FINAL PROJECT

CSCI 3901 Winter 2021

Introduction

This project was made as part of the curriculum for the course CSCI 3901 (Software Development Concepts) for the January 2021 class. While creating this project, the concepts that were taught in the class throughout the term have been extensively applied. Concepts like Version Control, ADT implementation, Testing, Debugging, Good Software Design Principles, Database Designing and Usage etc. have been made use of to create a working solution. The overall design of the project follows the design principles of what a good software engineering project should look like. The project has made us of SOLID design principles as taught in the class.

The project is defined by 3 classes, two of them MobileDevice and Government class are the main working classes where the MobileDevice class is used to access the functionalities of the Government Class. We make use of the third class as a running class in which we invoke the methods of the MobileDevice class and which inturn, invokes the methods of the Government Class. This approach enables a smooth and secure control and data flow of the system. While building the solution, a considerable amount of thought has been given onto the fact that there are many aspects of the project which are to be assumed to be there and whose functionalities will be done by the means of method calls. So bearing in mind of that, the solution has been developed to encompass all the said related problems. Below is the figure which describes the common flow.

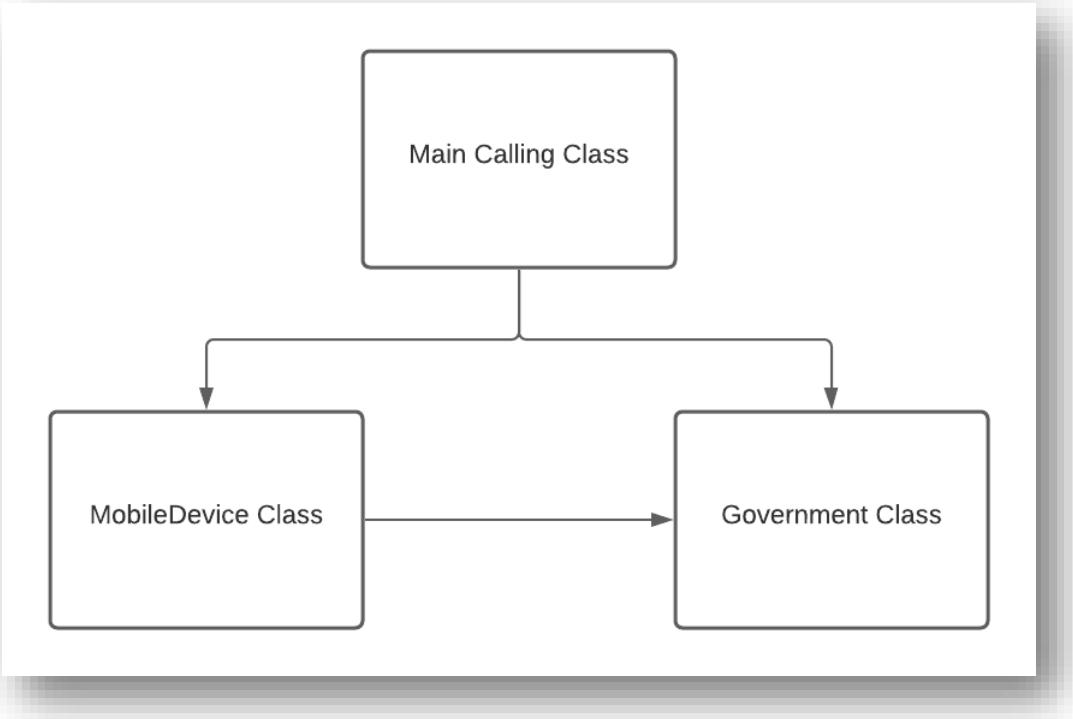

Fig 1: **The Flow of the program**

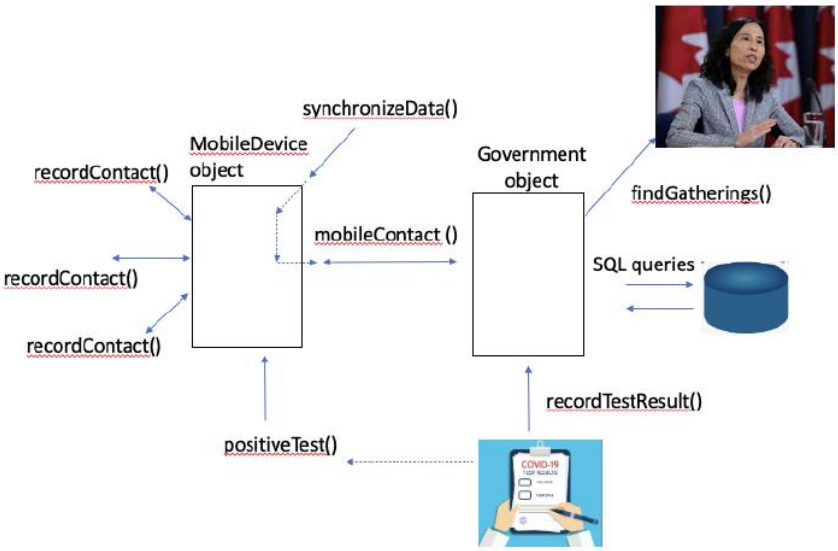

Fig 2: Proposed Structure of the System (taken from the final assignment)

Explanation

In the beginning, we firstly must look at how the project would begin, how the mobile devices will be conceptualized. The question arises, how does the project begin, how is the data which is to be processed is prepared? To answer these questions, I will take a step-by-step approach to explain how the project is designed and how it does the operations.

Step - 1 Data Preparation

- Firstly, I created the configurations required for the Government Class, an object was created which stores the properties of the class, such as the username and password of the Database server to which the program will connect to. The URL of the server is also present in the configuration file. We make use of the Properties class to store these properties of the Government Class. I have made use of the objects of the MobileDevice class to conceptualize a mobile device belonging to an individual. Each mobile device has some properties such as its Address, Device Name and finally, Device Hash. I have created objects of all the devices beforehand along with their properties. To store the properties of the mobile devices, I have used the Properties Class of Java which uses a Key-Value pair annotation to store the configuration of the mobile devices. It is an efficient