ENIGMA MACHINE EMULATOR

Candidate Name: Abubakar Aqiil

Candidate Number: [redacted]

Centre Number: [redacted]

CONTENTS

ANALYSIS

PROBLEM IDENTIFICATION

I find history very fascinating and I am particularly interested in the use of cryptography to censor sensitive military communications during the first world war. One of the successful methods employed by the German navy was the use of the Enigma machine in 1926 with later military divisions adopting the Enigma machine in 1935[1].

For my project, I will develop a program to imitate the German army's variant of the Enigma, the Reichswehr Enigma I, so that it can be used to demonstrate the cryptographic process of encrypting and decrypting messages in History lessons and to also be used by Computer science teachers to explain the complex algorithms and the workings behind the Enigma machine. I also enjoy computing solutions to mathematical problems such as those presented by the complex cipher of the enigma and that is also another reason why I have chosen this project.

RESEARCH

PROSPECTIVE USERS

The primary end user will be computer science and history teachers to use the Enigma machine model to help pupils understand the workings of the Enigma machine and how it was employed by the Germans to protect their communication of sensitive information. It will have all the original functions of the Enigma and using this model, they would be able to demonstrate and explain to students, the operation of the enigma, using the design flowcharts, and the workings behind the machine, by analysing the program code.

INTERVIEWS

1. What is the benefit of using an Enigma machine?
2. What is the most difficult thing about using the Enigma machine?
3. What are some limitations of the Enigma machine?
4. What could be improved about the Enigma machine?
5. What features would you like to see on a future Enigma?

[1] Dyson, George (2012). *Turing's Cathedral*. Pantheon. Chapter 13. ISBN 9780307907066.

Question 1 identifies what the user intends to use the enigma machine for. This will allow me to create a machine that caters towards the specific needs of the user focusing on what they aim to achieve out of the experience.

Questions 2 and 3 aim to identify what drawback the current system has and how it affects the user's experience. Identifying these issues will allow me to build a better user experience by removing the difficulties that would've been a nuisance with other systems.

Questions 4 and 5 inquire about other features that the user may wish the solution to include. This specifically identifies any function that may previously have been inaccessible by the user and resolves this, making the experience using bespoke software more enticing.

Responses from Bletchley Park representatives:

1. **What is the benefit of using an Enigma machine?**

- "Encrypts a message in such a way that it is not (usually) vulnerable to frequency analysis"
- "Gives polyalphabetic encryption in a user-friendly machine"

2. **What is the most difficult thing about using the Enigma machine?**

- "Setting up the machine is a bit fiddly and takes strength to press keys on keyboard"
- "Easy to use although the keys are quite clunky to press and bulbs can be faint if wires are loose"
- "The weight! Carrying it from our classrooms to the offices is cumbersome."
- "You have to be very careful not to make an error with the settings"

3. **What are some limitations of the Enigma machine?**

- "There aren't many but the fact that it doesn't encrypt it's own letter."
- "Weight - as it is heavy"
- "The human element of some settings, and the fact that it is a symmetric cipher"

4. **What could be improved about the Enigma machine?**

- "More rotors; allow a letter to be encrypted as itself"
- "Making it an asymmetrical cipher, which would solve the problem of key distribution"
- "Lighter and smaller"

5. **What features would you like to see on a future Enigma?**

- "A visual representation of how each letter in the ciphertext is "chosen" when each letter in the message is pressed."

- “Reminders if you have not filled in some of the settings”
- “1. Display string of letters that have been typed. 2. Display string of letters that have been illuminated on the lightboard. 3. The ability to delete letters that have been typed (remembering to update displays 1 & 2 and rotor positions accordingly) so that can correct typing errors. 4. Making features 1, 2 and 3 optional so may choose if want authentic Enigma machine or one that is easier to use.”
- “An ability to change features of the machine that can't be changed easily on a "real" Enigma machine, and see the effect that this has. For example, adding extra slots for rotors; adding new rotors (with custom wiring). How these changes affect the security of the machine, and how that compares with modern cryptographic methods.”

Responses from a History teacher:

1. **What is the benefit of using an Enigma machine?**

- “To encrypt and decrypt messages for the communication of sensitive information”
- “To demonstrate cryptology and the uses of encryption”

2. **What is the most difficult thing about using the Enigma machine?**

- “We don't have access to the machine at all and use youtube videos or book sessions for representatives for bletchley park to come in.”

3. **What are some limitations of the Enigma machine?**

- “The enigma machine cannot encrypt the same letter”
- “The are so old and delicate they must be handled with extreme care and precaution”

4. **What could be improved about the Enigma machine?**

- “Allowing a character to encrypt itself”

5. **What features would you like to see on a future Enigma?**

- “An Enigma machine that visualises the encryption process in a way that is easy to understand by students or anyone not familiar with the system”

DERIVED OBJECTIVES

Missing fields or errors

To resolve the issue of missing fields or errors with the settings, I will implement an error message which will warn the user that there are missing fields and another error message if the users enter invalid entries for the settings.

Weight, stiff keys and faint bulbs

Developing a program will easily resolve the issue of the weight of carrying the Enigma machine, making it easier to demonstrate the functions of the Enigma. This will also resolve the issue with the stiff keys as the user may use their own keyboard. Having faint bulbs will also be resolved as the ciphertext will be output onto the screen itself.

Asymmetric encryption

To preserve the authenticity of the enigma, I will not be implementing an asymmetric cipher. However, the issues regarding the weight will be resolved as specified in the above resolution.

Display

The program will display the input and output strings so that the user can keep track of what has been recorded.

Extra Options

I will incorporate the options of allowing the user to delete typed letters to correct mistakes and the option of creating custom rotor wiring to see how these changes affect the security of the machine and how they compare with modern cryptographic methods.

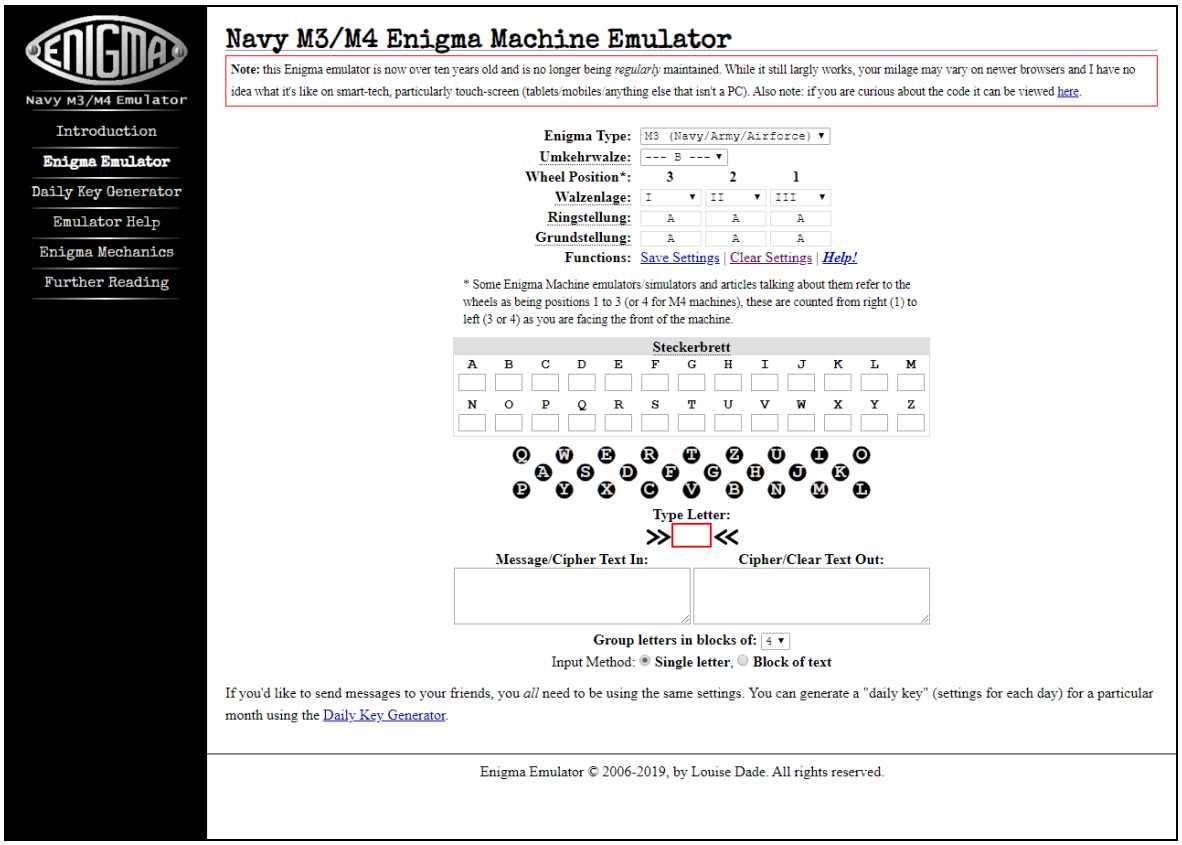
EXISTING SYSTEMS

THE REICHSWEHR ENIGMA I

The Reichswehr Enigma I is the physical enigma machine that everyone thinks of when they think of the Enigma. It is quite heavy and as there are only 350 worldwide, they are quite delicate and rare due to their age and scarcity. This means that it must be handled with extreme caution and that it is difficult to replace parts once they are broken.

I will emulate the functions that can be performed using this machine. Such functions include encrypting and decrypting with multiple rotors, using a reflector and a plugboard for better encryption, emulating the initial position of the rotors and also the ring settings (ringstellung).

JAVASCRIPT EMULATOR[2]



There was a now discontinued, javascript emulator of the Enigma machine which allowed the user to switch between the M3 (Military) and M4 (Navy only) Enigmas. It encoded messages using upto 8 different rotors but could only use 3 at any one time. It also allowed the same rotor to be used in each of the 3 wheel positions but outputted an error message if the user attempted to encode using the same rotor. The ring setting, or ringstellung, could be altered depending on the most recent character the Enigma rotors had rotated to during encryption. The emulator also featured the option to change the reflector wheel to either the B or C reflectors.

A small visual display of black bulbs that light up when outputting a letter, can also be seen. Once a user types in a key, the ‘Cipher/Clear Text Out’ textbox outputs each of the encrypted characters whilst simultaneously lighting up the corresponding bulb. It also allows the user to group the encrypted cipher text in blocks of 3, 4, 5 or 6 characters to make it easier to decipher the messages as the ‘space’ character is not encoded.

[2] Dade, Louise, (2009), Navy M3/M4 Enigma machine emulator: http://enigma.louisedade.co.uk/enigma.html

In my project, I will use the basic functionality of the enigma I used in this emulator and I will also allow the user to group the encrypted cipher text into blocks of 4 and 6 to make it easier to decipher the messages.

MODELLING OF THE PROBLEM

I will develop my project using object oriented programming. OOP is best suited for this project because each rotor uses the same attributes and methods allowing us to reuse class defined attributes and methods in each instance. Modelling the rotors as objects would allow us to use encapsulation to compile cleaner code as related variables and functions would be grouped together. Redundant code is also eliminated as inheritance allows us to reuse the same block of code throughout the program. Maintenance of the program would also be easier as abstraction would isolate the impact of future changes to the code and this will be especially useful when presenting the user with the option to create their own custom rotor wiring, plugboards and reflectors.

PROPOSED SOLUTION

The solution I will develop will be the Reichswehr Enigma. I will use Python as it is the language that I am most proficient in and it has an easier syntax than Visual Basic. I will be using SQLite to produce my database to store the exported data of the cipher table and I will also use it to read in the values from the cipher table if the user chooses to import an existing table.

SQLite is a good choice for my database because there is already an ‘SQLite 3’ module in python and there is no need for any external resources. It prevents the necessity of setting up a server if we were using MySQL for instance. Using SQLite 3 instead of a flat file will also allow us to retrieve the required fields rather than having to read the whole file into memory.

The operation of the enigma can be described in a few steps:

1. User types character into a standard keyboard
2. Character passes into a plugboard which substitutes the character
3. Substituted character passes through the rotors and changes through each rotor.
4. Reflector substitutes input character and sends the output back through steps 1-3 in the reverse order
5. First rotor steps by one (If it reaches step key, second rotor steps)
6. Reflected character passes through the plugboard and is substituted
7. Substituted character lights up respective bulb

A more detailed report on the functionality of the enigma machine along with my individual research can be found in **Appendix A**.

The user will require this as a basic functionality of the enigma. In order so that this may be achieved, the following key algorithms must be prototyped and developed:

- Substitution algorithm
  - Substitution is the main functionality of the plugboard and the reflector. This algorithm will also influence the rotors as they undergo a more complex substitution between the external and internal alphabets.
- Rotating functionality
  - Ability to make the rotors rotate so that the external alphabet cycles
- Stepping algorithm
  - The rotors must step, incrementing the following rotor, when they reach their step value

The following character table will assist us in creating these algorithms as they specify the turnover values for each rotor and they also detail the internal alphabet, or internal wiring, of the different rotors.

Enigma I character table

Wheel	ABCDEFGHIJKLMNOPQRSTUVWXYZ	Notch	Turnover	#
ETW	ABCDEFGHIJKLMNOPQRSTUVWXYZ			
I	EKMFLGDQVZNTOWYHXUSPAIBRCJ	Y	Q	1
II	AJDKSIRUXBLHWTMCQGZNPYFVOE	M	E	1
III	BDFHJLCPRTXVZNYEIWGAKMUSQO	D	V	1
IV	ESOVPZJAYQUIRHXLNFTGKDCMWB	R	J	1
V	VZBRGITYUPSDNHLXAWMJQOFECK	H	Z	1
UKW-A	EJMZALYXVBWFCRQUONTSPIKHGD			
UKW-B	YRUHQSLDPXNGOKMIEBFZCWVJAT			
UKW-C	FVPJIAOYEDRZXWGCTKUQSBNMHL			

[3]

[3] Museum, Crypto, Enigma, (2019, February), *Enigma wiring*, Engima I https://www.cryptomuseum.com/crypto/enigma/wiring.htm

SYSTEM REQUIREMENTS

- Must be able to run Python to execute applications.
- Standard peripherals needed (E.g. mouse and keyboard)
- Must run Windows, Mac or Linux for specific modules to work

MEASURABLE OBJECTIVES

Objectives added at the users request during the evaluation stage have been highlighted in *italics and underlined*.

1 Main Menu

1.1 Exit option

1.2 Encode message

1.2.1 User prompted to enter enigma settings

1.2.2 User asked if they would like to visualize encryption process

1.2.3 User prompted to enter message (*Message may be entered with space characters included for clarity*). Error message displayed when foreign characters are entered as input and user is returned to main menu (*Altered to allow the user to continue upon entering correct input*)

1.3 Decode message

1.3.1 Similar objectives for '1.2 Encode message'

1.3.2 User presented with option to group plaintext

1.4 Generate cipher table

1.4.1 Create a cipher table for the next 31 days

1.4.2 Export cipher table

1.4.2.1 File will be exported using SQLite

1.4.2.2 'datetime' module will be used to identify time and day of cipher table generation

1.4.2.3 File will be exported to folder for ease of use using the 'os' module

1.5 Load Cipher table

1.5.1 Cipher table will be read in

1.5.1.1 Error message will be displayed if there are missing fields

1.5.1.2 Error message will be displayed if the settings are outdated

1.5.1.3 Error message will be displayed if filename is incorrect

1.6 Guidance

1.6.1 Provide the user with guidance on how to operate the enigma machine and the format of the input when using the application.

