I. PROJECT DESCRIPTION

For concealment purposes during World War II, the German military used what's known as Enigma Machines to encrypt communications between army bases. With the help of those machines, the Germans inflicted heavy losses to the allies. The fatalities persisted until the day a genius mind named Alan Turing along with his team in Bletchley Park in England had finally cracked the codes. This project implements in Java programming language a simplified form of World War II German encryption machines. One variant of a complete form of them included three rotors, a reflector, a plug-board and rotor rings. When those parts were put in a coordinated manner, they were cleverly capable of encrypting messages with an astronomical number of ways. Thankfully, the machines had one deadly error: The exact same algorithm was used for both encryption and decryption. In this project and for the purpose of reducing design complexity, only the three rotors and the reflector are featured. In order to encrypt text in our case a proper setting of the machine to predetermined rotor positions must be done. After that, the process of encryption takes each character and puts through the wirings from left to right via the leftmost rotor, to the middle rotor, to the rightmost rotor and finally arrives to the reflector. At this stage, the character has now accomplished its half journey through the wirings. The second half is done from the reflector back to the rightmost rotor, passing through the middle rotor and finally through the leftmost rotor to be ready as an encrypted character. In other words and after proper rotor settings, a character is encrypted using seven different stages of encoding to leave place to another combination of rotor settings for the next character. As for decryption, the same exact process is applied using the same exact rotor positions that were used at the beginning of encryption.

THE ROTORS AND REFLECTOR MECHANISMS:

1- There are 26x26x26 different combinations of rotor settings witch are equivalent to 17576 wiring possibilities as each rotor has 26 input/output letters.
2- Only uppercase letters are encoded, all other character are left as they are.
3- Rotors advance only after an uppercase letter is encrypted, no proceeding otherwise.
4- Left most rotor advances one unit after each uppercase character and returns to 0 after reaching 25. After that, the middle rotor advances one unit. When the middle rotor passes from 25 to 0, the rightmost rotor advances one unit.
5- Rotors must have initial positions and move only after a character is encrypted. The three rotors and the reflector are wired in both directions. From left to right, all four parts are equipped with the 26 letters of the modern alphabet (A to Z). As for the opposite direction, its done as follows:

LEFT ROTOR	**: QWERTYUIOPLKJHGFDSAZXCVBNM**
MIDDLE ROTOR:	**ZAQWSXCDERFVBGTYHNMJUIKLOP**
RIGHT ROTOR	**: PLOKMIJNUHBYGVTFCRDXESZWAQ**
REFLECTOR	**: NPKMSLZTWQCFDAVBJYEHXOIURG**

HOW DOES THE ENCRYPTION & DECRYPTION WORK?

As said before, each rotor is set to a specific position in the beginning and then advances according to the rules cited above. Therefore, to encode a character from left to right for example, the following formula is used: `[(indexOfLeftCharacter(c) + position) % 26]`. With **"c"** representing a character, and "**position**" representing rotor position at a particular time. Now suppose that the rotor positions are **(5, 9, 14)** and we want to encode **"A",** the computations would be**:** (0+5) % 26 = "5"="Y" for left rotor, (5+9) % 26 = "14"="T" for the middle one, and (14+14) % 26 = "2"="O" for the right rotor. This translates to: A-->Y-->T-->O Then O-->V (O and V are at position 14 in the reflector). As for encoding from right to left the following formula is used:

`[((indexOfRightCharacter(c) - position) + 26) % 26]`. To continue our example, the computations would be: ((14-14)+26)%26 = 0 = "A" for the right rotor, ((0-9)+26)%26=17 ="S" for the middle one, and ((17-5)+26)%26 = 12="M" for the left rotor. This translates to: **V-->A-->S-->M.**

The entire process is**:** (input) **A ===> Y** ---> **T** ---> **O ---->V ----> A ---> S ---> M** (output).

THE PROJECT SERVES THREE MAIN PURPOSES:

1- Encrypting and decrypting a string of characters.
2- Discovering the proper rotor settings used to encrypt a text in real English.
3- And finally decrypting an entire text file using the previously discovered settings.

To achieve goal #1, the encryption and decryption use the same logic as described in the example above. The exception is that, instead of one character, a stream of characters is put through the machine and a function that encrypts a line is provided in the source code. However, each time a character in encrypted the rotors must advance according to the rules cited above. The goal is achieved successfully, the source code as well as a listing of the run are provided for more detail.

