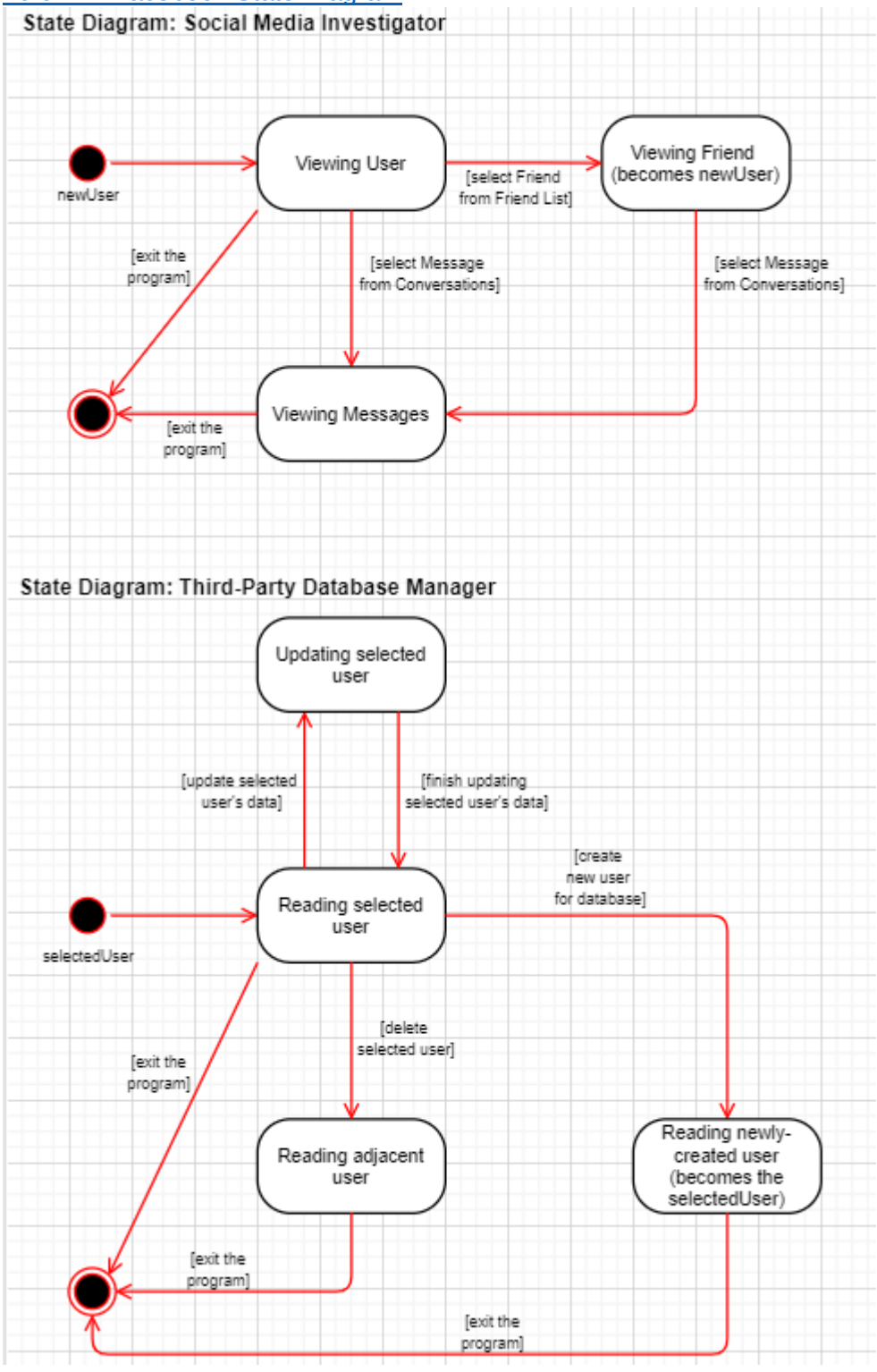
For the Sequence Diagram, two diagrams were created. The TPDM has an identical loop to the SMI, as the SMI's viewing of information is equivalent to the TPDM's reading function. The difference lies with the TPDM having three more methods, each of which return straight from a User object to the TPDM. These are the remaining CRUD functions.