and cleaner way to define the properties of the mobile devices. Along with this the combination of the device's name & address are used as a parameter for the **HashGenerator(),** this is a static method which is used to generate the hash of the device using SHA-256 hashing format using Java's MessageDigest Class, this function returns the device hash which is a secure way to store a device's information. These devices would be referred by their hashes throughout the program, the government will not be able to identify the mobile devices individuals as only hashes of the devices are used. The MobileDevice class has a

parameterized constructor which takes the configuration file of the mobile device and the object of the government class, **contactTracer**. This ensures great deal of privacy and security for the users. When we are done instantiating the mobiledevices with their configurations and the government class object. We will then move onto the next step.

Step - 2 **Recording Data**

- In an ideal scenario, this would be done by a Bluetooth device, but since there is absence of this. To conceptualize the devices meeting each other, I have made use of the **recordContact** function. This method takes the hash of a mobile device which came into contact with the host device along with the date and the duration of the contact. The date here is the number of days since 1st January, 2021. To manage the dates, we make use of the **SimpleDateFormat** class, which parses the strings into the proper format. This method makes use of a HashMap to store the contacts of a device such that, each key is a contact instance, which in simple terms is Contact-ID and the value are the properties of that contact, which include the contact device's hash, date of contact and the duration of the contact. So the HashMap stores integer as its key and a properties type variable as its value. This serves as the local memory of a mobile device, in which the contacts of a mobile device are listed in a very easy to understand and secure manner.
  Further, the user also needs to enter the positive test hashes incase he receives a positive test report. So initially we call the **positiveTest()** method from the main class which sends a test hash as its parameter. We again, employ an arraylist and a hashmap to store the test hashes of a device. In some cases, a user might have more than 1 test hashes so we have made use of arraylists and hashmaps to accommodate that situation aswell. We also employ a global test register, which saves the tests on the basis of the devices hashes, such that each key is a device hash and the value is the list of the tests reported by the user.

  In addition to this, the Government class's method **recordTestResult()** is used to record the tests onto the government's database, it uses parameters such as the hash of the test, the date of the test and lastly, the result of the test. We make use of the SQL queries to insert the data into the **TestRecord** Table.