2 Mechanical design

2.1 Rotor

2.2.1 Rotors must rotate

2.2.2 Stepping mechanism on rotors must step on following values respectively:

Rotor I must step from Q

Rotor II must step from E

Rotor III must step from V

Rotor IV must step from J

Rotor V must step from Z

2.2.3 Option to switch between rotors

2.2.3.1 Error message will be displayed if the user tries to use two of the same rotors

2.2.4 Option to use 3 of the 5 rotors during encryption

2.2.5 External alphabet on rotors must be matched to internal rotor wiring

2.2.6 Internal wiring must adjust to match external alphabet when a rotor rotates

2.2.7 Single step and double step instances must be recognized and completed

2.2 Reflector

2.2.1 Reflector must 'reflect' the input and output the substitute character

2.2.2 Option to use a A-reflector

2.2.3 Option to use a B-reflector

2.2.4 Option to use a C-reflector

2.3 Plugboard

2.3.1 Option to connect to different letters

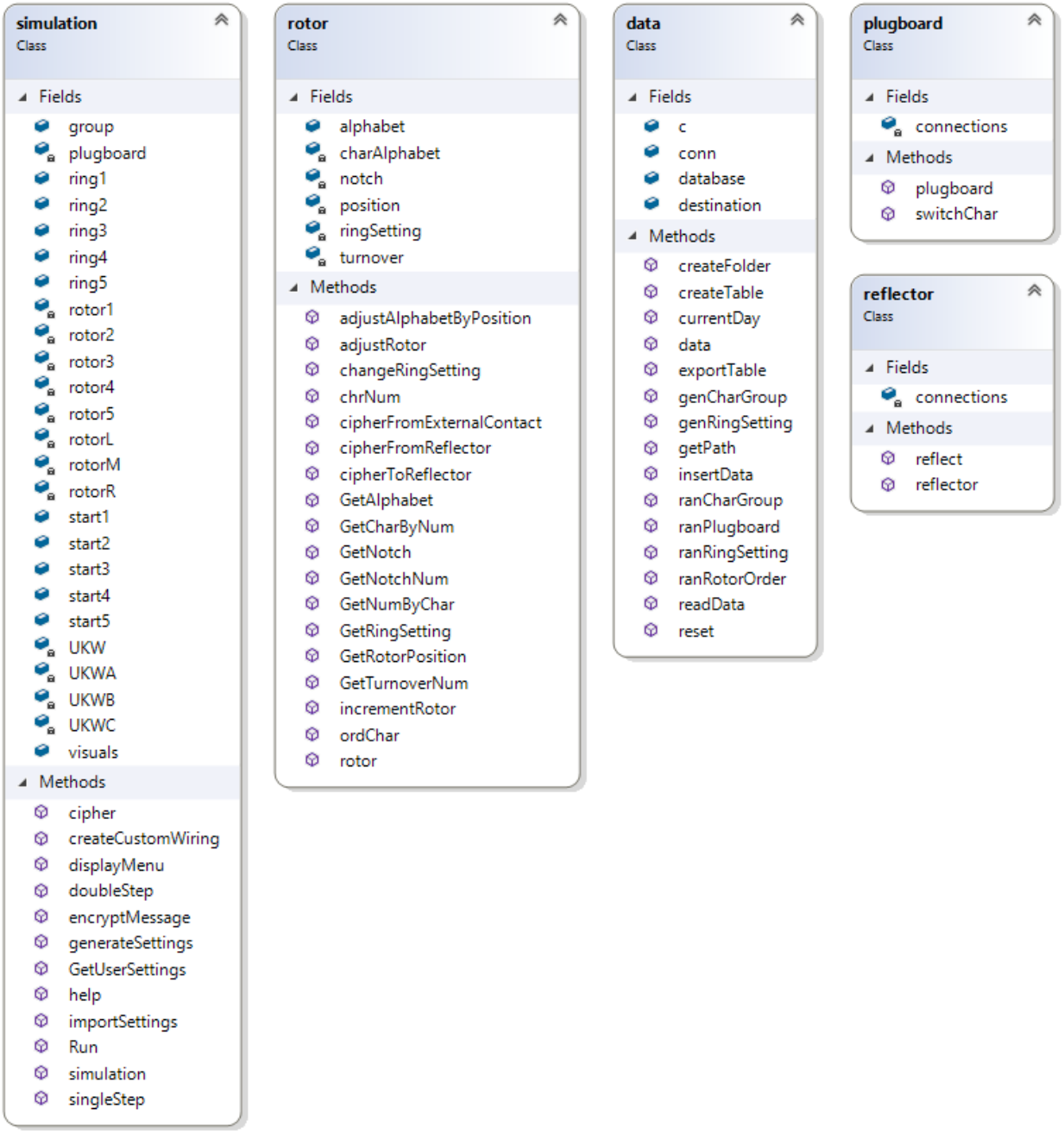
2.3.1.1 Prevent user from encrypting to the same character

ACCEPTABLE LIMITATIONS

- May not be able to encrypt modern characters, such as '@', as those were not widely used at the time and thus, they were not featured in the original encryption methods.
- There was no spacebar on the original enigma so the ciphertext will have to be grouped into words manually using the string split method where characters are grouped in 2's or 4's to make it easier to read the text.

DOCUMENTED DESIGN

HIGH LEVEL OVERVIEW



The above class object diagram shows the attributes and methods in each class providing an overview of the whole program. There will be 4 main classes in the project:

Rotor class

Creates the rotors needed for the operation of the enigma. Allows for the operation of incrementing the rotors and ciphering to and from the reflector.

Plugboard class

Allows the user to encrypt the input and output values from and to the rotors inside the enigma machine.

Reflector class

Reads the ciphered input from the user and substitutes a character to be ciphered as the output.

Data class

Manages the process of importing and exporting data to and from the enigma machine.

Simulation class

Runs the simulation of the enigma machine. Presents the user with the options to change the settings and create their custom wiring.

Some of the methods with the 'Get' prefix are not actually getter methods and the reason why they have not been changed is explained in the evaluation section.

MODULES

The table below describes the use of each of the imported modules in the application.

Module	Description
time	Allows us to create a delay before the application ends so that an end message can be output to the user
datetime	Allows us to identify the current date to import daily settings
sqlite3	Creation and use of sqlite3 databases
os	Interaction with the operating system (Windows, Mac or Linux)
random	Used to make random choices
re	Allows us to use Regular Expressions to evaluate conditions
collections	Container used to store collections of data such as arrays

METHODS AND ATTRIBUTES

The table below describes the methods and attributes in each class.

Simulation Class	Method	Description
	chiper	Takes an input and ciphers it through the enigma machine
	createCustomWiring	Takes in user input in order to create custom rotor wiring, plugboard and reflector pairs, turnover and notch positions and ring setting and rotor starting positions for experimental purposes.
	displayMenu	Outputs the main menu to the user
	doubleStep	Checks if a double step will occur on the next key press
	encryptMessage	Encrypts the message and outputs the encrypted ciphertext
	generateSettings	Creates a data object and exports the daily settings outputting the destination of the user.
	GetUserSettings	Retrieves the users settings for encryption using default settings
	help	Provides the user with guidance on the use of the application
	importSettings	Retrieves the daily settings from the enigma database
	Run	Functionality of the main menu
	simulation	Runs the enigma machine simulation
	singleStep	Checks if a single step will occur on the current key press
	Attribute	**Description**
	group	Stores the frequency of the character grouping
	plugboard	Stores the plugboard pairs for default settings
	ring1, ring2, ring3, ring4, ring5	Initialises the ring value for each rotor for imported settings
	rotor1, rotor2, rotor3, rotor4, rotor5	Creates rotor objects for imported settings

	rotorL, rotorM, rotorR	Stores the rotor type for each of the three rotors rotor
	start1, start2, start3, start4, start5	Initialises the start position value for each rotor for imported settings
	UKW	Stores the reflector type
	UKWA, UKWB, UKWC	Stores the reflector pairs for each reflector type
	visuals	Stores the user preference on visualizing encryption

Rotor Class	**Method**	**Description**
	adjustAlphabetByPosition	Adjusts the alphabet so that it can start at different positions
	adjustRotor	Rotates the rotor without incrementing it or stepping
	changeRingSetting	Changes ring setting and creates offset variable
	chrNum	Finds the ASCII code of input index position
	cipherFromExternalContact	Cipheres the users input character by incrementing the rotor before ciphering
	cipherFromReflector	Ciphers character produced by previous rotor travelling from reflector
	cipherToReflector	Cipher character produced by previous rotor travelling to reflector
	GetAlphabet	Parses the alphabet set as dictionary
	GetCharByNum	Find the corresponding character of the input number
	GetNotch	Gets notch position
	GetNotchNum	Gets numerical value of notch
	GetNumByChar	Finds the corresponding number of the input character
	GetRingSetting	Gets ring setting
	GetRotorPostion	Gets the current position of the rotor
	GetTurnoverNum	Get the numerical value of turnover character

	incrementRotor	Rotates rotor forward by one position also adjusting the internal wiring to rematch the external positions
	ordChar	Finds the index value of input ASCII code
	rotor	Creates a rotor object
	Attribute	**Description**
	alphabet	Stores the dictionary of the {positon: character} key and value pairs for each character in the rotor alphabet
	charAlphabet	Stores the rotor alphabet
	notch	Stores the notch character
	position	Stores the rotor position
	ringSetting	Stores the ring setting value
	turnover	Stores the turnover character

Data Class	**Method**	**Description**
	createFolder	Create a folder to store all the databases
	createTable	Create a daily settings table within the database
	currentDay	Finds the current day in order to retrieve the daily settings
	data	Creates a database object
	exportTable	Exports the database into the ‘Enigma settings’ folder for the user to import at a later stage
	genCharGroup	Creates a character group of 3 characters
	genRingSetting	Generates a random ring setting value between 0 and 25
	getPath	Identifies the path of the application to create the new folder
	insertData	Inserts the daily settings into the database
	ranCharGroup	Generates a random string of 4 character group for the kenngruppen
	ranPlugboard	Creates random pairs for the plugboard

	ranRingSetting	Creates a random string for the ring setting values
	ranRotorOrder	Selects three random rotors for the daily settings
	readData	Selects the daily settings record for the current day
	reset	Creates a new database connection
	Attribute	**Description**
	c	Stores the database cursor for data set traversal
	conn	Establishes connection to the database
	database	Stores the name of the database
	destination	Stores the export location for the database

Plugboard Class	**Method**	**Description**
	plugboard	Creates a plugboard object
	switchChar	Substitutes the input with the character it's connected to
	Attribute	**Description**
	connections	Stores the plugboard connection pairs

Reflector Class	**Method**	**Description**
	reflect	Substitutes the input with its paired character
	reflector	Creates a reflector object
	Attribute	**Description**
	connections	Stores the reflector pairs

Each class will now be described in more detail explaining the algorithms and data structures behind them. For a more thorough understanding of how the enigma machine works, please refer to **Appendix A**.

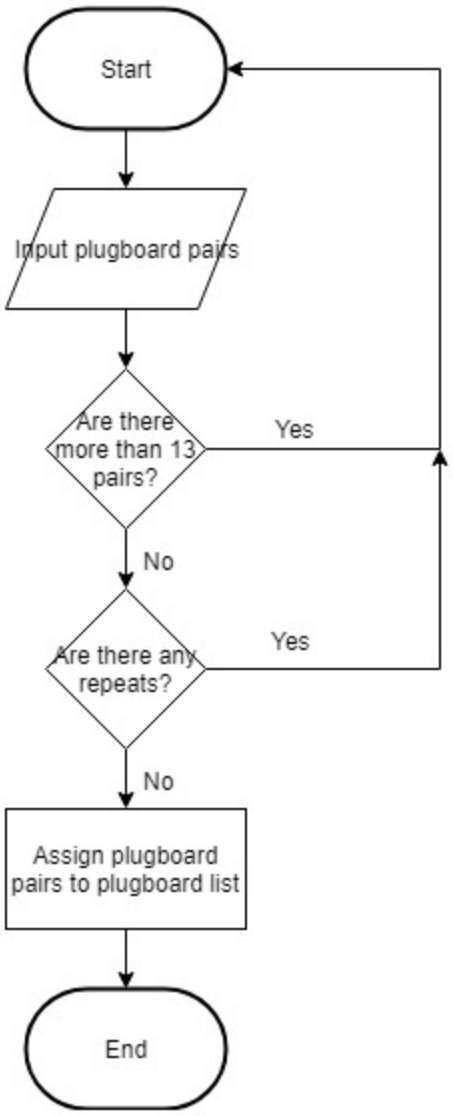
PLUGBOARD CLASS

The plugboard is the first function to be executed by the program. It takes an input character and produces a substituted value before and after encryption prior to being output to the user.

From our primary objectives, the plugboard class must:

- Allow the user to create paired connections between each character (MO 2.3.1)
- Output an error message if the user connects to the same character (MO 2.3.1.1)

SUBSTITUTION ALGORITHM



Validation

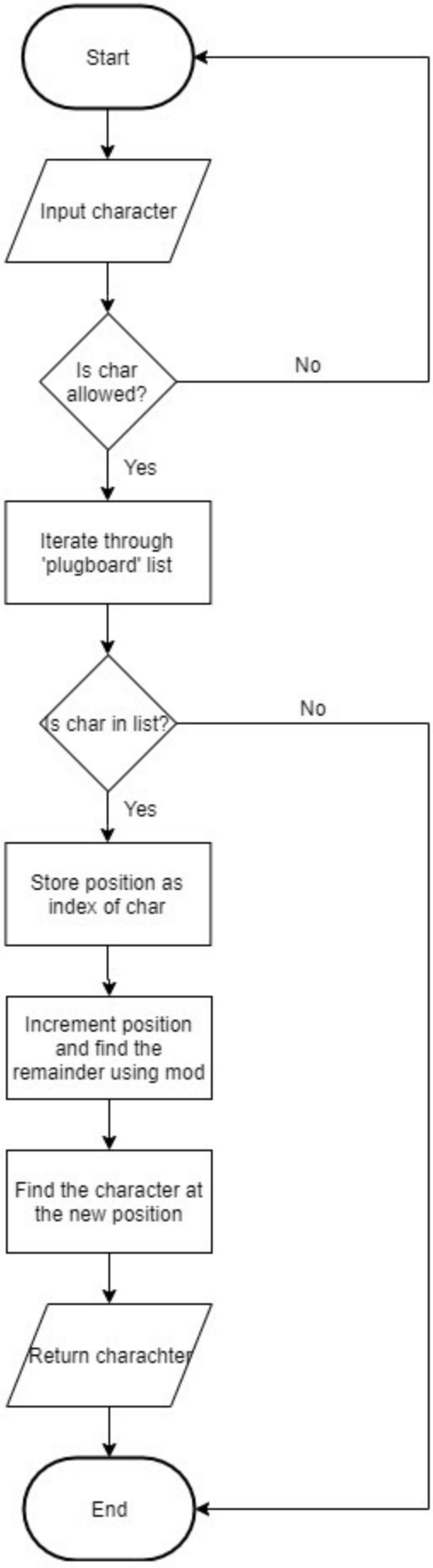
In order so that the user enters the input in the correct format, we must validate their input and prompt them to re-enter data entered in an incorrect format. Assuming the user connects all the plugs, there will be a maximum of 13 connections, therefore, if there are more than 13 pairs of characters or repeating characters, an error message should be displayed.

We can check these conditions after the user has entered a string input of pairs. This input can then be converted to a list type using the split() method which will afford us the ability to iterate through the list at a later stage.

Substitution

To allow the plugboard class to substitute input, we can create a method. When the plugboard receives input, the input must be validated to ensure it is within the alphabet. After doing so, the method can refer back to the plugboard list attribute and look for the following character in the pair by iterating through each of them. By finding the index of the input character and incrementing it by one, we can apply mod of the length of each pair, which is 2, allowing us to use the remainder to find the output value.

If a user has not used all 13 connections, the algorithm will recognise this and return the input character as it is not substituted. A description of the methods operation is depicted in the flowchart below.


Start
Input character
Is char allowed?
No
Yes
Iterate through 'plugboard' list
Is char in list?
No
Yes
Store position as index of char
Increment position and find the remainder using mod
Find the character at the new position
Return charachter
End

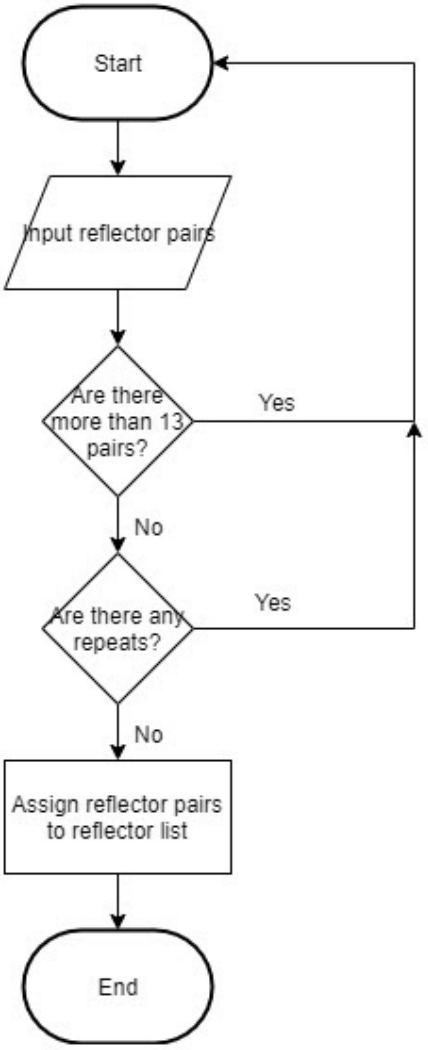
REFLECTOR CLASS

The reflector can be modelled in a similar fashion to the plugboard as they both essentially substitute input. The reflector takes the character travelling from the external contact and through the rotors as the input and substitutes it, returning the output through the rotors and out to the bulb.

From our primary objectives, the reflector class must:

- Substitute the input value and produce an output (MO 2.2.1)
- Allow the reflector type to be switched between A, B and C (MO 2.2.2, 2.2.3)

SUBSTITUTION ALGORITHM



Validation

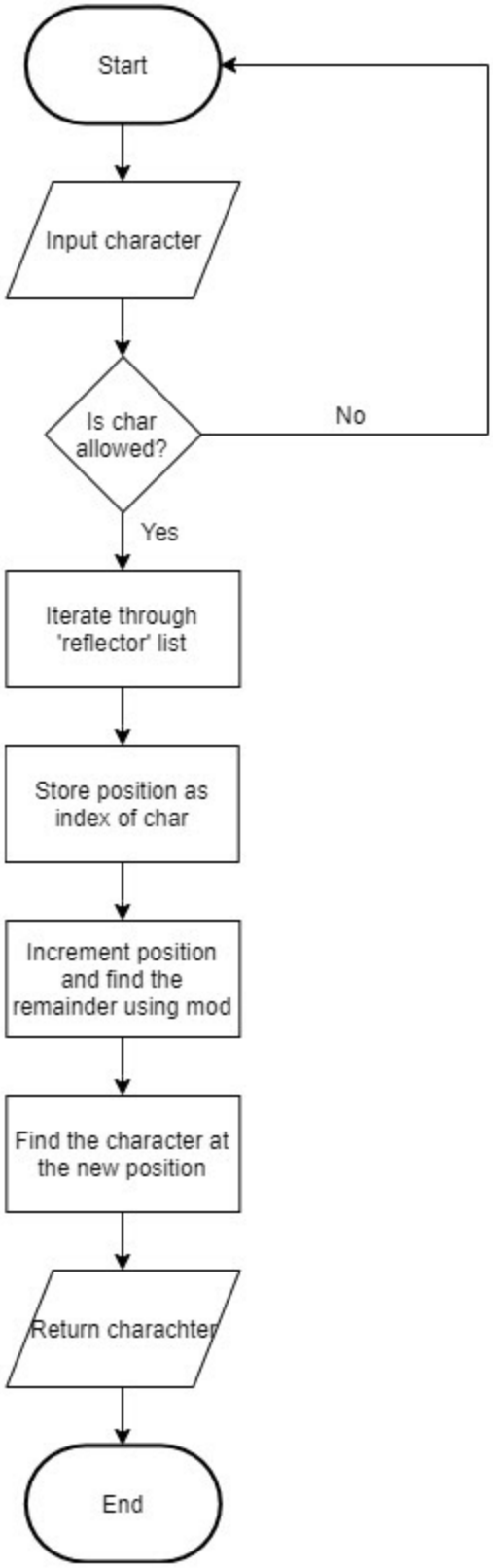
The reflector differs slightly from the plugboard as it must always have 13 connections. Therefore, we must ensure that there are 13 input pairs and no repeated characters so that each letter in the alphabet is paired up.