[20.04.14 Facebook Sequence Diagram]



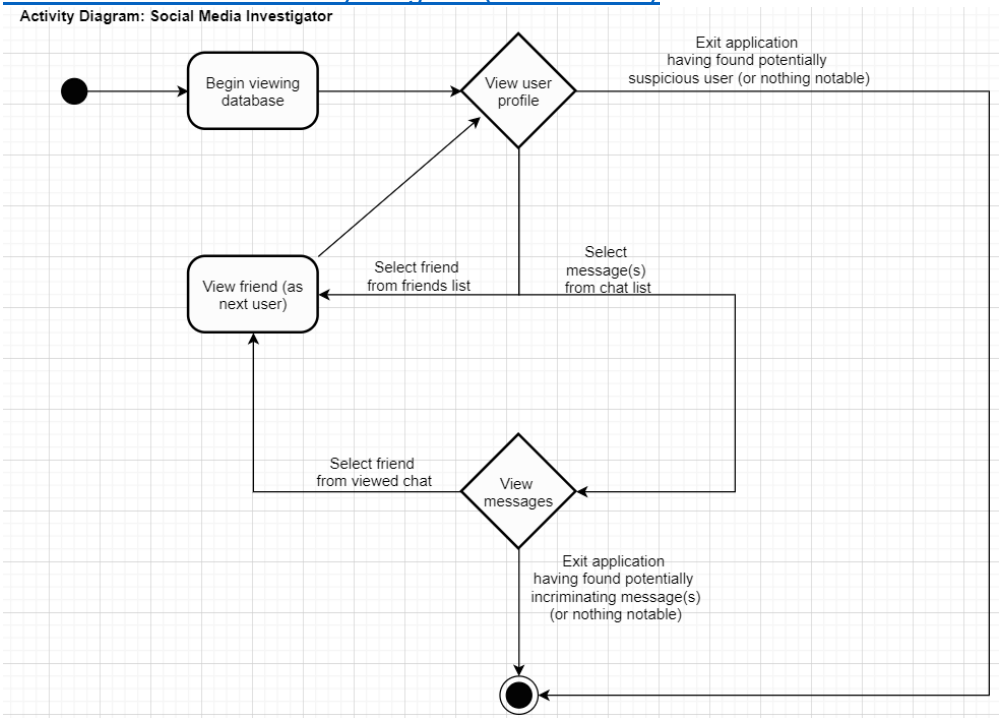
There is one State Diagram for each user. They are arguably rather visually simple and clear. For the SMI, three states (other than the entry and exit functions) exist and they are all states of viewing different pieces of information. The TPDM's states are also all reading information, but they are based upon actions in between that define what is being read. When a user is created, it is that new user being read, for example.

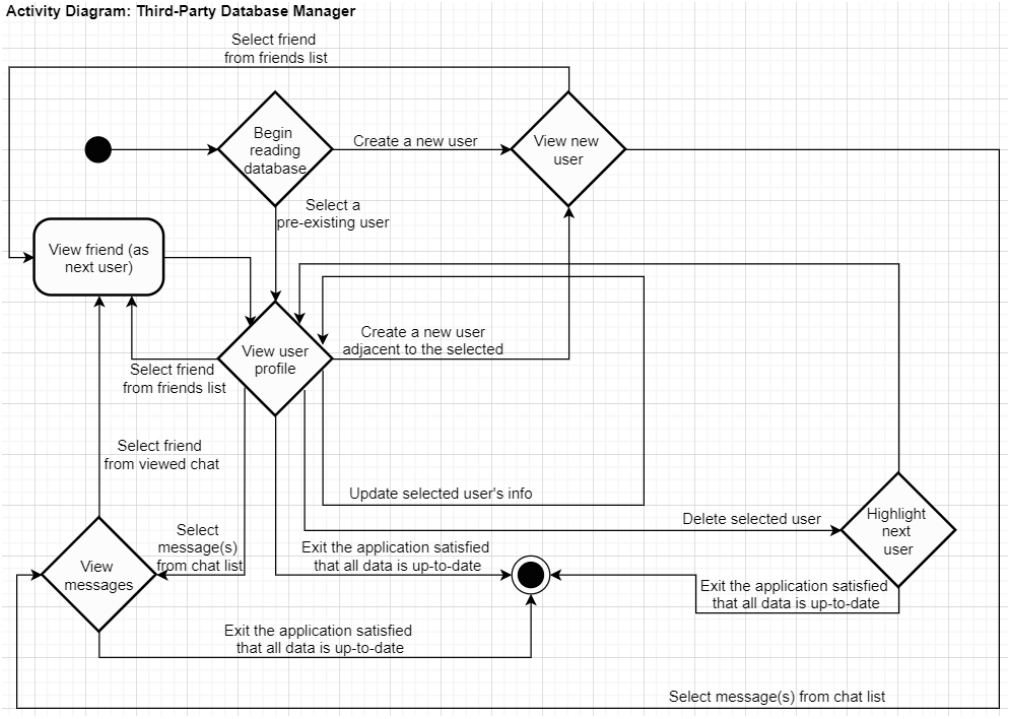
[20.04.14 Facebook State Diagram]