Step - 3 **Preparing Data for Synchronization**

- After the data has been stored locally, then the task of preparing the data to be synchronized with the government database server has to be performed. To conceptualize this process we have made use of the **synchronizeData()** such that this method is called by a device to synchronize the data that is stored locally to the data onto the database of the government. We make sure that only new data is synchronized. Along with this, the most important role of this method is to let the user know if they have been in contact with a device whose owner has tested positive for COVID-19 in the last 14 days. We take the absolute of the date of the test such that if any user came in contact prior or after 14 days of taking the test, they are then alerted by updating the **devicerecord** table in the database and

when their mobile devices synchronize with the government database, they are told that they have been in contact with a COVID-19 positive person. Firstly, to prepare the data to be sent to the government server, we make use of the **XML** File Format, such that by making use of the **Documentation Object Model**, we use The **W3C Document Object Model (DOM)** to convert the string containing the information of the device such as its contacts, date of contacts and the duration of contacts into a XML Format String, we name that string as **contactInfo.** We make use of Transformers to transform the string into a XML string. The reason for converting it into a XML format is that we made use of the Document Model in which we create a Tree like structure of the document. Such that this structure is easy to understand, we created nodes for the device's own hash, the device's contacts hashes, the dates of contacts and their corresponding durations. All of this information is packed into a XML String which is then sent to the government using the **contactTracer** object. The object calls the **mobileContact** method using this object and the data is then sent to be processed by this method.

Step - 4 **Data Processing**

- The **mobileContact** receives the **initiator** and the **contactinfo** as its parameters. This function is responsible for parsing the information that is received in the contactinfo string. The initiator string contains the hash of the device, along with this the contactinfo XML String, contains all the information related to the contacts. It uses DocumentBuilderFactory and the StringReader instance to convert the XML String into a document object. This is done so that we can easily access the information stored in the elements of the document by their tag names. This method also establishes connection to the Government's database using the **ConnectionEshtablisher()** method such that it uses the information of the Government's configfile which we previously initialized when we created the object. The ConnectionEshtablisher method takes a string parameter in which we send the initiator such that it does a query to the database which involves the hash of the source device. The resultSet is then assigned this returning table and on the basis of this table the mobileContact method begins its processing. Firstly, we check if the resultSet is empty, if it is then it means that the database is empty, on the other hand if it is not then it means there are already entries in the database with that device hash. In case the resultSet is empty, we then firstly check the contacts of the source device, if the contacts of the device have tested positive within 14 days, the main purpose of this method has been achieved, it has found that the user was **in contact with a covid positive person in the last 14 days.** We set the value of a Boolean variable comecontact which says that it has come in contact. Moving ahead, we then add the contacts of this device and its own record onto the database in the **contact_tracker** table, along with this the test results of the device are also matched with the government's test results and are verified. On the other hand, if the device already had its record, we check if there are any new records, if there are we then add them to the database and then perform a check to see if the contacts of this device have turned positive since the contact date. If they have, we set the cometrue variable as true. We also look at any pending test hashes that might not have been verified with the government

database. We verify them and check that the results match with the government's results. Finally, we check if the Boolean variable comecontact is true, if it is true, that means that the device has come in contact with a covid positive person and hence we return **true** the calling function which in turn also returns true to the main calling function.

Step - 5 Private Methods Aiding in Processing of Data

- **ConnectionEshtablisher –** This method takes the initiator, which is the hash of the host device and then it establishes connection with the database using the configuration of the government class. In addition to this, it executes a query in which the resultSet is assigned the information returned from the query which asks the database to return the information associated with this device from the **devicerecord** table.
- **WithinPositveRange** – This method is used to tell if a given contact date falls within the positive date range.
- **daysAddition –** This method is used to add days to a given date.

Step - 6 Finding the gatherings on a given date

- In this method we tend to make use of properties of the Sets, such that we firstly make a query to the database in which we set the condition as we want to find the contacts that happened on the pre-determined date.
- In the resultant set, we then find the pairs of individuals that contacted each other on that given day.
- We then find the intersections of these pairs following which we then set a constraint that a set must have a minimum number of individuals in it.
- We then make use of the previously saved HashMap which contained the duration of the contacts, to retrieve the duration of the contact, we check if the duration is equal or above the benchmark of minimum duration which is to be considered.
- Then we count the number of individual pairs in this Set (c), we also count the maximum number of possible pairs we had prior to enforcing the mintime constraint (m).
- We then create the ratio of c/m, such that if this ratio is equal or more than density then this gathering is considered and the gathering counter is incremented.
- Finally, in the end we return the gatherings to the main function.