After the user has been prompted to enter a string input of pairs for the reflector, the input is converted to a list type using the split() method which will afford us the ability to iterate through the list in a similar fashion to the plugboard.

Substitution

To allow the reflector to substitute input, we must first validate the input to ensure it is within the alphabet. After doing so, we can iterate through the reflector pairs until we have located the position of the input character. By incrementing this value by one, we can find the position of the following value. In order so that the algorithm loops back to the beginning if the input character is the last value in the pair, we can use the mod of this sum and find the remainder which will give us the position of the first character.

A visual representation is provided in the flowchart below.


Start
Input character
Is char allowed?
No
Yes
Iterate through 'reflector' list
Store position as index of char
Increment position and find the remainder using mod
Find the character at the new position
Return charachter
End

ROTOR CLASS

The rotors of the enigma machine connect to the internal wiring of the enigma machine. Input passes through these connections between each of the rotors as it is ciphered though each rotor. The rotors rotate to increase the security of the system by encrypting each input with different characters during each rotation as the internal wiring rematches with the external rotor.

From our primary objectives, the rotors class must:

- Allow rotors to rotate (MO 2.2.1)
- Step each rotor from their respective values (MO 2.2.2)
- Switch between the rotors and display an error message if the same rotor type is used (MO 2.2.3, 2.2.3.1)
- Allow the user of 3 or 4 rotors (Extra MO 2.2.4)
- Match the external alphabet to the internal wiring (MO 2.2.5)
- Adjust the external alphabet and internal wiring as the rotors rotate (MO 2.2.6)
- Recognise the occurrence of single and double step sequences (MO 2.2.7)

SUBSTITUTION ALGORITHM

Character permutations

In order so that we have a better understanding of the workings of the enigma, we need to understand how input is encrypted. Using the default enigma settings with rotor I at position A, the permutation of the initial input character can be given using the cyclical representation below:

(AELTPHQXRU) (BKNW) (CMOY) (DFG) (IV) (JZ) (S)

Using the above permutations, we can create an algorithm to encode input characters as each character is encoded as the subsequent value within each individual array.

```
rotor ← (('AELTPHQXRU'), ('BKNW'), ('CMOY'), ('DFG'), ('IV'), ('JZ'), ('S'))
char ← USERINPUT

FOR sequence IN rotor
        FOR letter IN sequence
                IF letter = char
                        index ← POSITION(sequence, letter)
                        index += 1
                        sub ← sequence(index MOD LEN(sequence))
                        OUTPUT sub
                ENDIF
        ENDFOR
ENDFOR
```

In the code above, the rotor permutations are defined in a list. The block of code that encodes the input character sequentially compares each character to the input. Upon finding a match, the function finds the index of the character and increments it by one. This is so that it can identify which character must be substituted in place of the input. When you reach the end of an array, you will need to use the modulus function to calculate the remainder so that the algorithm is able to loop back to the beginning of the array and find the following character to substitute.

Considering the function has 2 nested loops, its time complexity would be n^2 indicating that with a larger set, the time taken to compare the character and input would increase polynomially. However, this is an overestimate as some of the arrays in the list have only one value. Regardless, it would not be practical to use this method to calculate how each input is ciphered as you must determine all the permutations beforehand which would be an extremely tedious task.

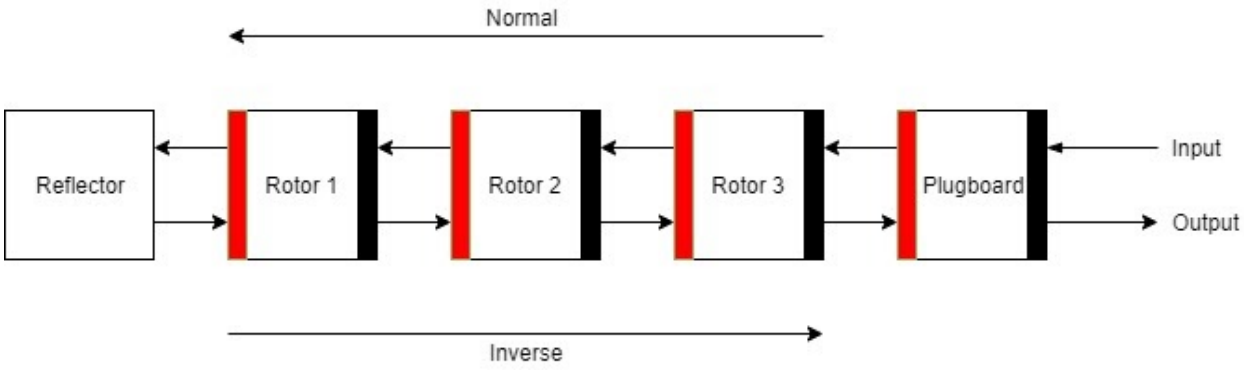
Substitution algorithm

We can develop on the above model using the Enigma I character table so that each of the input characters is paired with their substitute character in a similar fashion to the plugboard using the format (character, substitution). As the character is at the index[0] position, we can use the subroutine encode() below to substitute it with the substitution at index[1]. Again, this algorithm uses the modulus function to calculate the remainder of 'index MOD length' determining the position of the following value.

```
SUBROUTINE
encode(rotor, char)
        FOR pair IN rotor
                letter ← pair[0]
                IF letter = char
                        index = POSITION(pair,
        letter)
                        index += 1
                        sub ← pair(index MOD
        LEN(pair))
                        OUTPUT sub
                ENDIF
        ENDFOR
ENDSUBROUTINE
```

```
SUBROUTINE
encodeInverse(rotor, char)
        FOR pair IN rotor
                letter ← pair[1]
                IF letter = char
                        index = POSITION(pair,
        letter)
                        index += 1
                        sub ← pair(index MOD
        LEN(pair))
                        OUTPUT sub
                ENDIF
        ENDFOR
ENDSUBROUTINE
```

After the character has been reflected, it must go through the rotors again but now it is the inverse of the encode function as it travels through the rotors in the reverse order. The diagram below describes this operation. This means that we now have to retrieve the substitution at the index[1] position and use that to find the following character as demonstrated in the subroutine encodeInverse() above.



The table below shows the test data for the encryption of the first input character . The first row shows the input data and the last shows the output data. As you can see in the table, the symmetric nature of the enigma is evident as each output entered as an input returns the same value as the initial input. Encoding the character ‘A’ through rotors, I II III, AAZ, would produce ‘U’ as the output and vice-versa.

Input	A	U	Q	V
Rotor3	B	K	I	M
Rotor 2	J	L	X	W
Rotor 1	Z	T	R	B
Reflector B	T	Z	B	R
Rotor 1	L	J	W	X
Rotor 2	K	B	M	I
Rotor 3	U	A	V	Q

```
>>> encode(rotor3, 'A')
B
>>> encode(rotor2, 'B')
J
>>> encode(rotor1, 'J')
Z
>>> reflector(reflectorB, 'Z')
T
>>> encodeInverse(rotor1, 'T')
L
>>> encodeInverse(rotor2, 'L')
K
>>> encodeInverse(rotor3, 'K')
U
>>>
```

```
>>> encode(rotor3, 'U')
K
>>> encode(rotor2, 'K')
L
>>> encode(rotor1, 'L')
T
>>> reflector(reflectorB, 'T')
Z
>>> encodeInverse(rotor1, 'Z')
J
>>> encodeInverse(rotor2, 'J')
B
>>> encodeInverse(rotor3, 'B')
A
>>>
```

Testing these functions, we can see that our results are similar to the expected data indicating that we have successfully replicated the cipher of the enigma machine. However, this model does not account for the rotation of the rotor when a key is pressed on the enigma machine. To account for this, we must adapt the code so that the values increment by one and so that the next rotor increments by one when the turnover character is reached.

ROTATION ALGORITHM

Mapping internal contacts to external values on the rotor

As we have modelled the rotors as a class, we can define a position attribute to keep track of the position of the external rotor. In order to increment the rotor, we only need to add one to the current position. However, as the rotors rotate, the internal wiring must be rematched to each external value on the rotor again.

To do this, we must create algorithms to carry out the following operations:

- Increment the position attribute
- Find the internal wiring contact for each external rotor value
- Encode an input by substituting with the internal wiring contact depending on the external rotor value

As the inner wiring of the rotor is input by the user as a string, our algorithm must match the inner wiring to the external numerical values on the rotor. We can map this relationship as a dictionary. The following algorithm converts the string into a list and sequentially assigns a numerical value to each character based on it's position in relation to the external rotor position.

E.g.

alphabet = EKMFLGDQVZNTOWYHXUSPAIBRCJ

For example, the above alphabet is from rotor I. This means that when the rotor is in the 01 position, the internal wiring is connected to the letter ‘E’, therefore, encoding an input of ‘A’, the first character, as ‘E’.

```
SUBROUTINE
GetAlphabet()
        alphabet ← LIST(externalRotorAlphabet)
        numbers ← [i FOR i IN RANGE(0, 26)]
        numberOff ← DICT( ZIP(alphabet, numbers))

        RETURN numberOff
ENDSUBROUTINE
```

The ‘GetAlphabet()’ method cerates a list of the external rotors alphabet and another list of numbers from 0 to 25 as the index positions of the alphabet characters is one less than their position in the alphabet. A ‘numberOff’ dictionary is then created where the alphabet is matched to the chronological position values creating a dictionary where each character is the key and the value is their position in the rotors alphabet.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 0, 0)
>>> rotor1.GetAlphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 10
'N': 11, 'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S': 19,
'P': 20, 'A': 21, 'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
```

Testing this, we can see that the algorithm is working perfectly. Now that we have mapped the rotor values to the internal wiring contacts, we need to create an algorithm to increment the values when the rotor rotates.

Retrieving the key and value

In order so that we can access the values of each element in the alphabet dictionary, we need to create another algorithm. The subroutines shown below iterate through the dictionary and obtain the key and value for each element in the dictionary, comparing it with the input character or number until they find a match. Then the complementary character or number value is returned to the user.

The ‘GetCharByNum()’ method finds the corresponding character of a number input and the ‘GetNumByChar()’ method finds the corresponding number of an input character.

```
SUBROUTINE
GetCharByNum(inputNum)
        FOR char, num IN GetAlphabet.items()
                IF num = inputNum THEN
                        RETURN char
                END IF
        END FOR
ENDSUBROUTINE
```

```
SUBROUTINE
GetNumByChar(inputChar)
        FOR char, num IN GetAlphabet.items()
                IF char = inputChar THEN
                        RETURN num
                END IF
        END FOR
ENDSUBROUTINE
```

Testing these functions, we can see that we are producing our desired results indicating that this method of using a dictionary works just as well as our previous method.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 0, 0)
>>> rotor1.GetAlphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 10
'N': 11, 'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S': 19,
'P': 20, 'A': 21, 'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
>>> rotor1.GetCharByNum(1)
E
>>> rotor1.GetNumByChar(1)
>>> rotor1.GetNumByChar('E')
1
>>>
```

Substituting from the rotor to the wiring

When a character is passed through the rotors to be ciphered, the same encryption process is repeated for each rotor. Below are algorithms detailing the process of encrypting to and from the reflector.

```
SUBROUTINE
cipherToReflector(char)
        inputCharNum ← ordChar(char)
        outputChar ←
GetCharByNum(inputCharNum)
        RETURN outputChar
ENDSUBROUTINE
```

```
SUBROUTINE
cipherFromReflector(char)
        inputCharNum ←
GetNumByChar(char)
        outputChar ← chrNum(inputCharNum)
        RETURN outputChar
ENDSUBROUTINE
```

The cipherToReflector() method above takes an input string value and converts the value to its equivalent position value in the alphabet. This is done using ord() which finds the ASCII value of a string. As the letter A is 65 in ASCII, we subtract 64 from the ord() value so that the range of all the characters are 1 to 26, taking their position in the alphabet. This value is then used to find the letter in the equivalent position of the rotor alphabet. For example, input ‘Q’ would produce 17 as

its position in the alphabet and the corresponding value of the rotor alphabet would be ‘X’, returning this as the output value. The inverse is done for the cipherFromReflector method.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor1._position
1
>>> rotor1.cipherInputVal('A')
'E'
>>> rotor1.cipherInputVal('Q')
'X'
>>>
```

cipherInputVal() and cipherOutputVal() have since been changed to cipherToReflector and cipherFromReflector respectively for better clarity as more complex methods have been introduced.

Testing this method successfully encodes the input character using the connections of the external rotor to the internal wiring contact.

Rotating the rotors

To simulate the anticlockwise rotation of the rotors, we can break the process into two stages, shifting the rotor anticlockwise and then changing the key to the previous letter. To further understand this process, you should imagine a column of numbers representing the external rotor and a column of the corresponding values in each position on the internal wiring. Using the table below, we can get a clearer understanding of this process using the rotor I.

Starting positions of internal wiring and rotor		**Stage 1: Shift the rotor anticlockwise**		**Stage 2: Change to previous letter**	
01 A	E	**26 Z**	E	**26 Z**	D
02 B	K	**01 A**	K	**01 A**	J
03 C	M	**02 B**	M	**02 B**	L
04 D	F	**03 C**	F	**03 C**	E

You should note that the internal wiring, the letters in the right column, do not change, rather it is the external rotor, in the left column, that rotates. The rotor initially starts with the external rotor corresponding to the internal wiring so that the first characters of each alphabet are lined up. When a key is pushed, the rotor increments and the rotor shifts anticlockwise followed by the character changing to its predecessor in the chronological alphabet.

The first part of this algorithm, the shift stage is modelled below.

```
SUBROUTINE
incrementRotor()
        position += 1
        position ← position MOD 26

        FOR k, v IN alphabet.items()
                alphabet[k] ← v - 1 MOD 26
        ENDFOR
ENDSUBROUTINE
```

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor1.alphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 10, 'N': 11,
'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S': 19, 'P': 20, 'A': 21,
'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
>>> rotor1.incrementRotor()
>>> rotor1.alphabet
{'E': 26, 'K': 1, 'M': 2, 'F': 3, 'L': 4, 'G': 5, 'D': 6, 'Q': 7, 'V': 8, 'Z': 9, 'N': 10,
'T': 11, 'O': 12, 'W': 13, 'Y': 14, 'H': 15, 'X': 16, 'U': 17, 'S': 18, 'P': 19, 'A': 20,
'I': 21, 'B': 22, 'R': 23, 'C': 24, 'J': 25}
>>>
```

Testing the method produces our desired results as the method simulates the rotor rotating anticlockwise by one position. The rotor now shows the corresponding value of the internal wiring and the rotor positions. To implement the second stage where the letters themselves are assigned to the preceding letter, we must decrease the letter value by one.

However, we have run into a problem as keys cannot be changed in python. Instead, we will have to turn the keys into values and the values into keys so that we can momentarily change the characters. This will mean that the format {‘Key’: value} will change to {value: ‘Key’} allowing us to append the key values.

We will be able to find the preceding character values by subtracting one after we have used the GetNumByChar() method. We can then convert this numerical value back to its string equivalent using GetCharByNum() and then reversing the position of the {value: ‘Key’} back to {‘Key’: value} with our newly updated key values.