The Activity diagram is somewhat complex, though simple in presentation enough to understand sufficiently. The SMI's diagram is the clearer of the two, containing only four total activities (two of which are decisions). There exists a loop to represent friends being their own users by having the viewing of friends directly lead to viewing a user again. Though more complex, the TPDM's diagram is as simple as could be achieved. Largely, the decisions result in viewing a user profile, and the application can be exited from there. Again, unfortunately, displaying only specific users when interacted with was not able to be achieved in C#.

[20.04.15 Facebook Activity Diagram (Coursework)]

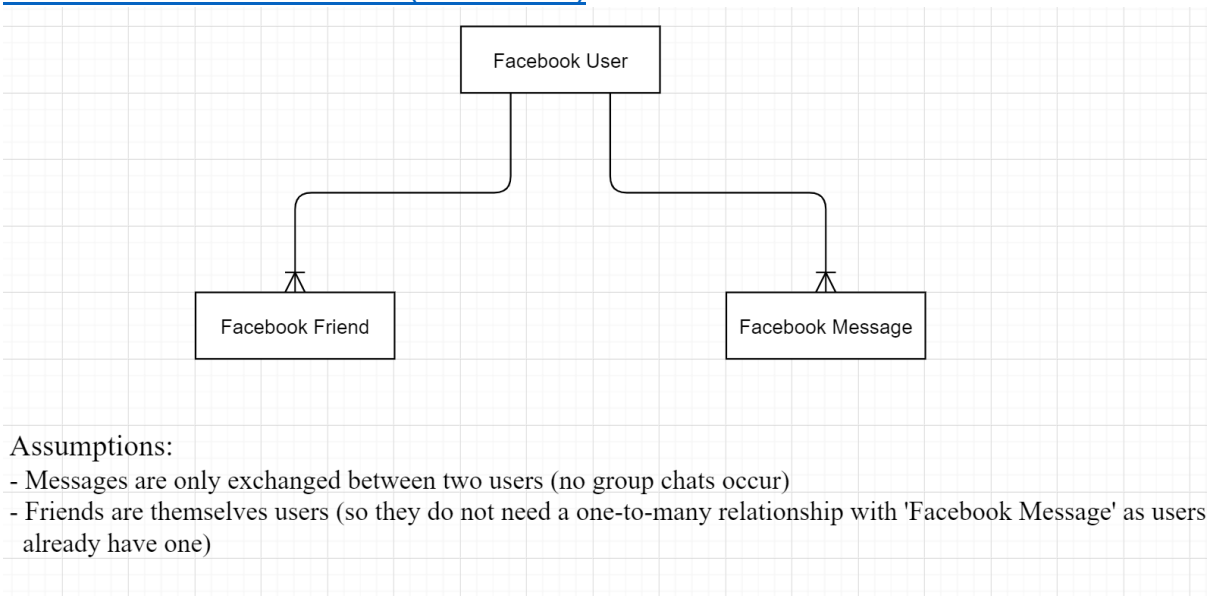




Entity Relations:

The initial Entity Relationship Diagram is likely what one would expect it to be. The only entities are User, Friend, and Message and there is a one-to-many relationship between the User and both Friend and Message. Friend and Message are not directly linked themselves due to an assumption (which is stated on the diagram) that it is not necessary on account of friends themselves being users. The other assumption made was that messages did not refer to group chats, and hence were only between two users at a time (preventing the potential for a many-to-one relationship between Message and User respectively).