Database Design – I have created three tables by the name of devicerecord, contact_tracker and TestRecord. All of these tables are independent of one another, here

- **devicerecord** table -
  - devicehash varchar **[not null primary key]**
  - contact_list mediumtext
  - last_contact varchar
  - positive_contact varchar
  - pcontact_date varchar

- positive_status varchar
- ptest_date mediumtext
- usernotified varchar

- **contact_tracker** table **–**
  - source_device varchar [not null]
  - contact_device varchar [not null]
  - contact_date varchar [not null]
  - contact_duration varchar [not null]

- TestRecord –
  - TestHash varchar [not null PRIMARY KEY]
  - TestDevice varchar
  - testDate varchar [not null]
  - TestResult varchar [not null]

Test Plan – The following types of tests were done by me to ensure proper operation of the program. Testing strategies such as unit testing, integration testing and system testing were done to ensure that the program was working properly. This is in addition to the extensive testing which was done on different types of data which ensured that the constraints of the program were followed. I have also made use of JUnit testing framework for unit testing the program, to make sure that the results of the program were proper when illegal data was thrown onto it. The presence of constant checks ensures the same robustness, while doing integration testing it was made sure that to check how cohesive the code is and to determine that there is low coupling between essential snippets in the code. These were done while keeping the principles in check. Here are the testing techniques with which the program has been tested with :-

1. **Input Validation Tests (A <u>Black Box Testing</u> Technique) –** Here we tabularize the input validation tests for methods given in the problem statement, the Test Case Description column gives a brief description of the test case, the Test Case Input field gives an idea of how the input would be entered, the Expected Result Column displays what would happen according to the input entered, an assumption of how a certain method would work, in the essence, these following test cases are just a way of checking of how the program acts under unforeseen situations, to make sure that the software behaves in a predictable manner despite unexpected inputs.

   - **MobileDevice Class**

- **Method** - recordContact(String individual, int date, int duration)

Test Case #	Test Case Description recordContact(individual, date, duration)	Expected Result
1	When the individual string is empty or null	Invalid Data
2	When date is negative	Invalid Data
3	When duration is 0 or negative	Invalid Data

- **Method** - positiveTest(String testHash)

Test Case #	Test Case Description positiveTest(testHash)	Expected Result
1	When TestHash is empty	Invalid Data
2	When TestHash is null	Invalid Data

- **Method** - MobileDevice(String configFile, Government contactTracer)

Test Case #	Test Case Description MobileDevice(configFile, contactTracer)	Expected Result
1	When the configFile is empty or null	Invalid Data
2	When the contactTracer object is initialized with empty or null data	Invalid Data
3	When duration is 0 or negative	Invalid Data

❖ Government Class

- **Method** - Government(String configFile)

Test Case #	Test Case Description Government(configFile)	Expected Result
1	When the configFile is empty or null	Invalid Data

2	When the object is initialized with empty or null data	Invalid Data
3	When duration is 0 or negative	Invalid Data

- **Method** - boolean mobileContact(String initiator, String contactInfo)

Test Case #	Test Case Description boolean mobileContact(initiator, contactInfo)	Expected Result
1	When the initiator is null or empty	False
2	When the contactinfo is null or empty	Invalid Data

- **Method** - Government(String configFile)

Test Case #	Test Case Description Government(configFile)	Expected Result
1	When the configFile is empty or null	Invalid Data
2	When the object is initialized with empty or null data	Invalid Data
3	When duration is 0 or negative	Invalid Data

- **Method** - recordTestResult(String testHash, int date, boolean result)

Test Case #	Test Case Description recordTestResult(testHash, date, result)	Expected Result
1	When the testHash is null or empty	Invalid Data
2	When date is negative	Invalid Data

- **Private Methods**

- **Method** - private static String HashGenerator(String deviceName)

Test Case #	Test Case Description HashGenerator(deviceName)	Expected Result
1	When the deviceName is empty or null	Invalid Data

- **Method** – private boolean ConnectionEshtablisher(String initiator)

Test Case #	Test Case Description ConnectionEshtablisher(String initiator)	Expected Result
1	When the initiator is empty or null	False

- **Method** – private voidString daysAddition(String date, int days)

Test Case #	Test Case Description String daysAddition(date, days)	Expected Result
1	When the date is empty or null	Invalid Data

- **Method** - private boolean WithinPositveRange(String cdate, String ptdate, String ftdate)

Test Case #	Test Case Description HashGenerator(deviceName)	Expected Result
1	When cdate, ptdate and ftdate are null or empty	Invalid Data

2. **Boundary Testing –** In this technique we test the extreme boundaries of the input values, because input values near the boundaries have a higher chance of error.