The following method actualizes this concept.

```
SUBROUTINE
incrementRotor()
```

```
        position += 1
        position ← position MOD 26

        FOR k, v IN alphabet.items()
                alphabet[k] ← v - 1 MOD 26
        ENDFOR

        tempAlphabet ← {}
        FOR k, v in alphabet.items()
                tempAlphabet[v] ← k
        ENDFOR
        alphabet ← tempAlphabet

        FOR k, v in alphabet.items()
                charNum ← ordChar(v)
                prevCharNum ← (charNum - 1) MOD 26
                prevChar = chrNum(prevCharNum)
                alphabet[k] = prevChar
        ENDFOR

        tempAlphabet ← {}
        FOR k, v in alphabet.items()
                tempAlphabet[v] ← k
        ENDFOR
        alphabet ← tempAlphabet
ENDSUBROUTINE
```

Breaking down the pseudocode above, we can see that the incrementRotor() function begins by incrementing the position attribute upon a keypress on the enigma. Whilst pressure is applied to the key, a circuit is created and current flows through the enigma. This method simulates the flow of current through the first rotor. The first iteration simulates the anticlockwise shift of the rotor by decreasing the rotor value by one so that the rotor is matched to the following character in the internal wiring.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor1.alphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 1
0, 'N': 11, 'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S':
19, 'P': 20, 'A': 21, 'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
>>> rotor1.incrementRotor()
>>> rotor1.alphabet
{'E': 26, 'K': 1, 'M': 2, 'F': 3, 'L': 4, 'G': 5, 'D': 6, 'Q': 7, 'V': 8, 'Z':
9, 'N': 10, 'T': 11, 'O': 12, 'W': 13, 'Y': 14, 'H': 15, 'X': 16, 'U': 17, 'S':
18, 'P': 19, 'A': 20, 'I': 21, 'B': 22, 'R': 23, 'C': 24, 'J': 25}
>>>
```

The second iteration creates a new list, tempAlphabet and reverses the position of the keys and values. This new list is then assigned to the original alphabet attribute so that it now produces a list of reversed values which have rotated by one position.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor1.alphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 1
0, 'N': 11, 'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S':
19, 'P': 20, 'A': 21, 'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
>>> rotor1.incrementRotor()
{26: 'E', 1: 'K', 2: 'M', 3: 'F', 4: 'L', 5: 'G', 6: 'D', 7: 'Q', 8: 'V', 9: 'Z
', 10: 'N', 11: 'T', 12: 'O', 13: 'W', 14: 'Y', 15: 'H', 16: 'X', 17: 'U', 18:
'S', 19: 'P', 20: 'A', 21: 'I', 22: 'B', 23: 'R', 24: 'C', 25: 'J'}
>>>
```

The third iteration then finds the integer value of each key and stores them in the integer variable, charNum. This value is then decreased by one position and converted back to its string equivalent, assigning each character in the alphabet attribute to its preceding letter.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor1.alphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 1
0, 'N': 11, 'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S':
19, 'P': 20, 'A': 21, 'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
>>> rotor1.incrementRotor()
>>> rotor1.alphabet
{26: 'D', 1: 'J', 2: 'L', 3: 'E', 4: 'K', 5: 'F', 6: 'C', 7: 'P', 8: 'U', 9: 'Y
', 10: 'M', 11: 'S', 12: 'N', 13: 'V', 14: 'X', 15: 'G', 16: 'W', 17: 'T', 18:
'R', 19: 'O', 20: 'Z', 21: 'H', 22: 'A', 23: 'Q', 24: 'B', 25: 'I'}
>>>
```

Finally, the rotor then reverses the position of the key and value to the original format of {'letter': rotor position}.

```
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor1.alphabet
{'E': 1, 'K': 2, 'M': 3, 'F': 4, 'L': 5, 'G': 6, 'D': 7, 'Q': 8, 'V': 9, 'Z': 1
0, 'N': 11, 'T': 12, 'O': 13, 'W': 14, 'Y': 15, 'H': 16, 'X': 17, 'U': 18, 'S':
19, 'P': 20, 'A': 21, 'I': 22, 'B': 23, 'R': 24, 'C': 25, 'J': 26}
>>> rotor1.incrementRotor()
>>> rotor1.alphabet
{'D': 26, 'J': 1, 'L': 2, 'E': 3, 'K': 4, 'F': 5, 'C': 6, 'P': 7, 'U': 8, 'Y':
9, 'M': 10, 'S': 11, 'N': 12, 'V': 13, 'X': 14, 'G': 15, 'W': 16, 'T': 17, 'R':
18, 'O': 19, 'Z': 20, 'H': 21, 'A': 22, 'Q': 23, 'B': 24, 'I': 25}
>>>
```

Testing the algorithms above produces the intended results showing that this is an accurate simulation of the rotation mechanism of the enigma machine

Ciphering the initial user input

Ciphering the initial user input from the keys of the enigma machine differs from the normal function of the rotors as the rotor increments before ciphering the character. To acknowledge this, we must call the incrementRotor() algorithm as soon as the user inputs a key. This is because the input is communicated when the keys on the enigma machine are pressed and is not needed for the output as the encrypted letter is output to a bulb.

The following algorithm describes this process.

```
SUBROUTINE
cipherFromExternalContact(char)
        incrementRotor()
        char ← cipherToReflector(char)
        RETURN char
END SUBROUTINE
```

Below is a table of test values for our new method. The table shows the expected values for a machine in the position, I II III, using a UKW-B reflector in the position of AAA, with a ring setting of AAA. As the rightmost rotor is in position A, when an input is received, the rotor will increment and encode using the rotor positions AAB. This means that the rightmost rotor, III, will experience a shift.

	From position AAA → AAB		From position AAB → AAC	
Input	A	B	A	D
Rotor3	C	E	D	J
Rotor 2	D	S	K	B
Rotor 1	F	S	N	K
Reflector B	S	F	K	N
Rotor 1	S	D	B	K
Rotor 2	E	C	J	D
Rotor 3	B	A	D	A

Below are the results from testing the different components of the encryption process. As you can see, the program has accurately reproduced the test data of the rotor shift and cipher process. Encrypting from AAA → AAB, produces 'A' → 'B' and encrypting from AAB → AAC, produces 'A' → 'D'. Resetting the rotors and encrypting the input` again produces 'B' → 'A' from AAA → AAB and 'D' → 'A' from AAB → AAC.

```
>>> rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 1, 1)
>>> rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 1, 1)
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor3.cipherFromExternalContact('A')
'C'
>>> rotor2.cipherToReflector('C')
'D'
>>> rotor1.cipherToReflector('D')
'F'
>>> reflector(UKWB, 'F')
S
>>> rotor1.cipherFromReflector('S')
'S'
>>> rotor2.cipherFromReflector('S')
'E'
>>> rotor3.cipherFromReflector('E')
'B'
>>> rotor3.cipherFromExternalContact('A')
'D'
>>> rotor2.cipherToReflector('D')
'K'
>>> rotor1.cipherToReflector('K')
'N'
>>> reflector(UKWB, 'N')
K
>>> rotor1.cipherFromReflector('K')
'B'
>>> rotor2.cipherFromReflector('B')
'J'
>>> rotor3.cipherFromReflector('J')
'D'
>>>
```

```
>>> rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 1, 1)
>>> rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 1, 1)
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
>>> rotor3.cipherFromExternalContact('B')
'E'
>>> rotor2.cipherToReflector('E')
'S'
>>> rotor1.cipherToReflector('S')
'S'
>>> reflector(UKWB, 'S')
F
>>> rotor1.cipherFromReflector('F')
'D'
>>> rotor2.cipherFromReflector('D')
'C'
>>> rotor3.cipherFromReflector('C')
'A'
>>> rotor3.cipherFromExternalContact('D')
'J'
>>> rotor2.cipherToReflector('J')
'B'
>>> rotor1.cipherToReflector('B')
'K'
>>> reflector(UKWB, 'K')
N
>>> rotor1.cipherFromReflector('N')
'K'
>>> rotor2.cipherFromReflector('K')
'D'
>>> rotor3.cipherFromReflector('D')
'A'
>>>
```

Now that we have fully functioning rotating rotors, we must adjust our code to accommodate the turnover and double stepping of the rotors when each rotor reaches its turnover notch.

ADJUSTMENT ALGORITHM

Currently, our rotor assumes the start position of all the rotors is at the default 01 position, however in reality, this was not the case and so we must adapt our code to allow the rotors to start at different positions.

When a rotor is initialised, the GetAlphabet() method assigns each character on the rotor a numerical value based on its position in the rotor character sequence. We can adjust these values by simply incrementing the rotor until it reaches that value on the rotor.

```
SUBROUTINE
adjustAlphabetByposition
        For i IN RANGE(0, position)
                adjustRotor
        END FOR
```

```
END SUBROUTINE
```

The above method does just that and iterates through the adjustRotor() method which consists of the same code as the incrementRotor() method but does not increment the position attribute. This is because the rotor position is initialised at the 01 position and must iterate to reach the input position without actually incrementing any of the rotors as it is the starting position. Another reason why I have had to make a separate method is because the incrementRotor() method will need to include additional code to account for rotor turnover. However, the rotor does not need to turnover when its position is being adjusted as each rotors initial position is independent of the other rotors.

The algorithm below is identical to the 'incrementRotor()' method only differing as it does not increment the rotor.

```
SUBROUTINE
adjustRotor()
        FOR k,v IN alphabet.items()
                alphabet[k] ← (v - 1) MOD 26
        ENDFOR

        tempAlphabet ← {}
        FOR k,v IN alphabet.items()
                tempAlphabet[v] ← k
        ENDFOR
        alphabet = tempAlphabet

        FOR k,v IN alphabet.items()
                charNum ← ordChar(v)
                prevCharNum ← (charNum - 1) MOD 26
                prevChar ← chrNum(prevCharNum)
                alphabet[k] ← prevChar
        ENDFOR

        tempAlphabet ← {}
        FOR k,v IN alphabet.items()
                tempAlphabet[v] ← k
        ENDFOR
        alphabet ← tempAlphabet
ENDSUBROUTINE
```

Below are the expected results when encoding A from rotor positions BBB → BBC and YYY → YYZ on an enigma machine. I have chosen to test the YYY position as it is past the turnover values which will support my thesis that the initial rotor position is independent of the other rotors and so turnover is not accounted for.

	From position BBB → BBC		From position YYY → YYZ	
Input	A	P	A	H
Rotor3	D	U	P	D
Rotor 2	R	X	V	L
Rotor 1	R	B	R	B
Reflector B	B	R	B	R
Rotor 1	X	R	L	V
Rotor 2	U	D	D	P
Rotor 3	P	A	H	A

From our results below, we can see that my thesis was correct and that the initial rotor positions are not affected by the turnover notch of each position. As such, I will keep the adjustRotor() and incrementRotor() functions as separate methods. This will also allow me to include code to deal with turnover in the incrementRotor() method.

Ciphering input A to output P

```
>>> rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 2, 1)
>>> rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 2, 1)
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 2, 1)
>>> rotor3.cipherFromExternalContact('A')
'D'
>>> rotor2.cipherToReflector('D')
'R'
>>> rotor1.cipherToReflector('R')
'R'
>>> reflector(UKWB, 'R')
'B'
>>> rotor1.cipherFromReflector('B')
'X'
>>> rotor2.cipherFromReflector('X')
'U'
>>> rotor3.cipherFromReflector('U')
'P'
>>>
```

Ciphering input P to output A

```
>>> rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 2, 1)
>>> rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 2, 1)
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 2, 1)
>>> rotor3.cipherFromExternalContact('P')
'U'
>>> rotor2.cipherToReflector('U')
'X'
>>> rotor1.cipherToReflector('X')
'B'
>>> reflector(UKWB, 'B')
'R'
>>> rotor1.cipherFromReflector('R')
'R'
>>> rotor2.cipherFromReflector('R')
'D'
>>> rotor3.cipherFromReflector('D')
'A'
>>>
```

Ciphering input A to output H

```
>>> rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 25, 1)
>>> rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 25, 1)
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 25, 1)
>>> rotor3.cipherFromExternalContact('A')
'P'
>>> rotor2.cipherToReflector('P')
'V'
>>> rotor1.cipherToReflector('V')
'R'
>>> reflector(UKWB, 'R')
'B'
>>> rotor1.cipherFromReflector('B')
'L'
>>> rotor2.cipherFromReflector('L')
'D'
>>> rotor3.cipherFromReflector('D')
'H'
>>>
```

Ciphering input H to output A

```
>>> rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 25, 1)
>>> rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 25, 1)
>>> rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 25, 1)
>>> rotor3.cipherFromExternalContact('H')
'D'
>>> rotor2.cipherToReflector('D')
'L'
>>> rotor1.cipherToReflector('L')
'B'
>>> reflector(UKWB, 'B')
'R'
>>> rotor1.cipherFromReflector('R')
'V'
>>> rotor2.cipherFromReflector('V')
'P'
>>> rotor3.cipherFromReflector('P')
'A'
>>>
```

TURNOVER ALGORITHM

Turnover occurs when a turnover value is reached by each rotor which causes the next rotor to increment by one. There are two instances where turnover occurs in a single step sequence and in a double step sequence.

A single step sequence occurs when the rightmost rotor reaches its turnover position. Once the right rotor is in the turnover position, a single step sequence is now imminent and the algorithm will increment the following middle rotor upon the next keypress of the enigma as the right rotor steps with the middle rotor.

Rotor position	Description	Single step
AAU	Normal step of right rotor	False
AAV	Right rotor in the turnover position	True
ABW	Right rotor steps with middle rotor	False
ABX	Normal step of right rotor	False

A double step sequence occurs when the step of the right rotor, from its turnover position, causes the following middle rotor to increment to its turnover position. Then as the middle rotor is in the turnover position, the next incrementation of the right rotor will increment both the right and middle rotors causing all three rotors to turn together.

Rotor position	Description	Single step	Double step
ADU	Normal step of right rotor	False	False
ADV	Right rotor in the turnover position	True	False
AEW	Right rotor steps with middle rotor, Middle rotor in turnover position	False	True
BFX	Middle rotor double steps	False	False
BFY	Normal step of right rotor	False	False

We can identify when turnover will occur by using an algorithm to check the state of each rotor throughout each encryption of the input character between the 3 rotors of the normal enigma. We can create the methods singleStep and doubleStep to do this.

The algorithm ‘singleStep()’ checks for a single step sequence by retrieving the current position and the turnover position of the rightmost rotor. As the rightmost rotor is not static and increments before a character is encrypted through it, we must increase the turnover value by one so that it steps from that transition. The algorithm then compares the current position of the rotor and the turnover position and upon finding that they are a match, it increments the following middle rotor.

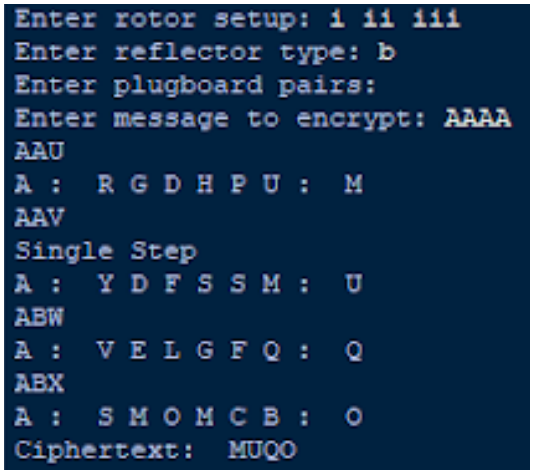
```
SUBROUTINE
singleStep(rotorL, rotorM, rotorR)
        Rpos ← rotorR.GetRotorPosition()
        Rturn ← rotorR.GetTurnoverNum()

        Rturn +=1
        IF Rpos = Rturn THEN
                rotorM.incrementRotor()
        ENDIF
ENDSUBROUTINE
```

```
SUBROUTINE
doubleStep(rotorL, rotorM, rotorR)
        Mpos ← rotorM.GetRotorPosition()
        Mturn ← rotorM.GetTurnoverNum()