20.04.02 Facebook Initial ERD (Coursework)



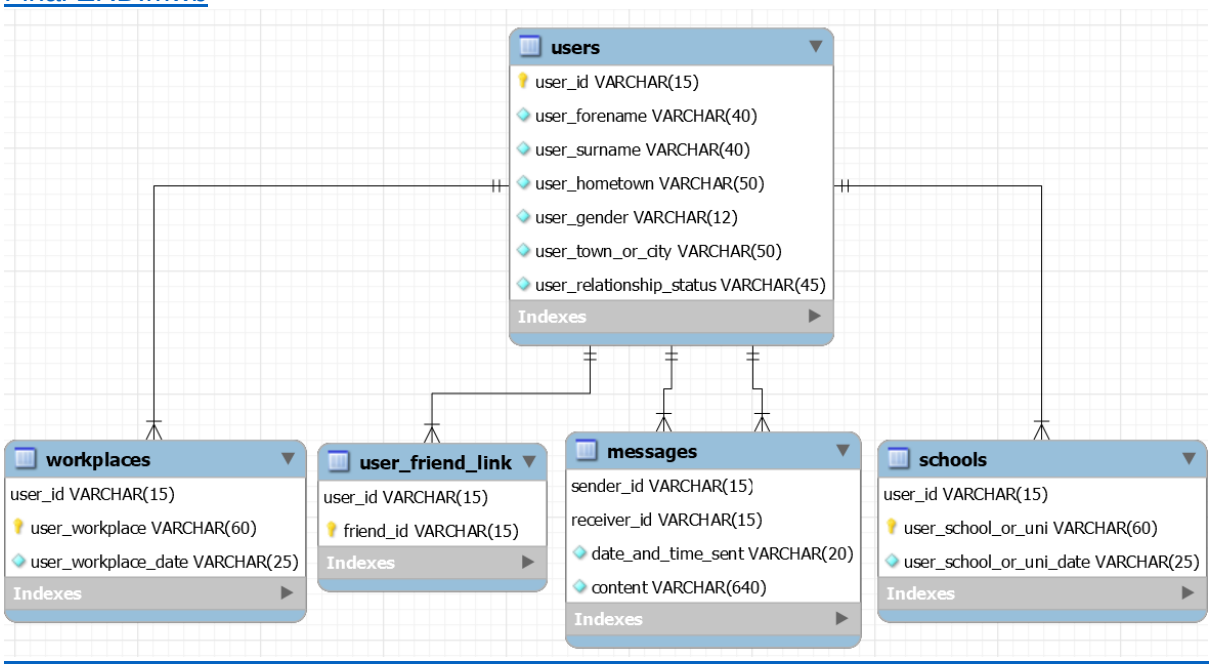
The process of Normalisation was completed as it standardly would be. It did, however, experience one change (which was later reverted). To avoid the repetition of data under separate headings, initially “Sender ID” and “Receiver ID” were considered to be classed under “User ID” and “Friend ID”. Whilst this worked for the Normalisation, it meant that there would be no relation in the database between users and friends other than via the messages table, which shouldn’t be the case. Reverting this change meant that there could be a table specifically for connecting users with friends. Additionally, this still worked for the Normalisation.

20.04.05 Facebook Normalisation.docx

UNF	1NF	2NF	3NF
User ID User First Name User Last Name User Hometown User Gender Relationship Status User Town/City (User Workplace Workplace Date) (User School/Uni School/Uni Date) (Friend ID Friend First Name Friend Last Name Date/Time Sent Text) Sender ID Receiver ID	User ID User First Name User Last Name User Hometown User Gender Relationship Status User Town/City User ID User Workplace Workplace Date User ID User School/Uni School/Uni Date User ID Friend ID Friend First Name Friend Last Name Sender ID Receiver ID Date/Time Sent Text	User ID User First Name User Last Name User Hometown User Gender Relationship Status User Town/City User ID User Workplace Workplace Date User ID User School/Uni School/Uni Date User ID Friend ID Friend ID Friend First Name Friend Last Name Sender ID Receiver ID Date/Time Sent Text	User ID User First Name User Last Name User Hometown User Gender Relationship Status User Town/City User ID User Workplace Workplace Date User ID User School/Uni School/Uni Date User ID Friend ID Friend ID Friend First Name Friend Last Name Sender ID Receiver ID Date/Time Sent Text

The final Entity Relationship Diagram was actually initially made on draw.io, but it was brought to attention that it needed to be generated by MySQL such that it could be compared to other diagrams in the project (as it would be guaranteed to represent the database). It should be noted that while it is perhaps considered unconventional to see *two* relations between messages and users, this was done because it was realised that both the "sender_id" and "receiver_id" were linked to the "user_id" (as they were both users); more information on this is provided below when regarding the Messages table.