- **Method** - recordContact(String individual, int date, int duration)

Test Case #	Test Case Description	Expected Result
1	When individual's testhash has only 1 character in the string	Contact Recorded
2	When individual's testhash is very large	Contact Recorded

3	When the date is 0 days	Contact Recorded
2	When the duration is 1	Contact Recorded
3	Date is a large number of days	Contact Recorded
4	Duration is a large number of minutes	Contact Reported

- **Method** - positiveTest(String testHash)

Test Case #	Test Case Description	Expected Result
1	When testHash is only 1 character long	Recorded
2	When the testhash is very long	Recorded

- **Method** - MobileDevice(String configFile, Government contactTracer)

Test Case #	Test Case Description	Expected Result
1	When the configFile contains only 1 characters of information for every property	Intialized
2	When the configFile contains large characters of information	Initialized

- **Method** - boolean mobileContact(String initiator, String contactInfo)

Test Case #	Test Case Description	Expected Result
1	When the initiator or contactinfo contains only 1 characters of information for every property	Accepted
2	When the initiator or contactinfo contains large characters of information	Accepted

- **Method** - recordTestResult(String testHash, int date, boolean result)

Test Case #	Test Case Description	Expected Result
1	When the TestHash or contactinfo contains only 1 characters of information for every property	Recorded
2	When the TestHash contains large characters of information	Recorded
3	When Date is 0	Recorded
4	When Date is a large number	Recorded

- **Method** - int findGatherings(int date, int minSize, int minTime, float density)

Test Case #	Test Case Description	Expected Result
1	When Date is 0	Accepted
2	When Date is a large number	Accepted
3	When minSize or minTime is 1	Accepted
4	When minSize is 0	Accepted
5	When minSize or minTime is a large number	Accepted
6	When density is 0	Accepted
7	When density is 1	Accepted

3. **Data Flow Testing (White Box Testing Technique) –** In data flow testing, we test how the data flows through the code, in this we invoke methods firstly in the 'normal order' to see how the program would function under normal circumstances, then this is followed by invoking methods in different orders to see how the data objects are being defined, initialized and being used.

Test Case #	Test Case Description	Test Case Input	Expected Result

1.	The normal order of the program	Create Instances of MobileDevice
		recordContact();
		recordTestResult();
		positiveTest();
		synchronizeData();
		mobileContact();
		findGatherings();
			True or False is returned depending on the data. Gathering number is returned

2.	Call synchronizeData first	synchronizeData();	False
3.	Call positiveTest() first	positiveTest();	Depends on the next calls
4.	Calling findgatherings before recording any contact	findGatherings();	0
5.	Calling synchronize before and after recording contact.	synchonizeData() recordContact() synchronizeData()	The new contacts will be recorded, true or false will be returned depending on the positive status of these contacts
6.	Calling loadpuzzle and solve() without calling validate	loadPuzzle(stream) solve()	False
7.	Calling recordTestResult and then calling synchronizeData()	recordTestResult(); synchronizeData();	The Test Result will be recorded by the method itself and thensynchronizeData will be called

4. **Control Flow Testing (White Box Testing Technique) –** In control flow testing, we test the different structures that exist in the software, how the sequence of instructions executes to better understand the internal implementation of the program. To make sure that each and every element of the code is working correctly.
    - **Many Control Flow cases of the above methods have already been covered under boundary test cases, so I have not mentioned them in the following table.**

Test Case #	Test Case Description	Expected Result
1.	Calling recordContact again and again on the same data	The duration of the contact will be added
2.	Calling synchronizeData after getting in contact with a positive person	True

3.	Calling synchronizeData again after getting notified that one has been in contact with a positive person	False
4.	Calling recordTestResult again and again on the same testhash	Won't be allowed, integrity constraint violation
5.	Calling findgatherings again and again on the same date when there was no contact on that date.	0
6.	Calling postiveTest multiple times on the same testHash	Only one testHash will be recorded, rest will be integrity constraint violations

Submitted By – Harjot Singh (B00872298)