        IF Mpos = Mturn THEN
                rotorM.incrementRotor()
                rotorL.incrementRotor()
        ENDIF
ENDSUBROUTINE
```

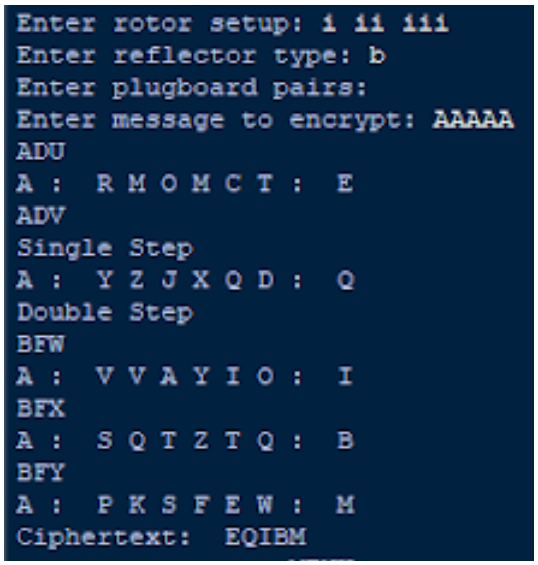
The doubleStep algorithm is similar in nature to the ‘singeStep()’ algorithm but differs as it checks the position and turnover of the static middle rotor. As the leftmost rotor does not increase with each keypress, instead incrementing upon the turnover value being reached by the middle rotor, the algorithm again compares the position and turnover of the middle rotor and if a match is found, both the middle and left rotor are incremented to simulate the double step sequence.

Testing these algorithms below finds that they work as intended and successfully emulate the turnover step sequences of the enigma machine. However, as the singleStep() and doubleStep() algorithms need to compare the individual rotor objects, it would not make sense to put them within the rotor class. Therefore, we will put them in the simulation class where the class will be able to evaluate the above algorithms.

Testing the single step transition from V to W on rotor III.



Testing the double step transition from ADV to BFW using rotors I II III.



DATABASE DESIGN

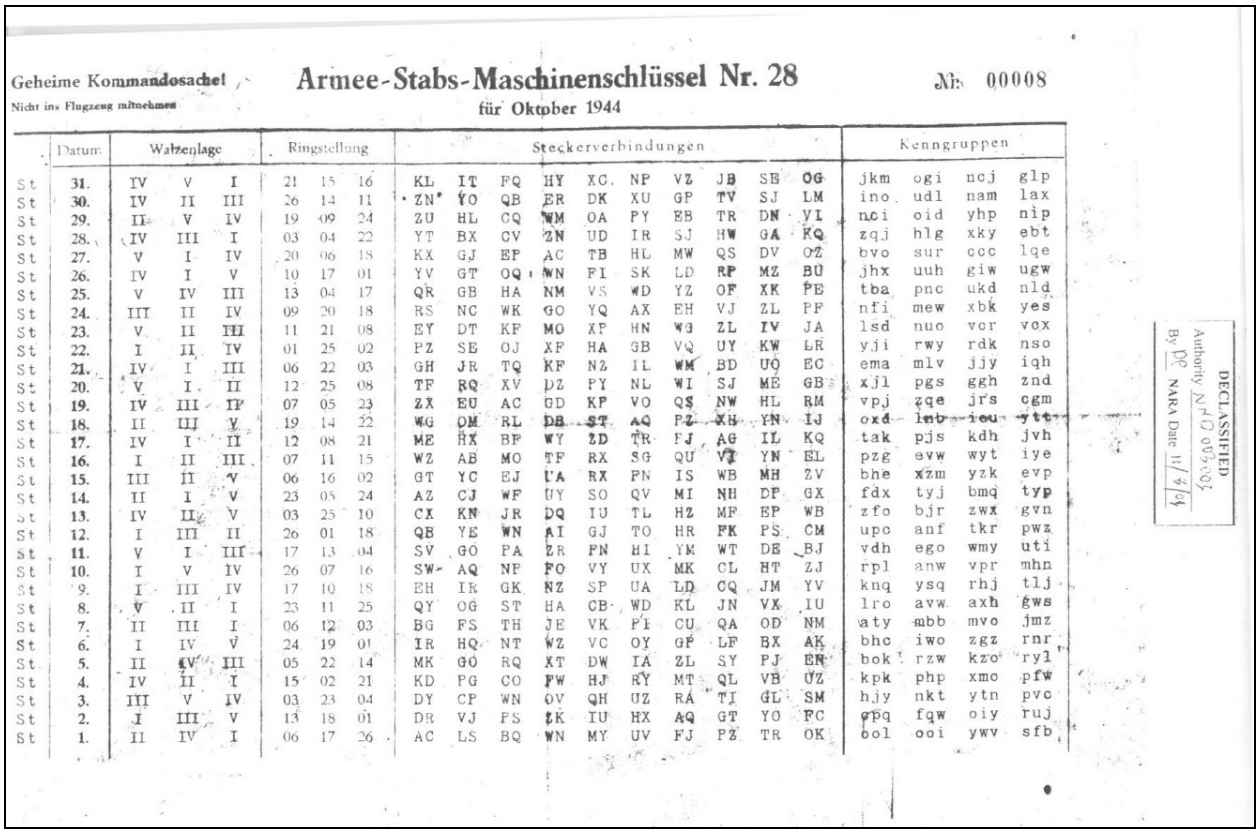
Now we will design a database to store the user settings of the enigma so that they can be imported and exported. The import and export process will be covered in more detail in the simulation class.

From our primary objectives, the application must:

- Generate cipher tables for the month (MO 1.4, 1.4.2.2)
- Export cipher tables (MO 1.4.2, 1.2.2.1)
- Import cipher tables and validate input (MO 1.5, 1.5.1, 1.5.1.1, 1.5.1.2)

EXPORTING DATABASE

Our database will be in the same format as the original cipher tables used by the Germans in WW2. Below is an example of one of these tables for the month of October in 1944:



Geheime Kommandosache!
Nicht ins Flugzeug mitnehmen

Armee-Stabs-Maschinenschlüssel Nr. 28
für Oktober 1944

Nr. 00008

	Datum	Walzenlage			Ringstellung			Steckerverbindungen										Kenngruppen			
St	31.	IV	V	I	21	15	16	KL	IT	FQ	HY	XC	NP	VZ	JB	SE	OG	jkm	ogi	ncj	glp
St	30.	IV	II	III	26	14	11	ZN	YO	QB	ER	DK	XU	GP	TV	SJ	LM	ino	udl	nam	lax
St	29.	II	V	IV	19	09	24	ZU	HL	CQ	WM	OA	PY	EB	TR	DN	VI	nci	oid	yhp	nip
St	28.	IV	III	I	03	04	22	YT	BX	CV	ZN	UD	IR	SJ	HW	GA	KQ	zqj	hlg	xky	ebt
St	27.	V	I	IV	20	06	18	KX	GJ	EP	AC	TB	HL	MW	QS	DV	OZ	bvo	sur	ccc	lqe
St	26.	IV	I	V	10	17	01	YV	GT	OQ	WN	FI	SK	LD	RP	MZ	BU	jhx	uuh	giw	ugw
St	25.	V	IV	III	13	04	17	QR	GB	HA	NM	VS	WD	YZ	OF	XK	PE	tba	pnc	ukd	nld
St	24.	III	II	IV	09	20	18	RS	NC	WK	GO	YQ	AX	EH	VJ	ZL	PF	nfi	mew	xbk	yes
St	23.	V	II	III	11	21	08	EY	DT	KF	MO	XP	HN	WG	ZL	IV	JA	lsd	nuo	vcr	vcx
St	22.	I	II	IV	01	25	02	PZ	SE	OJ	XF	HA	GB	VQ	UY	KW	LR	yji	rwy	rdk	nso
St	21.	IV	I	III	06	22	03	GH	JR	TQ	KF	NZ	IL	WM	BD	UO	EC	ema	mlv	jjy	iqh
St	20.	V	I	II	12	25	08	TF	RQ	XV	DZ	PY	NL	WI	SJ	ME	GB	xjl	pgs	ggh	znd
St	19.	IV	III	II	07	05	23	ZX	EU	AC	GD	KP	VO	QS	NW	HL	RM	vpj	zqe	jrs	cgm
St	18.	II	III	V	19	14	22	WG	OM	RL	DB	ST	AQ	PZ	XB	YN	IJ	oxd	lnb	icu	ytt
St	17.	IV	I	II	12	08	21	ME	HX	BF	WY	ZD	TR	FJ	AG	IL	KQ	tak	pjs	kdh	jvh
St	16.	I	II	III	07	11	15	WZ	AB	MO	TF	RX	SG	QU	VI	YN	EL	pzg	evw	wyt	iye
St	15.	III	II	V	06	16	02	GT	YC	EJ	UA	RX	PN	IS	WB	MH	ZV	bhe	xzm	yzk	evp
St	14.	II	I	V	23	05	24	AZ	CJ	WF	UY	SO	QV	MI	NH	DP	GX	fdx	tyj	bmq	typ
St	13.	IV	II	V	03	25	10	CX	KN	JR	DQ	IU	TL	HZ	MF	EP	WB	zfo	bjr	zwx	gvn
St	12.	I	III	II	26	01	18	QB	YE	WN	AI	GJ	TO	HR	FK	PS	CM	upc	anf	tkr	pwz
St	11.	V	I	III	17	13	04	SV	GO	PA	ZR	FN	HI	YM	WT	DE	BJ	vdh	ego	wmy	uti
St	10.	I	V	IV	26	07	16	SW	AQ	NF	FO	VY	UX	MK	CL	HT	ZJ	rpl	anw	vpr	mhn
St	9.	I	III	IV	17	10	18	EH	IR	GK	NZ	SP	UA	LD	CQ	JM	YV	knq	ysq	rhj	tlj
St	8.	V	II	I	23	11	25	QY	OG	ST	HA	CB	WD	KL	JN	VX	IU	lro	avw	axh	gws
St	7.	II	III	I	06	12	03	BG	FS	TH	JE	VK	PI	CU	QA	OD	NM	aty	mbb	mvo	jmz
St	6.	I	IV	V	24	19	01	IR	HQ	NT	WZ	VC	OY	GP	LF	BX	AK	bhc	iwo	zgz	rnr
St	5.	II	IV	III	05	22	14	MK	GO	RQ	XT	DW	IA	ZL	SY	PJ	EN	bok	rzw	kzo	ryl
St	4.	IV	II	I	15	02	21	KD	PG	CO	FW	HJ	RY	MT	QL	VB	UZ	kpk	php	xmo	pfw
St	3.	III	V	IV	03	23	04	DY	CP	WN	OV	QH	UZ	RA	TI	GL	SM	hjy	nkt	ytn	pvc
St	2.	I	III	V	13	18	01	DR	VJ	FS	EK	IU	HX	AQ	GT	YO	FC	qpq	fqw	oiy	ruj
St	1.	II	IV	I	06	17	26	AC	LS	BQ	WN	MY	UV	FJ	PZ	TR	OK	ool	ooi	ywv	sfb

DECLASSIFIED
Authority NND 963102
By DC NARA Date 11/9/04

4

[4] Rijmenants, Dirk, (2020), The Heer and Luftwaffe Procedures: http://users.telenet.be/d.rijmenants/en/enigmaproc.htm

For the database , we will need the following fields in our database:

- Datum - Date
- Walzenlage - Rotor order
- Ringstellung - Ring setting
- Steckerverbindungen - Plugboard connections
- Kenngruppen - Starting positions of the rotors.

We can model our database using an entity relationship diagram. However, is not necessary for our database because we only have a single table which can be modelled as Armee-Stabs-Maschinenschlüssel (Datum, Walzenlage, Ringstellung, Steckerverbindungen, Grundstellung).

To create the database, we will be using SQLite3. This is because SQLite3 is more advantageous than other relational database management systems for the purposes of this emulator as it is extremely portable and accessible. It's syntax is also shorter than if you were to use procedural code and it is significantly faster than a traditional file system. Another important advantage is the ability to select precise records as you require, only reading in or overwriting the specified data. This will allow us to select the settings for the current day when we develop an algorithm to import the settings.

In order to generate the values for each record, we need to create algorithms for each field. The 'datum' field can easily be generated by decreasing a variable starting from 30 after each entry. In order to get the current day, we can import the datetime module and use the following code below which retrieves the full date but only returns the day.

```
SUBROUTINE
currentDay()
        day ← datetime.datetime.today().day
        RETURN day
ENDSUBROUTINE
```

The 'Walzenlage' field can be generated by importing the random module to select a random rotor and remove it from a rotor list containing all the rotors. This value is then added to another list called rotor order and a count variable is incremented by one. After this count value is equal to 3, the algorithm concatenates the 3 rotor values in rotor order, separating them with a 'space' character and returns the rotor order value.

```
SUBROUTINE()
randomRotorOrder()
        count ← 0
        rotors ← ['I', 'II', 'III', 'IV', 'V']
        rotorOrder = []
        WHILE count != 3
                choice ← random.choice(rotors)
                rotorOrde.append(choice)
                rotors.remove(choice)
                count += 1
        ENDWHILE
        rotorOrder = ' '.join(rotorOrder)
ENDSUBROUTINE
```

The 'Ringstellung' field can be generated by choosing a random number between 0 and 25 as the alphabet is indexed with these start and end values. The subroutine below selects a random number using the random module and converts the number into a double digit format which is stored as a string, mimicking the format of these values in original cipher tables.

```
SUBROUTINE
generateRingSetting()
        num ← RANDOM_INT(0, 25)
        IF num < 10
                num ← INT_TO_STRING(num)
                num ← '0' + num
        ENDIF
        RETURN num
ENDSUBROUTINE
```

Using the above algorithm, we can create another algorithm which runs the code above three times and collates them to form the ring settings, returning them as a string.

```
SUBROUTINE
randomRingSetting()
```

```
        ringSetting ← generateRingSetting() + ' ' + generateRingSetting() + ' ' +
generateRingSetting()
        RETURN ringSetting
ENDSUBROUTINE
```

The 'Steckerverbindung' field can be generated using the algorithm below. It works by creating a list of all the letters in the alphabet and an empty plugboard list. Using the len() function, the algorithm checks if the alphabet list is empty. Whilst this list is not empty, it removes a random character and adds it to the plugboard list, generating a series of random characters. After all the characters have been removed from the alphabet list and its len() is equal to 0, the plugboard list is concatenated forming a single string. As the format of the cipher tables require the letters to be paired up, the concatenated string is then iterated through, splitting the string after every two characters to create a list. This list is then concatenated again separating each pair of characters with a 'space' character.

```
SUBROUTINE
randomPlugboard()
        alphabet ← LIST('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        plugboard ← []
        WHILE LEN(alphabet) != 0
                i = random.choice(alphabet)
                alphabet.remove(i)
                plugboard.append(i)
        ENDWHILE

        raw_plugboard ← ''.join(plugboard)
        split ← 2
        plugboard ← [(raw_plugboard[i:i+split]) FOR i IN RANGE(0, LEN(raw_plugboard), split)]

        RETURN ''.join(plugboard)
ENDSUBROUTINE
```

The 'kenngruppen' field can be generated by using a similar method to the plugboard. A list containing all the characters in the alphabet is created again with another empty group list. Using the random module to select 3 random characters and store them in the group list. We can join these values together and then return them as a string.

```
SUBROUTINE
generateCharGroup()
        alphabet ← LIST('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        group ← []
        count ← 0
        WHILE count != 3
                I ← random.choice(alphabet)
                alphabet.remove(i)
                group.append(i)
                count += 1
        ENDWHILE

        RETURN STR(''.join(group))
ENDSUBROUTINE
```

As we can now create a single group of characters, we can create another subroutine to generate a string of three character groups by concatenating each group to form an entry for the kenngruppen field.

```
SUBROUTINE
randomCharGroup()
        group ← self.generateCharGroup() + ' ' + self.generateCharGroup() + ' ' +
self.generateCharGroup()

        RETURN group
ENDSUBROUTINE
```

Now that we have the means to generate our input values, we can develop a database to utilise these functions. The algorithm below inserts data into a table using SQL queries along with python commands.

```
SUBROUTINE
insertData()
        Datum ← 31
```

```
        WHILE Datum != 0
                Walzenlage = self.randomRotorOrder()
                Ringstellung = self.randomRingSetting()
                Steckerverbindungen = self.randomPlugboard()
                Kenngruppen = self.randomCharGroup()
                self.c.execute("INSERT INTO Enigma(Datum, Walzenlage, Ringstellung,
Steckerverbindungen, Kenngruppen) VALUES (?, ?, ?, ?, ?)",
(int(Datum), str(Walzenlage), str(Ringstellung), str(Steckerverbindungen), str(Kenngruppen)))
                Datum -= 1
        ENDWHILE

        self.conn.commit()
ENDSUBROUTINE
```

The algorithm begins by assigning the values of the walzenlage, ringstellung, steckerverbindung and kenngruppen using their respective methods above. After these values are obtained, a WHILE loop is used to iterate an ‘INSERT INTO’ SQL command which inserts each of the values into their respective fields. After the record has been completed, the algorithm repeats, generating a new set of values at the end of each loop to be used for the next iteration. The loop begins at 31 and after each cycle, it is decreased by one. When the ‘datum’ identifier is finally equal to zero, the loop will end and the changes will be committed to the database.

Testing the database above, we are able to read the database using ‘DB Browser for SQLite’ and we can see that the following values are generated for each respective field.

	Datum	Walzenlage	Ringstellung	Steckerverbindungen	Kenngruppen
	Filter	Filter	Filter	Filter	Filter
1	1	V I IV	09 15 12	AE RP GN ZS KJ TU QC LW IB HO FV MY DX	brj fxr lho lvx
2	2	V III II	13 03 23	AO LC WR TY NU PD BX VZ MF IE SK GQ JH	ndp kyl elr mke
3	3	IV I III	14 18 17	OG TH YD AS NQ XI BU PL ZR EK CM JV WF	zst izy abo yex
4	4	V I IV	04 02 02	CY TA UW RI XP QN JD HO GE KM VS ZB FL	brn hwg uca epz
5	5	I IV V	15 02 19	ZS CR EA YB IF VJ NM TW HU OD XQ KP GL	jxh iza iqk tqk
6	6	IV V I	14 20 09	BG XV PW KQ HL FR OA NI SU EC ZD JM YT	cxj eqh gki azv
7	7	II V I	01 23 17	ZA WO GU SF PM JQ HD KR EY LI NB CV TX	pri uwe ixk yhn
8	8	V III II	21 10 18	PM KS TH QL JI RY WF XD EG NO BV ZC UA	wgk twl ygr vrg
9	9	IV II V	00 18 24	KQ LJ GP UO ZB NT RM VI DH EF SA YC XW	ptj qvw och diw
10	10	I IV II	07 05 16	TO MF VU YJ RI BE AH GK LD NW QC ZS XP	xaf dky mha xdu
11	11	V I III	10 06 04	UF RP NJ XC GL OY KW IZ SH BD ME QA TV	iov omj xei nyd
12	12	II V IV	15 06 21	RA JT WO EV QD FM SG BX PC IN HK UY LZ	nge qwl iuo dlq
13	13	III IV V	11 05 19	VO ZR NG SC BJ MI DF HL AQ YE WX UP KT	ihu nvk bpf hgf
14	14	IV III V	10 19 07	HR BC OX ZY DU AS MN QL JT WE IV GF KP	yns uth huf cuw
15	15	I III V	17 22 19	YN UX DQ CL OZ IB KT VE MJ RA WG HF PS	bvr gne wed bzj
16	16	I II III	16 23 24	ZO SX TW MK HV UI RL NJ AB FP DC YE QG	uzv etc vzt gfr
17	17	I V IV	16 04 15	BL OF WE JU VZ IN MA QS XD YT CH RP KG	tjd yws dov xph
18	18	I IV II	06 00 08	RM FY XN JB GZ UO PC HW KI SE TD LQ AV	muo gzk iku ozj
19	19	V IV I	12 05 23	TJ KU MC BD EN YX FA WQ OS IH PV LR GZ	miq tef ivo wgq
20	20	III I V	20 10 23	UC EW KZ GR IH QV NB AT LD MJ PO YF SX	nad gct sqf yxo
21	21	I IV III	15 15 23	YI TN WG QP ES HR LZ DU JO VC MF AK XB	xvf wku khe phx
22	22	IV II I	04 05 17	SH NR TQ GJ AL IX ZK WM PY CD BE UF VO	hjz olc vju rgy
23	23	IV I II	20 06 02	QK DH WT RZ CS JN XA GP MY IO EU LV BF	lak dzn sld sch

1 - 23 of 31 Go to: 1

FOLDER CREATION

One issue raised by the users was the fact that the databases were disorganised. To resolve this, I created a method for the program to create a folder to store all the databases called ‘Enigma Settings’.

The code below uses the ‘os’ module to identify the path of the enigma machine program. Once it has identified the path, it adds ‘Enigma Settings’ to the end of it so that this path can be used as the destination of the new folder it will create.