Final ERD.mwb



SQL Query Statements:

The Users table was one of the simplest to create, containing no foreign keys and only data of type VARCHAR (variable characters). It had one primary key, the “user_id”, and was the basis for the connection between the other tables. It had no need to be edited at any point. The VARCHARs were set to limits that were thought to be appropriate – their values encompassed what was thought to be the likely maximum limit for each attribute (for example, double-barrelled surnames and those with two forenames were two considerations that affected the decision about the values used). It should be noted that the “relationship_status” attribute was dropped temporarily as the importing failed when the column was present. It was brought back, despite its lack of data, seeing that it was populated by further SQL Query statements. The wizard was admittedly used to bring the column back, but it effectively never left the database (it *only* did due to failed imports), so it is still listed below as the table was originally created.

```
CREATE TABLE isad157_jkinver.users
(
user_id VARCHAR(15) NOT NULL,
user_forename VARCHAR(40) NOT NULL,
user_surname VARCHAR(40) NOT NULL,
user_hometown VARCHAR(50) NOT NULL,
user_gender VARCHAR(12) NOT NULL,
relationship_status VARCHAR(20) NOT NULL,
user_town_or_city VARCHAR(50) NOT NULL,
PRIMARY KEY (user_id)
);
UPDATE isad157_jkinver.users
SET user_relationship_status = "Currently Unknown";
```

The Workplace(s) table was the subsequent table created. It wasn’t quite as simple as the Users table as it contained a composite key and a foreign key, both of which were manually programmed in through query statements (rather than by use of the wizard). It may have been the case that the ‘constraint’ was unnecessary, but it was seen fit as to better the understanding of coding a foreign key (the understanding being that the foreign key itself was the “user_id”, where its ‘name’, if you will, was “workplace_user_id”). It should be noted that, similar to “relationship_status” from the Users table, the “user_workplace_date” was dropped temporarily from the Workplace(s) table. Again, no data was given for the column, meaning it had to be manually populated (by the aid of more statements). Additionally, it was originally thought of to make the “user_workplace_date” of type DATE, but because it stated from *and* until dates, it made more sense to be a VARCHAR.

```
CREATE TABLE isad157_jkinver.workplaces
(
user_id VARCHAR(15) NOT NULL,
user_workplace VARCHAR(60) NOT NULL,
user_workplace_date VARCHAR(25) NOT NULL,
PRIMARY KEY (user_id, user_workplace),
CONSTRAINT workplace_user_id FOREIGN KEY (user_id)
REFERENCES isad157_jkinver.users(user_id)
ON UPDATE CASCADE
ON DELETE CASCADE
);
UPDATE isad157_jkinver.workplaces
SET user_workplace = "N/A"
WHERE user_workplace = "";
```

```
UPDATE isad157_jkinver.workplaces
SET user_workplace_date = "N/A"
WHERE user_workplace = "N/A";

UPDATE isad157_jkinver.workplaces
SET user_workplace_date = "4 Sep 2017 - Present"
WHERE user_workplace <>"N/A";
```

The Schools/Universities table was virtually identical to the workplaces table. It too included two primary keys forming a composite key, a foreign key, and the same datatypes (even down to the number of characters for the VARCHARs). This was deliberate, as its function was almost exactly the same: the only difference between the two really was the content: for where one table detailed the workplace(s) of each user, the other stated the educational establishments of which they were a part. Such repetition is also arguably a reasonable design choice: it demonstrates consistency in the database, which might arguably reinforce the user's understanding of the UI. Another similarity between this table and the Workplace(s) (as well as the Users) table was the temporary exclusion of a field. And, as was the case for the Workplace(s) table, the date column was the one to be dropped for the same reason as the other two. Furthermore, as was so with the Workplace(s), it was temporarily considered to make the date of type DATE, until it was realised that it wouldn't work.

```
CREATE TABLE isad157_jkinver.schools
(
user_id VARCHAR(15) NOT NULL,
user_school_or_uni VARCHAR(60) NOT NULL,
user_school_or_uni_date VARCHAR(25) NOT NULL,
PRIMARY KEY (user_id, user_school_or_uni),
CONSTRAINT school_user_id FOREIGN KEY (user_id)
REFERENCES isad157_jkinver.users (user_id)
ON UPDATE CASCADE
ON DELETE CASCADE
);
UPDATE isad157_jkinver.schools
SET user_school_or_uni = "N/A"
WHERE user_school_or_uni = "N/A";

UPDATE isad157_jkinver.schools
SET user_school_or_uni_date = "N/A"
WHERE user_school_or_uni = "N/A";

UPDATE isad157_jkinver.schools
SET user_school_or_uni_date = "23 Sep 2018 - Present"
WHERE user_school_or_uni <>"N/A";
```