The achieve goal #3, the process is straight forward as once the proper rotor settings for English were discovered, we just need to set the rotors to their initial positions, a way to read data from the encrypted file (in this project ***encrypted.text***), then encode it line by line. Each time a line is decrypted, it must be stored in a specified storage file (in this project ***decrypted.text***).

The tedious task is how to discover the proper settings that were used to encrypt the file? In order to achieve, the goal we need to go through all **17576** possible wirings through triple-looping and set the rotors to one position at a time as follows: `em.setRotors(i, j, k)` with "em" as an enigma machine object. The **indexes i, j, and k** represent left, middle, and right rotors. From there, we need to decrypt a number of lines from the encrypted file (`decryptLines(em, dataFile)`**).** Once, the lines are decrypted, we need to test whether the result matches real English `if (matchesEnglish(decodedLines)).` To achieve this purpose, we first need to count the occurrence of each letter individually `countAllLetters(String s)` ,and store the results in an array. After that, we need to count errors according to the frequency of how they appear in a text of English. As the frequencies are not always exact, each letter has an allowable deviation percentage. Last but not least, the algorithm decides whether a certain decryption is allowed for review by aye if it does not exceed an allowed number of errors. If it matches these conditions, then the lines will be stored in a separate file for review along with the combination of rotor settings (`storeLines(i, j, k, decodedLines, storageFile);`) and (`rs.saveRotorSettingsForEnglish(i, j, k);`). Lastly, and since we cannot be sure about when to stop, we to run through all the **17576** combinations, so the latter counters need to be reset to 0 to make place for the next iteration. (`English.resetLetterCounters();`). All the details in the source code are given. In this project the number of lines to decode was put to 10-12 lines, allowed errors to 7 and a multiplier to 2.5. The multiplier allows to a little above and a little bellow the allowed character deviations. The goal was successfully achieved with exactly one combination of rotor settings that happen to be the one used to encrypt the file. The **Computation #: (23, 12, 17)**

PROJECT'S CLASS DESIGN AND HOW THEY SERVE THE THREE MENTIONED PURPOSES:

The project contains eight classes in total. A class that designs how the rotors work, a class that extends the Rotor class designs how the reflector works, an class that designs the mechanics of the machine named the EnigmaMachine class, a class that encapsulates information related to the English Language, and finally a class that helps save then recover the proper rotor settings needed to retrieve a text in real English and it's called FindRotorSettings. There are two additional user classes, an EnigmaMachineUser class that is used to encrypt and decrypt a character string using the same rotor settings for both encryption and decryption. The other user class called DecryptUser is used to run the algorithms needed to discover real English, return the proper settings used to encrypt a text in English and finally decrypt an entire text file.

CONCLUTION:

The Enigma Machine project was such a delight for one to think about as it involves at the same time a piece of world history, discovering Alan Turing's Bletchley Park team and thinking about the tremendous effort that was required in order to crack Enigma. Implementing it using a computer and a programming language is an eye opener to the deep horizons that we can access through programming a computer. All the main goals were successfully implemented and produce the desired results.

II. TECHNICAL INFORMATION

Programming language: Java with jdk8

Running on Linux OS: file extensions are (.text) and the file encrypted.text needs to be inside the main project folder. The results are generated to the IDE's console for program 8.2 and to the files "ENGLISH.text" decrypted "DECRYPTED.text". With the command line:

Running on Windows OS: file extensions are (.txt) and the file encrypted.txt needs to be present in the main project folder. The results are generated to the IDE's console for program 8.2 and to the files "ENGLISH.txt" decrypted "DECRYPTED.txt". Using the command line: > **javac {className}.java for each class**, then **> java EnigmaMachineUser.**

The development of the project started using the jGrasp IDE on Windows 10, and all classes were put inside a project folder along with the encrypted file "**encrypted.txt".** The execution is straight forward after successful compiling of all java class files. Run using the run button and choosing one of the main classes. Separation into two users is for the sake of organizing results. The same effect is achievable through one main class. Users if they wish too, can specify the name of both the English results file (here ENGLISH.txt) as well as for the decryption results file (here DECRYPTED.txt).

The project files were moved after a working version of the programs to NetBeans IDE on Windows 10 for rigorous testing and potential bug reporting. And then to Eclipse IDE on Linux Ubuntu-mate OS. Eclipse IDE is preferred as it suggested to make some of the variables package-protected to make the code more secure.