```
SUBROUTINE
getPath(self)
        path ← os.path.dirname(os.path.realpath(__file__))
        path ← (os.path.dirname(os.path.realpath(__file__))+ '\\Enigma Settings')
        RETURN path
END SUBROUTINE
```

```
SUBROUTINE
createFolder(self)
        self.destination ← self.getPath()
        TRY
                os.makedirs(self.destination)
        EXCEPT FileExistsError
                pass
        ENDTRY
ENDSUBROUTINE
```

The code above gets the path where the new folder should be created and attempts to create a folder at that destination. If however, an error is encountered, the program realises that the folder it is trying to create already exists and continues with the program.

The ‘createFolder’ method is called every time the program generates a database object in order to ensure that there is a folder to save the database in. It is nested in the ‘__init__’ method of the data class.

IMPORTING DATABASE

Now that we have some generated values, we must develop an algorithm to assist the user in reading in this data.

```
SUBROUTINE
readData()
        dayToday ← self.currentDay()
        TRY
                self.c.execute("SELECT * FROM Enigma WHERE Datum = " + str(dayToady))
                data ← self.c.fetchall()
```

```
                RETURN data
        EXCEPT
                OUTPUT 'Error reading database. Please choose another database.'
                self.reset()
                self.readData()
ENDSUBROUTINE
```

The algorithm above works by using the datetime function, explained in the previous section, to find the current day. Using this, it selects the settings for the current day by running an SQL query selecting all the data from the table where the record states the current day. Running this algorithm, however, produces a 'NoneType' error when the user enters a non-existent database followed by the correct database.

```
**************     MENU     **************
1. Encrypt/Decrypt using default wiring
2. Encrypt/Decrypt using custom wiring
3. Encrypt/Decrypt using imported settings
4. Generate cipher table
5. Exit application
Choose an option: 3

Enter the name of the database: testDB
Error reading database. Please choose another database.

Enter the name of the database: testDB1
Traceback (most recent call last):
  File "D:\Documents\Enigma Machine Emulator\EnigmaMachineOOP.py", line 583, in
<module>
    Main()
  File "D:\Documents\Enigma Machine Emulator\EnigmaMachineOOP.py", line 580, in
Main
    EnigmaSim.Run()
  File "D:\Documents\Enigma Machine Emulator\EnigmaMachineOOP.py", line 560, in
Run
    self.importSettings()
  File "D:\Documents\Enigma Machine Emulator\EnigmaMachineOOP.py", line 474, in
importSettings
    row = dailySettings[0]
TypeError: 'NoneType' object is not subscriptable
```

Reviewing the code reveals that the try condition in the readData() function was returning the incorrect data after running the first try causing it to crash when the try condition was satisfied. In order to resolve this issue, I have created an 'ELSE' condition where the data is returned only when the try condition is satisfied, preventing the program from crashing. The revised code is shown below.

```
SUBROUTINE
readData()
        dayToday ← self.currentDay()
        TRY
                self.c.execute("SELECT * FROM Enigma WHERE Datum = " + str(dayToady))
                data ← self.c.fetchall()
        EXCEPT
                OUTPUT 'Error reading database. Please choose another database.'
                self.reset()
                self.readData()
        ELSE
                RETURN data
ENDSUBROUTINE
```

Testing this function successfully returns the whole record to the user for the current day's settings. Below is the format of the database in which the data is stored.

	Datum	Walzenlage	Ringstellung	Steckerverbindungen	Kenngruppen
	Filter	Filter	Filter	Filter	Filter
1	1	IV III II	20 19 07	BO YX HP TV RU KF QL MS JA NW ZC GD EI	wgt jgv cog lhx
2	2	I V II	18 22 05	NH OB LX IQ FT MC YZ AS EV RJ KG DU PW	pgs muh nve iow
3	3	I V II	07 05 06	NM WA OV IF TB KU RQ DG JS ZC PE YL XH	tje mex hed usw
4	4	II IV I	10 15 21	BI SY VK HW NT OG ZA CL QJ RX UM ED PF	awu tsw xur yxr

In order to make use of the returned data, we can store the data into the variable ‘dailySettings’ and define a second variable, ‘row’ which is the first element in the daily settings. As this selects all the information, we will need to break it down further to assign each data element to its respective variable.

```
rotors ← row[1]
rotorL, rotorM, rotorR ← rotors.split()
rotorL, rotorM, rotorR ← STR(rotorL), STR(rotorM), STR(rotorR)
```

Obtaining the rotors

In order to obtain the rotors, we can select the element at index 1 and split it into three string variables. This allows us to store the rotors into three rotor variables.

```
ringSettings ← row[2]
ringL, ringM, ringR ← ringSettings.split()
ringL, ringM, ringR ← INT(ringL), INT(ringM), INT(ringR)
```

Obtaining the ring setting

We can use a similar method to obtain the ring settings by selecting the element at index 2 and splitting it into three integer variables. This allows us to store the ring settings into three ring variables.

Obtaining the plugboard pairs

We can obtain the plugboard pairs by selecting the element at index 3 in the row array. As the plugboard pairs are already stored as pairs, we do not need to do anything further as they are in a format in which the plugboard class can deal with.

Obtaining the character groups

The character groups can be obtained by selecting the element at index 4 in the row. We do not need to do anything further as they are output as they are for the user to decide which group to use.

Now that we have extracted all the necessary data from the imported data, we can take in user input to determine other user defined settings.

```
startL, startM, startR ← input('\nEnter rotor starting positions: ').split()
startL, startM, startR ← INT(startL), INT(startM), INT(startR)
```

Obtaining starting positions

We can prompt the user to enter the starting positions for each rotor and we can split their entry into 3 individual starting positions for each rotor.

```
reflectorType ← input("Enter reflector type: ").upper()
```

Obtaining reflector type

We can obtain the reflector type by also prompting the user to enter which reflector they would like to use.

Initializing the start positions and ring settings

We must initialise the start positions and ring setting so that we can generate rotor objects at the ring setting and starting positions specified by the user. Combining all the code above into the code below, we initialise the ring settings as 0 and the starting positions as 0.

After the user has entered the rotor starting positions, the method searches through the ring dictionary to find the corresponding ring variable for each rotor. It then sets this value to the value input by the user and repeats this process for the start positions.

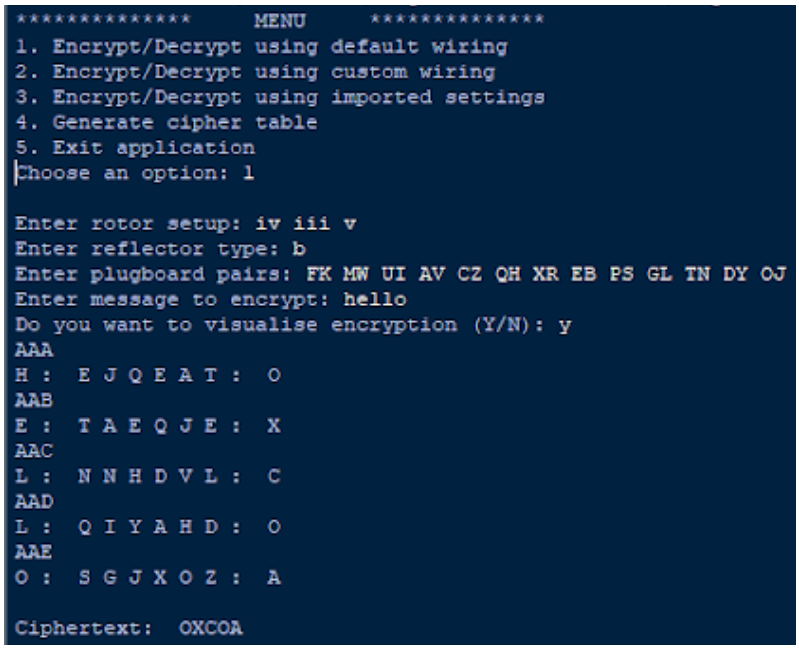
```
ring1, ring2, ring3, ring4, ring5 ← 0, 0, 0, 0, 0
start1, start2, start3, start4, start5 ← 0, 0, 0, 0, 0

ringSettings ← row[2]
ringL, ringM, ringR ← ringSettings.split()
ringL, ringM, ringR ← int(ringL), int(ringM), int(ringR)

startL, startM, startR ← (input('\nEnter rotor starting positions: ').split())
startL, startM, startR ← int(startL), int(startM), int(startR)

rings ← {'I':ring1, 'II':ring2, 'III':ring3, 'IV':ring4, 'V':ring5}
start ← {'I':start1, 'II':start2, 'III':start3, 'IV':start4, 'V':start5}

setLRing ← rings.get(rotorL)
setMRing ← rings.get(rotorM)
setRRing ← rings.get(rotorR)

setLRing ← ringL
setMRing ← ringM
setRRing ← ringR

setLStart ← start.get(rotorL)
setMStart ← start.get(rotorM)
setRStart ← start.get(rotorR)

 setLStart ← startL
 setMStart ← startM
 setRStart ← startR
```

```
self._rotor1 ← rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', start1, ring1)
self._rotor2 ← rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', start2, ring2)
self._rotor3 ← rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', start3, ring3)
self._rotor4 ← rotor('ESOVPZJAYQUIRHXLNFTGKDCMWB','R','J', start4, ring4)
self._rotor5 ← rotor('VZBRGITYUPSDNHLXAWMJQOFECK','H','Z', start5, ring5)
```

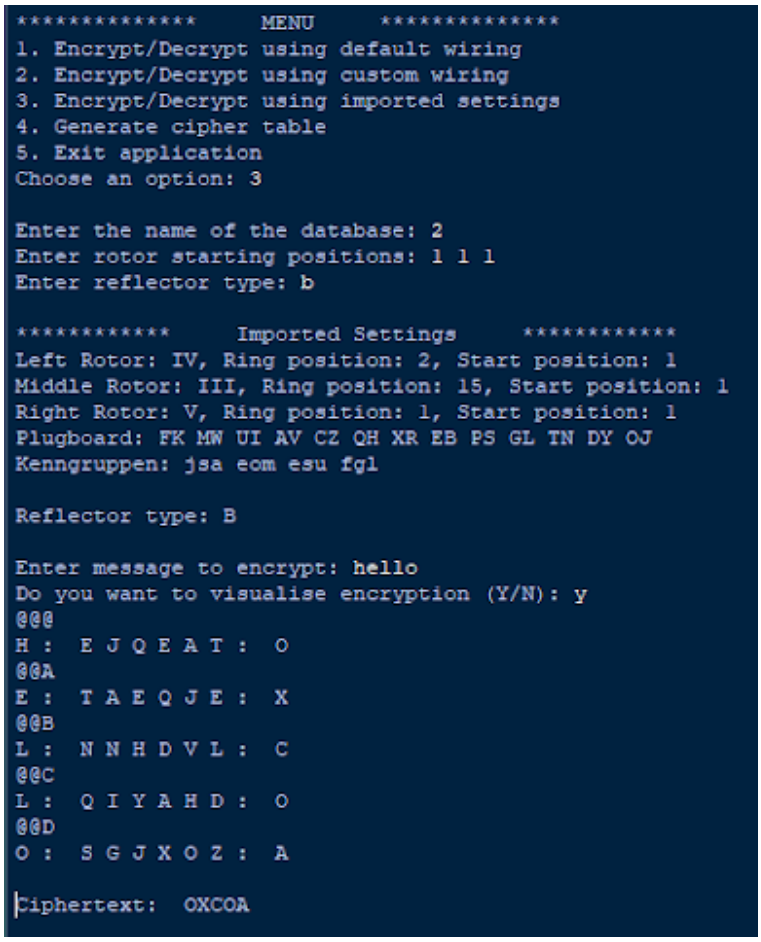
After the start positions and ring positions have been initialised, each of the 5 rotors is created as an object for the user to use during encryption.

From the screenshots below, we can see that function accurately replicates the encryption of the enigma, however, there seems to be an issue when displaying the rotors as they rotate during the encryption process.

Default setting rotation produces the correct rotor positions:



Imported setting rotation produces erroneous rotor positions:



Analysing the code, we can see that the cause of this issue is that the start positions of the rotor were initialised as 0 as can be seen in the code provided in the simulation class section. We can resolve this issue by initialising the start positions as 1 because then the decremented value is within the range specified for the indexed alphabet of the rotor positions which are from indexes 0 to 25 instead of 1 to 26.

However, this issue has made it clear that the rotors are not acknowledging the start positions specified by the user for positions other than the start positions. In fact, they are using the initialized values and so we need to adapt the code to accept the user input positions.

```
self.ring1, self.ring2, self.ring3, self.ring4, self.ring5 ← 0, 0, 0, 0, 0
self.start1, self.start2, self.start3, self.start4, self.start5 ← 1, 2, 3, 4, 5
```

```
ringSettings ← row[2]
ringL, ringM, ringR ← ringSettings.split()
ringL, ringM, ringR ← int(ringL), int(ringM), int(ringR)

startL, startM, startR ← input('\nEnter rotor starting positions: ').split()
startL, startM, startR ← int(startL), int(startM), int(startR)

ring ← {'I':'ring1', 'II':'ring2', 'III':'ring3', 'IV':'ring4', 'V':'ring5'}
start ← {'I':'start1', 'II':'start2', 'III':'start3', 'IV':'start4', 'V':'start5'}

setLStart ← str(start.get(rotorL))
setMStart ← str(start.get(rotorM))
setRStart ← str(start.get(rotorR))

vars(self)[setLStart] ← startL
vars(self)[setMStart] ← startM
vars(self)[setRStart] ← startR

setLRing ← str(ring.get(rotorL))
setMRing ← str(ring.get(rotorM))
setRRing ← str(ring.get(rotorR))

vars(self)[setLRing] ← ringL
vars(self)[setMRing] ← ringM
vars(self)[setRRing] ← ringR

self._rotor1 ← rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', self.start1, self.ring1)
self._rotor2 ← rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', self.start2, self.ring2)
self._rotor3 ← rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', self.start3, self.ring3)
self._rotor4 ← rotor('ESOVPZJAYQUIRHXLNFTGKDCMWB','R','J', self.start4, self.ring4)
self._rotor5 ← rotor('VZBRGITYUPSDNHLXAWMJQOFECK','H','Z', self.start5, self.ring5)
```

Revising the code, we can change the ring and start dictionaries to include a string of the respective ring and start position variables. We can then assign this string value to a variable and use this variable to create a new dynamic variable using the vars() function which returns a dictionary containing the objects changeable attributes. By converting the string of the variables

into the actual variable names, we can assign the user input values to each variable. As the variables are already defined above, their values are updated for the user to use when the rotor objects are instantiated in the following lines.

For example, if we take rotor L as ‘II’, the line ‘setLStart ← str(start.get(rotorL))’ would look through the start dictionary and select the string ‘start2’ as the corresponding start variable for rotor II.

The line ‘vars(self)[setLStart] ← startL’ would then create a dynamic variable from the value of ‘setLStart’ and initialise its value as ‘startL’ which is the user input for the starting position of the leftmost rotor. Therefore, we assign the user input to the variable start2.

Testing this, we can see that the rotor positions are correctly updated with the user input, 4 5 6 (D E F) and that the application no longer uses the initial values for the start and ring positions.