The User-Friend Link table was arguably the simplest table to create. It only had two fields, both of which formed a composite key and one of which was a foreign key, and the only datatype used for it was, again, VARCHAR. Though initially it was thought of as a good idea to make both fields foreign keys so as to link the now-removed Friends table to the others, it was realised (with the removal of the Friends table) that only the user_id needed to be a foreign key, as that would be enough to link the User-Friend Link table to the other tables. The former Friends table had no link otherwise, but with the dataset not actually containing a separate file for friends' IDs, forenames, and surnames (on account of the significant point that friends are themselves considered users for this database), it was arguably an

unnecessary inclusion, despite its place in the normalisation. Hence, the table was dropped. This is also the reason for the somewhat strange naming of this table compared to the others.

```
CREATE TABLE isad157_jkinver.user_friend_link
(
user_id VARCHAR(15) NOT NULL,
friend_id VARCHAR(15) NOT NULL,
PRIMARY KEY (user_id, friend_id),
CONSTRAINT link_user_id FOREIGN KEY (user_id)
REFERENCES isad157_jkinver.users (user_id)
ON UPDATE CASCADE
ON DELETE CASCADE
);
```

Again, the Friends table was seen, with regards to the database, as an unnecessary inclusion as it would only cause the repeat of data, which is something to avoid in a relational database. Following is the code used to create the Friends table before it was completely removed:

```
CREATE TABLE isad157_jkinver.friends
(
friend_id VARCHAR(15) NOT NULL,
friend_forename VARCHAR(40) NOT NULL,
friend_surname VARCHAR(40) NOT NULL,
PRIMARY KEY (friend_id)
);
```

The Messages table was another table that was, initially, not overly complex, containing only a composite key and a total of four fields. There lay a problem, however, which was that it ended up not having a direct relation to any other table in the database (which was seen in the first Final ERD). Resultantly, the decision to make a foreign key to relate them was made. It was additionally realised that because the “sender_id” and “receiver_id” were interchangeable to some extent, and with friends themselves being users, it was seen as fit to relate both values to the “user_id”; both the sender and receiver were of course their own user, after all. Because of this, two foreign keys were made for this table, and because the thought materialised after the creation *and* population of the table, it was not a simple option to drop and re-code the table in this case: the wizard was admittedly used to create these two particular foreign keys. As such, the code that follows does not contain the creation of the two foreign keys (though above query statements prove it is known how to create them manually).

```
CREATE TABLE isad157_jkinver.messages
(
sender_id VARCHAR(15) NOT NULL,
receiver_id VARCHAR(15) NOT NULL,
date_and_time_sent DATETIME NOT NULL,
content VARCHAR(640) NOT NULL,
PRIMARY KEY (sender_id, receiver_id)
);
```

The only other query statements to show are the SELECT statements used by MySQL through C#. The queries were declared as strings each named “query” in the C# program and their values were assigned to exactly what is shown below (one line per query). Through the use of these strings, the tables in the database could be viewed through the C# application in data grids: one grid for each table.

```
"SELECT * FROM isad157_jkinver.users"
"SELECT * FROM isad157_jkinver.user_friend_link"
"SELECT * FROM isad157_jkinver.workplaces"
"SELECT * FROM isad157_jkinver.schools"
"SELECT * FROM isad157_jkinver.messages"
```

Conclusion:

What can be said overall about this project is that while the database works and displays all the necessary data, it is certainly arguable that it could have had more functionality added to it with more research behind the process of adding it. The (majority of the) diagrams are clear and describe CRUD functions as well as the main user's capabilities; the scenario chosen allowed for a second user to be invented (the TPDM) that would naturally include the CRUD elements in the diagrams without making the SMI have more access than what was wanted for their role.

https://github.com/jkinver/ISAD157-Coursework

[Word Count (excluding all Links and SQL Queries): 2,048 words]
[Please Note: It appears the links do not work – if they continue not to, use the screenshots of the diagram provided]