```
**************     MENU     **************
1. Encrypt/Decrypt using default wiring
2. Encrypt/Decrypt using custom wiring
3. Encrypt/Decrypt using imported settings
4. Generate cipher table
5. Guidance and operation
6. Exit application
Choose an option: 3

Enter the name of the database: testdb

Enter rotor starting positions: 4 5 6
Enter reflector type: b

************     Imported Settings     ************
Left Rotor: I, Ring position: 11, Start position: 4
Middle Rotor: II, Ring position: 13, Start position: 5
Right Rotor: IV, Ring position: 1, Start position: 6
Kenngruppen: hnf fzb eta cbn
Plugboard: SU ID GE FK RA HL QT OJ BM CY NV PZ WX
Reflector type: B

Enter message to encrypt: enigma

Enter the frequency of character grouping: 2
Do you want to visualise encryption (Y/N): y
DEF
E :  L Y H D C N :  B
Double Step
EFG
N :  H R E Q N Z :  B
EFH
I :  A D M O O I :  M
EFI
G :  Y F V W Q S :  T
EFJ
Single Step
M :  Y M T Z C Z :  X
EGK
A :  D V F S J O :  S

Ciphertext:  BB MT XS
```

An issue that I encountered during the making of this algorithm was that the vars() function created dynamic variables when it was not contained inside a method. After I had put it within the 'importSettings' method, it did not execute the vars() line correctly. In order to resolve this issue, I instantiated each of the start positions and ring settings as attributes using self and then I adjusted each instance of the variables accordingly so that they were all attributes of the class. This solution seems to work perfectly as intended.

SIMULATION CLASS

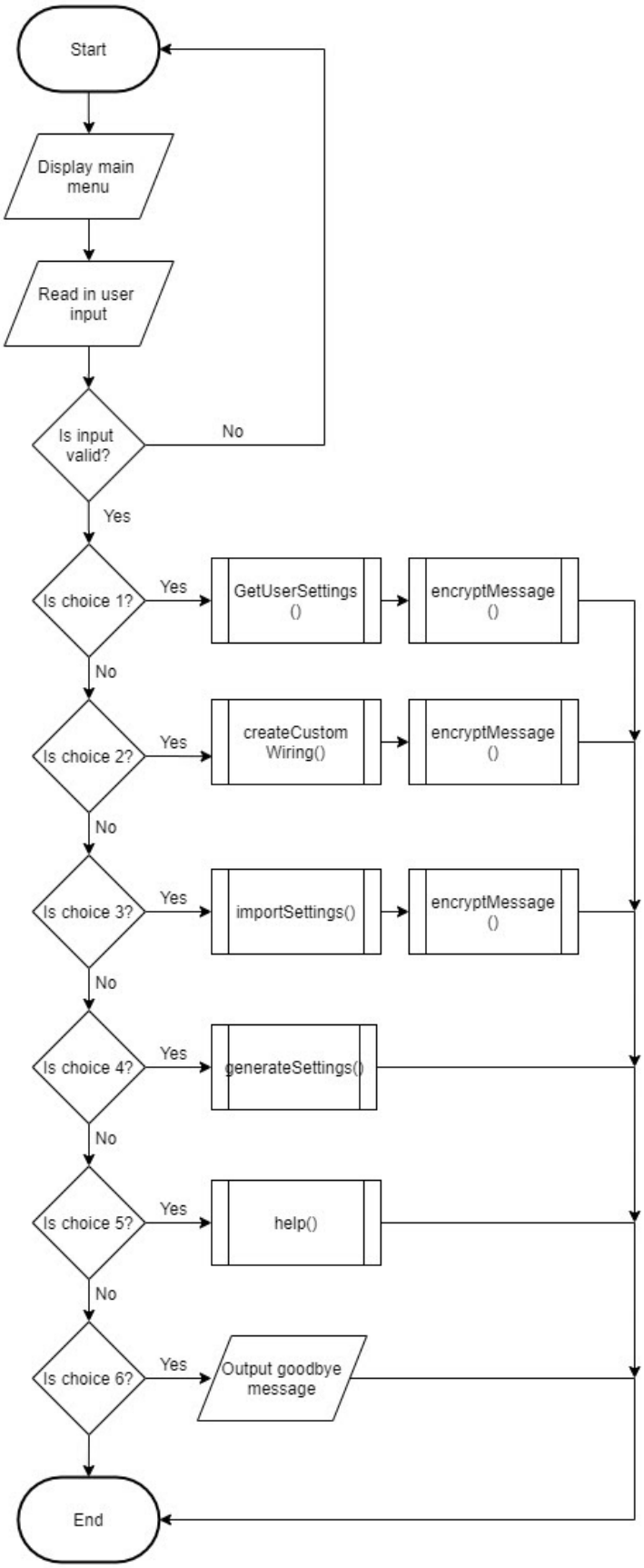
The simulation class brings all of the algorithms and classes above so that a simulation of the enigma machine can be run. It is the interface by which the user will interact with the program. It also allows us to use the stepping algorithms which we developed above so that we can compare different rotor objects after they have been instantiated.

From our primary objectives, the simulation class must:

- Allow user to exit program (MO 1.1)
- Encode messages by prompting the user to enter settings and validating their input (MO 1.2, 1.2.1, 1.2.3)
- Allow the user to visualise encryption (MO 1.2.2)
- Decode messages and allow the user to group ciphertext (MO 1.3, 1.3.1, 1.3.2)
- Generate cipher tables for the month (MO 1.4, 1.4.2.2)
- Export cipher tables (MO 1.4.2, 1.2.2.1)
- Import cipher tables and validate input (MO 1.5, 1.5.1, 1.5.1.1, 1.5.1.2)
- Provide the user with guidance on how to operate the enigma machine and the format of the input when using the application. (MO 1.6.1)

MAIN MENU

Upon starting the application, the user will be presented with a main menu and they will be prompted to make a choice. These choices will be to encode/decode using normal wiring, encode/decode using custom wiring, generating a cipher table, importing a cipher table and exiting the application. I have also added an extra option at the user's request so that there is a guidance section stating the format the input should be entered in and how to use the application.


Start
Display main menu
Read in user input
Is input valid?
No
Yes
Is choice 1?
Yes
GetUserSettings ()
encryptMessage ()
No
Is choice 2?
Yes
createCustom Wiring()
encryptMessage ()
No
Is choice 3?
Yes
importSettings()
encryptMessage ()
No
Is choice 4?
Yes
generateSettings()
No
Is choice 5?
Yes
help()
No
Is choice 6?
Yes
Output goodbye message
End

The main menu will be designed so that upon incorrect input, it will prompt the user for input again until the user makes a valid choice. This can be achieved by looping the menu until the user enters a valid input or until the user exits the application.

RECORDING INPUT SETTINGS

After the user has chosen to encrypt or decrypt a message, the user will be prompted to enter the settings for the enigma machine. Using the default settings will prompt the user to choose from rotors I II III IV V, reflectors A B C, any sequence of valid plugboard pairs, any valid starting position and any valid ring setting.

These settings will be limited by validating the input in order to check that they are in accordance with the actual guidance on operating the Enigma machine. For example, error messages should be displayed if the user enters settings containing two or more of the same rotor type or re-enters a character when entering the plugboard connections. This is because using the same rotor to encrypt a message would compromise the integrity of the system and having the same plugboard connection would be impossible as the socket would already be in use, connected to another letter.

If the user chooses to encrypt using their custom settings, they too will be prompted to define 3 new rotors, define a new reflector, any valid sequence of plugboard pairs, any valid combination of starting positions and any valid combinations of ring settings. Here, the user will be able to carry out the same operations as the default code but the operation will be allowed to reuse the same rotors and plugboard connections, if the user enters the same sequence or combinations, in order to experiment and see how this will affect the security and integrity of the enigma.

```
loopRotors ← True
WHILE loopRotors == True
        rotorL, rotorM, rotorR = USERINPUT("Enter rotor setup: ").upper().split()
        IF rotorL == rotorM OR rotorL == rotorR OR rotorM == rotorR
                loopRotors ← True
                OUTPUT 'Rotors can not be the same. Try again'
        ELSE
                loopRotors = False
        ENDIF
ENDWHILE
```

By reading in the user input settings as a string, we can verify that the rotors are different by iterating through this string to see if there are any matches when the user is using the default settings. The algorithm above works by splitting the string of rotors into their separate rotor variables and compares each variable with the others in order to see if there is a match. If there is a match, an error message will be displayed and the user will be prompted to enter the rotors again. If the input is valid, the 'loopRotors' boolean which loops the algorithm will be set to false allowing the user to continue using the program.

```
loopPlugboard  ← True
WHILE loopPlugboard == True
        plugboardPairs ← USERINPUT('Enter plugboard pairs: ').upper().strip()
        stringPairs ← plugboardPairs.replace(" ", "")

        IF LEN(stringPairs) != 0
                frequencies ← collections.Counter(stringPairs)
                repeated ← {}
                FOR k, v IN frequencies.items()
                        IF v > 1
                                repeated[k] ← v
                        ENDIF
                ENDFOR
                IF LEN(repeated) != 0:
                        OUTPUT 'Each character may only connect to another character. Try
                again.'
                        loopPlugboard ← True
                ELSE
                        loopPlugboard ← False
                ENDIF
        ELSE
                loopPlugboard ← False
        ENDIF
ENDWHILE
```

The plugboard input will also undergo a similar iteration to ensure that there are no repeating connections as each socket on the plugboard can only connect with one other socket in the

physical enigma machine. The algorithm above takes the plugboard connections and removes all the space characters in the string. This is because we will count the frequency of each recurring character in order to determine if there are any repeat characters.

The ‘IF’ statement on the 5th line checks to see if the user had entered any settings for the plugboard. If no connections have been made on the plugboard, the algorithm continues as normal. However, if the length of the user input is greater than 0, this indicates that the user has entered and made certain plugboard connections.

In order to check that these connections are unique, we can use the collections module and use the ‘Counter’ method to count the frequencies of each recurring character and this can be stored in the repeated dictionary. However, only characters with a frequency greater than one are recorded as characters will only repeat if there is more than one occurrence of that character. When the length of the repeated dictionary is checked, it should be empty to indicate no repeats at which point the ‘loopPlugboard’ boolean is set to False and the program carries on as normal. If however, the length of the repeated dictionary is greater than 0, the program will identify that there are repeating characters and will output an error message warning the user. The ‘loopPlugboard’ boolean will remain True and the algorithm will loop until the user has entered valid input.

Custom wiring

If the user chooses to enter their own custom wiring, the below algorithm will iterate through all the input again and prompt the user to enter their custom wiring for three individual rotors ensuring that only valid ASCII characters from A - Z are entered. This must be done as the code is not compatible with other characters as the algorithms adjusting the positions and observing the iterations and rotations are bespoke to only capital characters as the foundation of the algorithm depends on the ASCII codes for this limited range of capital characters.

```
loopAlphaInput ← True

WHILE loopAlphaInput == True:
        TRY
                rotorL ← STR(USERINPUT('Enter left rotor alphabet: ')).upper().strip()
                if not re.match("^[A-Z]+$", rotorL):
                        raise ValueError()
```

```
                rotorM ← STR(USERINPUT('Enter middle rotor alphabet: ')).upper().strip()
                if not re.match("^[A-Z]+$", rotorM):
                        raise ValueError()

                rotorR ← STR(USERINPUT('Enter right rotor alphabet: ')).upper().strip()
                if not re.match("^[A-Z]+$", rotorR):
                        raise ValueError()

        EXCEPT
                OUTPUT "Please limit characters to A - Z"

        ELSE
                loopAlphaInput ← False
        ENDTRY
ENDWHILE
```

We can check that only ‘A - Z’ characters are entered for the rotor alphabets by taking in the user input and converting it all to uppercase. We can then use the ‘re’, regular expressions, module in order to evaluate the user input and check that the input is in the character set.

By using the regular expression ‘^[A-Z]+$’, we can evaluate the user input so that it starts, denoted by ^, with any character for the alphabet, denoted by [A-Z], one or more occurance of any character in the alphabet, denoted by +, and it must end with any character from the alphabet, denoted by $. Therefore, we can ensure that only characters that can be encoded are entered.

The user will then be required to input their own reflector sequence and this will be iterated through to ensure that each instance in the character set occurs only once and that all the characters are paired. This is done because the reflector must facilitate all characters in the alphabet otherwise certain characters would not be able to be encoded as the reflector would not accommodate them and return a value for them if and when they are passed into the reflector. The reflector will use the same code that is used to encode the plugboard which is referenced above.

After the initial configuration of the wiring, the user will then be promoted to enter the positions of each rotor, the reflector they would like to use and finally the plugboard pairs. The plugboard pairs will not be validated in custom wiring for experimental purposes as stated above.

ENCRYPTING USER INPUT

We are able to encrypt the user input by evaluating valid user input using a similar regular expression algorithm as was used for the custom rotor wiring above. If the user enters characters that are not within the character set, they are returned to the main menu.

```
TRY
        plaintext ← STR(USERINPUT(“Enter message to encrypt: ”)).upper()
        IF NOT re.match(“^[A-Z]*$”, plaintext):
                raise ValueError()
        ENDIF
EXCEPT
        OUTPUT “Limit characters to A - Z”
        Main()
```

After the user has input their message, we can use the cipher method in the simulation class and use it to encrypt a character, repeating this process until the whole message is encrypted.

This code literally puts all the individual lines of code in the rotor class - rotation algorithm section into one method for ease of use. It ciphers the input through the plugboard, from the external contact, through the middle and left rotors, in and out of the reflector and back through the left middle and right rotors until the ciphered character is output to the user.

```
SUBROUTINE
cipher(self, char)
        position ← self._rotorL.chrNum(self._rotorL.GetRotorPosition() - 1),
        self._rotorM.chrNum(self._rotorM.GetRotorPosition() - 1),
        self._rotorR.chrNum(self._rotorR.GetRotorPosition() - 1)

        IF self.visuals == 'Y'
                OUTPUT ''.join(position)
        ENDIF
```

```
        char0 ← char
        char ← self._plugboard.switchChar(char)
        char1 ← char
        char ← self._rotorR.cipherFromExternalContact(char)
        self.singleStep(self._rotorL, self._rotorM, self._rotorR)
        char1 ← char
        char ← self._rotorM.cipherToReflector(char)
        char2 ← char
        char ← self._rotorL.cipherToReflector(char)
        char3 ← char
        char ← self._UKW.reflect(char)
        char4 ← char
        char ← self._rotorL.cipherFromReflector(char)
        char5 ← char
        char ← self._rotorM.cipherFromReflector(char)
        char6 ← char
        char ← self._rotorR.cipherFromReflector(char)
        char7 ← char
        char ← self._plugboard.switchChar(char)

        IF self.visuals == 'Y':
                OUTPUT char0,": ",char1, char2, char3, char4, char5, char6,": ",char
                time.sleep(0.1)
        ENDIF

        self.doubleStep(self._rotorL, self._rotorM, self._rotorR)
        RETURN char
ENDSUBROUTINE
```

We can visualise the encryption of the enigma machine by using the boolean attribute ‘visuals’ to determine if the user would like to see the encryption of each character. If the user enters a ‘y’ or ‘Y’ character, the program will print the current rotor positions followed by the encryption of the character ending with the output character. This is done as the algorithm records the character before it is input into the enigma machine, after it has passed through the plugboard, through all the rotors to the reflector, from the reflector through all the rotor back to the plugboard and from

the plugboard back to the output light bulb. The positions of the rotor are also output by getting the current rotor positions using the method ‘GetRotorPosition’. As this method adds one to the position because the positions are decremented by one when they are indexed from zero, we need to subtract one so that the ‘chrNum’ can find the ASCII value of the corresponding letter so that it may be output to the screen.

In the event of a single or double step, the program will also notify the user that a single or double step has occurred by printing a short statement in the sequence to indicate this. This is done using the stepping algorithms explained in the rotor class ection of the documented design. The program checks for a single step after the first character has passed through the plugboard and a double step after the user input has been ciphered. This is done because a single step only needs to increment the second rotor which can be done as the third rotor does not move, however, in a double step, all three rotors need to move and the first and second rotors move twice in succession so the program must know this in advance so that it can increment the middle rotor. If the double step was checked at the start instead, the middle rotor would not have incremented successively producing incorrect output.

```
ciphertext ← " "

FOR i IN plaintext
        char ← self.cipher(i)
        ciphertext ← ciphertext + char
ENDFOR
```

The ‘encryptMessage’ method above then adds the ciphered character to a ciphertext string until every character in the plaintext has been encrypted.

```
group ← INT(USERINPUT('Enter the frequency of character grouping: '))
ciphertext ← [ciphertext[i:i+INT(group)] FOR i IN RANGE(0, LEN(ciphertext), group)]
ciphertext ← ' '.join(ciphertext)
```

After the whole message has been encrypted, we can group the ciphertext, using the code above, as you would in a real enigma machine in order to deduce the constituent words of the message. By prompting the user to enter a grouping frequency, we can identify the interval in which to group the characters in the message and we can iterate through the message, splitting it every after character at the position of the user input. This then creates a sequence of

independent groups of characters and so we must use the ‘join()’ method in order to join them with a space character so that they are ready to be presented to the user when the ciphertext is output onto the screen.

IMPORTING AND EXPORTING SETTINGS

The user can import and use their own settings by using the methods specified in the database class. These methods are explained in more detail in the ‘exporting database’ and ‘importing database’ sections above. These methods will validate the input settings and also generate values for each day of the month, automatically changing their settings as the database is specified in the program. In doing so, we will eliminate the need to verify the settings are not outdated as the program will automatically select the settings for the current date by using the datetime module.

At the users request, I have also included code which creates a folder for the user to store their databases and code to display the path of the exported database for the user's reference. More on this is covered in the database section above.

TECHNICAL SOLUTION

PROGRAM CODE

The program code has been organised and alphabetized in the order of the class diagram in the high level overview for ease of use.

```
import time
import datetime
import sqlite3
from sqlite3 import Error
import os
import random
import re
import collections

class rotor:
    def __init__(self, alphabet, notch, turnover, position, ringSetting):
        """Create new rotor"""
        self._charAlphabet = alphabet #Initialises alphabet as tuple csv

        self.alphabet = self.GetAlphabet() #Parses the alphabet as a dictionary so that each character is connected to
a position
        self._notch = notch
        self._turnover = turnover
        self._position = int(position) - 1 #Finds the index of the rotor position
        self._ringSetting = int(ringSetting)
        self.adjustAlphabetByPosition()

    def adjustAlphabetByPosition(self):
        """Adjusts alphabet to start on different rotor positions"""
        for i in range(0, self._position): #Rotates rotors until it reaches the input position without stepping or
incrementing the position attribute
            self.adjustRotor()

    def adjustRotor(self):
        """Rotate the rotor without incrementing it or stepping"""
        #Ensures next positions are within alphabet range
        for k, v in self.alphabet.items():
            self.alphabet[k] = (v - 1)% 26 #Loops the alphabet from Z (26th letter) back to A (1st letter)

        #Swaps dictionary format from {'Key': value} to {value: 'Key'} so we can change the characters
        tempAlphabet = {}
        for k, v in self.alphabet.items():
            tempAlphabet[v] = k
        self.alphabet = tempAlphabet

        #Changes internal wiring contact to its previous contact
        for k, v in self.alphabet.items():
            charNum = self.ordChar(v)
            prevCharNum = (charNum - 1)% 26
            prevChar = self.chrNum(prevCharNum)
            self.alphabet[k] = prevChar
```

```
        #Swaps dictionary format from {value: 'Key'} back to original {'Key': value} format
        tempAlphabet = {}
        for k, v in self.alphabet.items():
            tempAlphabet[v] = k
        self.alphabet = tempAlphabet

    def changeRingSetting(self):
        """Changes ring setting and creates offset variable"""
        #Input code to accommodate function of Ring setting

    def chrNum(self, num):
        """Finds the ASCII code of input index position"""
        char = chr(num + 65)
        return char

    def cipherFromExternalContact(self, char):
        """Cipher the user input character"""
        self.incrementRotor() #Incements the rotor as the key is 'pressed' on the enigma machine
        char = self.cipherToReflector(char) #Ciphers a character in the forward direction to the reflector
        return char #Returns character

    def cipherFromReflector(self, char):
        """Cipher character produced by previous rotor travelling from reflector"""
        inputCharNum = self.GetNumByChar(char) #Finds the index of the value for the input character to find the
internal wiring connection
        outputChar = self.chrNum(inputCharNum) #Finds the ASCII code of the index value to find the external rotor
connection
        return outputChar #Returns the external character that the wiring is connected to

    def cipherToReflector(self, char):
        """Cipher character produced by previous rotor travelling to reflector"""
        inputCharNum = self.ordChar(char) #Finds the index value of the input ASCII code to find the external rotor
connection
        outputChar = self.GetCharByNum(inputCharNum) #Finds the corresponding character of the input to find the
internal wiring connection
        return outputChar #Finds the internal wiring contact that the character is connected to

    def GetAlphabet(self):
        """Parses alphabet as dictionary"""
        alphabet = list(self._charAlphabet) #Creates a list of the alphabet characters
        numbers = [i for i in range(0,26)] #Creates a list of numbers up to 25
        numberOff = dict( zip(alphabet, numbers)) #Pairs each character with a number in a chronological sequence to
number the characters from 0 to 25

        return numberOff

    def GetCharByNum(self, inputNum):
        """Find the corresponding character of the input number"""
        # ROTOR USE ONLY
        for char, num in self.alphabet.items():
            if num == inputNum:
                return char

    def GetNotch(self):
        """Gets notch position"""
        return self._notch

    def GetNotchNum(self):
        """Gets numerical value of notch"""
```

```
        num = self.ordChar(self._notch) + 1
        return num

    def GetNumByChar(self, inputChar):
        """Finds the corresponding number of the input character"""
        # ROTOR USE ONLY
        for char, num in self.alphabet.items():
            if char == inputChar:
                return num

    def GetRingSetting(self):
        """Gets ring setting"""
        return self._ringSetting

    def GetRotorPosition(self):
        """Gets position of rotor"""
        position = self._position + 1
        return position

    def GetTurnoverNum(self):
        """Get numerical value of turnover character"""
        num = self.ordChar(self._turnover) + 1
        num = num % 26
        return num

    def incrementRotor(self):
        """Rotates rotor forward by one position"""
        self._position += 1
        self._position = self._position % 26

        #Ensures next positions are within alphabet range
        for k, v in self.alphabet.items():
            self.alphabet[k] = (v - 1)% 26

        #Swaps dictionary format from {'Key': value} to {value: 'Key'} so we can change the characters
        tempAlphabet = {}
        for k, v in self.alphabet.items():
            tempAlphabet[v] = k
        self.alphabet = tempAlphabet

        #Changes internal wiring contact to its previous contact
        for k, v in self.alphabet.items():
            charNum = self.ordChar(v)
            prevCharNum = (charNum - 1)% 26
            prevChar = self.chrNum(prevCharNum)
            self.alphabet[k] = prevChar

        #Swaps dictionary format from {value: 'Key'} back to original {'Key': value} format
        tempAlphabet = {}
        for k, v in self.alphabet.items():
            tempAlphabet[v] = k
        self.alphabet = tempAlphabet

    def ordChar(self, char):
        """Finds the index value of input ASCII code"""
        char = char.upper()
        num = ord(char) - 65
        return num
```

```
class plugboard:
    def __init__(self, connections):
        """Connect plugboard plugs"""
        self._connections = connections.split()

    def switchChar(self, char):
        for pair in self._connections:
            for letter in pair:
                if letter == char: #If input matches the letter
                    index = pair.index(letter) #Find the index of the letter
                    index += 1 #Increment it by one
                    sub = pair[(index % len(pair))] #Ensure it is within the index range
                    return sub #Return the substituted character to the user
        else:
            return char #Return character if not in a pair


class reflector:
    def __init__(self, connections):
        """Set reflector substitution pairs"""
        self._connections = connections.split()

    def reflect(self, char):
        for pair in self._connections:
            for letter in pair:
                if letter == char: #If input matches the letter
                    index = pair.index(letter) #Find the index of the letter
                    index += 1#Increment it by one
                    sub = pair[(index % len(pair))] #Ensure it is within the index range
                    return sub #Return the substituted character to the user


class data:
    def __init__(self):
        """Database creation"""
        self.destination = ''
        self.createFolder() #Create a folder to export databases to
        self.reset() #Create a database connection

    def createFolder(self):
        """Create a folder to house the databases"""
        self.destination = self.getPath() #Find the destination to create the folder
        try:
            os.makedirs(self.destination) #Try and make a folder
        except FileExistsError:
            pass #Otherwise continue if an error is encountered because the file exists already

    def createTable(self):
        """Create a table within a database"""
        self.c.execute("CREATE TABLE IF NOT EXISTS Enigma(Datum INTEGER PRIMARY KEY, Walzenlage TEXT,
Ringstellung TEXT, Steckerverbindungen TEXT, Kenngruppen TEXT)")

    def currentDay(self):
        """Finds the current day"""
        day = datetime.datetime.today().day
        return day

    def exportTable(self):
        """Exports the table to the destination folder"""
        try:
```

```
            self.createTable() #Create a table
            self.insertData() #Insert the daily settings
            print('Database has been exported to ' + self.destination + '\\'+ self.database + '\n') #Export the table
        except:
            print('Enigma table already exists for this database. Please choose another database.') #Otherwise inform the
user that the table exists
            self.reset() #Prompt a new input for the database name
            self.exportTable() #Try and export the new database using recursion

    def genCharGroup(self):
        """Creates a character group"""
        alphabet = list('abcdefghijklmnopqrstuvwxyz') #Creates a list of all the alphabet characters
        group = []
        count = 0
        while count != 3: #While the loop total does not equal 3
            i = random.choice(alphabet) #Make a random choice
            alphabet.remove(i) #Remove it from the alphabet
            group.append(i) #And add it to the group array
            count += 1 #Add one to the loop total
        return str(''.join(group)) #Return the string of 3 characters to the user

    def genRingSetting(self):
        """Generates a ring setting value"""
        num = random.randrange(0,25) #Generates a random number from 0 to 25
        if num < 10: #If the number is a single digit
            num = str(num) #Turn it into a string
            num = '0' + num #Add a 0 before it
        return str(num) #Return the string of the number to the user in double digit format

    def getPath(self):
        """Identifies the path of the application to create the new folder"""
        path = os.path.dirname(os.path.realpath(__file__)) #Finds the path of the application
        path =(os.path.dirname(os.path.realpath(__file__))+ '\\Enigma Settings') #Adds to the directory to create a folder

        return path #Returns the folders directory

    def insertData(self):
        """Inserts the daily settings into the database"""
        Walzenlage = self.ranRotorOrder() #Obtains a random rotor order
        Ringstellung = self.ranRingSetting() #Obtains s series or random ring settings
        Steckerverbindungen = self.ranPlugboard() #Obtains random plugboard pairs
        Kenngruppen = self.ranCharGroup() #Obtains 4 groups of characters for the Kenngruppen

        Datum = 31 #Begins the data entry at day 31
        while Datum != 0:
            self.c.execute("INSERT INTO Enigma(Datum, Walzenlage, Ringstellung, Steckerverbindungen, Kenngruppen)
VALUES (?, ?, ?, ?, ?)",
                           (int(Datum), str(Walzenlage), str(Ringstellung), str(Steckerverbindungen), str(Kenngruppen))) #Inserts
the entries generated above into the database
            Walzenlage = self.ranRotorOrder() #Generates new random rotor order
            Ringstellung = self.ranRingSetting() #Generates new random ring settings
            Steckerverbindungen = self.ranPlugboard() #Generates new random plugboard pairs
            Kenngruppen = self.ranCharGroup() #Generates 4 new groups of characters for the Kenngruppen
            Datum -= 1 #Decreases the day by one for the next record entry

        self.conn.commit() #Commits all the changes to the database

    def ranCharGroup(self):
        """Generates a random string of 4 character group"""
        group = self.genCharGroup() + ' ' + self.genCharGroup() + ' ' + self.genCharGroup() + ' ' + self.genCharGroup()
```

```
        return group #Returns a string of 4 character groups

    def ranPlugboard(self):
        """Creates random pairs for the plugboard"""
        alphabet = list('ABCDEFGHIJKLMNOPQRSTUVWXYZ') #Creates a list of the alphabet characters
        plugboard = []

        while len(alphabet) != 0: #While there are characters remaining in the alphabet
            i = random.choice(alphabet) #Make a random choice
            alphabet.remove(i) #Remove it from the alphabet
            plugboard.append(i) #And add it to the plugboard array

        raw_plugboard = ''.join(plugboard) #Join all the characters in the plugboard array into a string

        plugboard = [(raw_plugboard[i:i+2]) for i in range(0, len(raw_plugboard), 2)] #Split the array of characters into
groups of 2

        return ' '.join(plugboard) #Join each of the individual arrays to make a string

    def ranRingSetting(self):
        """Creates a random string for the ring setting values"""
        ringSetting = self.genRingSetting() + ' ' + self.genRingSetting() + ' ' + self.genRingSetting()
        return ringSetting #Returns a string of 3 ring setting values

    def ranRotorOrder(self):
        """Selects three random rotors for the daily settings"""
        count = 0
        rotors = ['I','II','III','IV','V']
        rotorOrder = []
        while count != 3: #While 3 choices have not been made
            choice = random.choice(rotors) #Make a random choice from the rotors array
            rotorOrder.append(choice) #Add it to the rotorOrder array
            rotors.remove(choice) #And remove it from the rotors array
            count += 1 #Increment the count variable

        rotorOrder = ' '.join(rotorOrder) #Join the array to make a string of the 3 rotors

        return rotorOrder

    def readData(self):
        """Selects the daily settings record for the current day"""
        dayToady = self.currentDay()

        loopDbInput = True

        while loopDbInput == True: #While there is an error
            try:
                self.c.execute("SELECT * FROM Enigma WHERE Datum = " + str(dayToady)) #Select all the data in the
record for the current day
                data = self.c.fetchall() #Store the selected data in this variable
            except:
                print('Error reading database. Please choose another database.') #Inform the user that there is an error
connecting to the database
                self.reset() #Prompt the user to establish a new database connection
            else:
                loopDbInput = False #Otherwise continue with the program
                return data #And return the daily settings

    def reset(self):
        """Creates a new database connection"""
```

```
        os.chdir(self.destination) #Changes directory to the 'Enigma Settings' folder
        self.database = str(input('\nEnter the name of the database: ' ) + '.db') #Prompts user to enter database name
        self.conn = sqlite3.connect(self.database) #Established a connection to the database
        self.c = self.conn.cursor() #Creates a cursor to traverse the data set


class simulation:
    def __init__(self):
        """Run enigma simulation"""

    def cipher(self, char):
        position = self._rotorL.chrNum(self._rotorL.GetRotorPosition() - 1),
self._rotorM.chrNum(self._rotorM.GetRotorPosition() - 1), self._rotorR.chrNum(self._rotorR.GetRotorPosition() - 1) #Find
the current external position of each rotor

        if self.visuals == 'Y': #If the user wants to see the encryption process
            print(''.join(position)) #Output the current rotor positions
        char0 = char
        char = self._plugboard.switchChar(char) #Substitute the user input through the plugboard
        char1 = char
        char = self._rotorR.cipherFromExternalContact(char) #Increment the rotor and cipher the plugboard output
through the right rotor
        self.singleStep(self._rotorL, self._rotorM, self._rotorR)
        char1 = char
        char = self._rotorM.cipherToReflector(char) #Cipher the output from the right rotor through the middle rotor
        char2 = char
        char = self._rotorL.cipherToReflector(char) #Cipher the output from the middle rotor through the left rotor
        char3 = char
        char = self._UKW.reflect(char) #Reflects the character from the left rotor through the reflector
        char4 = char
        char = self._rotorL.cipherFromReflector(char) #Cipher the output from the reflector through the left rotor
        char5 = char
        char = self._rotorM.cipherFromReflector(char) #Cipher the output from the left rotor through the middle rotor
        char6 = char
        char = self._rotorR.cipherFromReflector(char) #Cipher the output from the middle rotor through the right rotor
        char7 = char
        char = self._plugboard.switchChar(char) #Substitute the output from the right rotor through the reflector
        if self.visuals == 'Y': #If the user wants to see the encryption process
            print(char0,": ",char1, char2, char3, char4, char5, char6,": ",char) #Output the encryption process of the input
            time.sleep(0.1) #Wait 0.1 seconds before continuing to cipher the next character for a gradual effect

        self.doubleStep(self._rotorL, self._rotorM, self._rotorR) #Check if a double step will occur on the next key press
        return char #Return the ciphered character

    def createCustomWiring(self):
        """Creates custom wiring"""
        loopAlphaInput = True

        while loopAlphaInput == True: #While user input is invalid
            try:
                rotorL = str(input('\nEnter left rotor alphabet: ')).upper().strip() #Prompt the user to enter the alphabet for the
left rotor
                if not re.match("^[A-Z]+$", rotorL): #Check if the input is within the alphabet
                    raise ValueError() #Otherwise raise an error

                rotorM = str(input('Enter middle rotor alphabet: ')).upper().strip() #Prompt the user to enter the alphabet for
the left rotor
                if not re.match("^[A-Z]+$", rotorM): #Check if the input is within the alphabet
                    raise ValueError() #Otherwise raise an error
```

```
            rotorR = str(input('Enter right rotor alphabet: ')).upper().strip() #Prompt the user to enter the alphabet for the
left rotor
            if not re.match("^[A-Z]+$", rotorR): #Check if the input is within the alphabet
                raise ValueError() #Otherwise raise an error

        except:
            print("Please limit characters to A - Z\n") #If an error is raised, inform the user that the input is outside the
range

        else:
            loopAlphaInput = False #Otherwise continue using the program

    stepChars = str(input('\nEnter turnover positions: ')).upper().strip() #Prompt the user to enter the 3 turnover
characters
    stepL, stepM, stepR = stepChars.split() #Split the user input into 3 seperate variables
    stepL, stepM, stepR = str(stepL), str(stepM), str(stepR) #Ensure the string of each variable is stored

    notchPos = input('Enter notch positions: ').upper().strip() #Prompt the user to enter the 3 notch characters
    notchL, notchM, notchR = notchPos.split() #Split the user input into 3 seperate variables
    notchL, notchM, notchR = str(notchL), str(notchM), str(notchR) #Ensure the string of each variable is stored


    ringSettings = input('\nEnter ring setting positions: ') #Prompt the user to enter the 3 ring setting values
    ringL, ringM, ringR = ringSettings.upper().split() #Split the user input into 3 seperate variables
    ringL, ringM, ringR = int(ringL), int(ringM), int(ringR) #Ensure the integer of each variable is stored

    startPos = input('Enter rotor starting positions: ') #Prompt the user to enter the 3 rotor starting positions
    startL, startM, startR = startPos.split() #Split the user input into 3 seperate variables
    startL, startM, startR = int(startL), int(startM), int(startR) #Ensure the integer of each variable is stored


    plugboardPairs = str(input('\nEnter plugboard pairs: ')).upper().strip() #Prompt the user to enter plugboard pairs

    loopReflectorInput = True

    while loopReflectorInput == True: #While user input is invalid
        reflectorPairs = str(input('Enter reflector pairs: ')).upper().strip() #Prompt user to enter reflector pairs
        reflectorInput = reflectorPairs.replace(" ", "") #Omit spaces in reflector pairs

        if len(reflectorInput) != 26: #Check if all characters in the alphabet have been entered
            print('Please enter all 26 characters from A - Z\n') #Inform user that all characters in the alphabet must be
entered

        else:
            reflectorFrequencies = collections.Counter(reflectorInput) #Count the frequency of each character
            reflectorRepeats = {}
            for k, v in reflectorFrequencies.items(): #For every frequency pair
                if v > 1: #If there is more than one occurrence
                    reflectorRepeats[k] = v #Add it to the reflectorRepeats dictionary
            if len(reflectorRepeats) != 0: #If there are repeats in the reflectorRepeats dictionary
                print('Each character  may only connect to another character . Try again.\n') #Notify user that there
repeats
                loopReflectorInput = True #Prompt the user to enter the reflector pairs again
            else:
                loopReflectorInput = False #Otherwise continue with the program

    print("\n**********     Custom Settings     **********") #Output custom settings
    print("Left Rotor:", rotorL + ", Ring position:", str(ringL) + ", Start position:", str(startL) + ", Notch position:",
str(notchL))
    print("Middle Rotor:", rotorM + ", Ring position:", str(ringM) + ", Start position:", str(startM) + ", Notch position:",
```

```
str(notchM))
        print("Right Rotor:", rotorR + ", Ring position:", str(ringR) + ", Start position:", str(startR) + ", Notch position:",
str(notchR))

        print("\nPlugboard:", plugboardPairs)
        print("Reflector pairs:", reflectorPairs + '\n')

        #FORMAT       rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1)
        self._rotorL = rotor(rotorL, notchL, stepL, startL, ringL) #Create a rotor object for the left rotor
        self._rotorM = rotor(rotorM, notchM, stepM, startM, ringM) #Create a rotor object for the middle rotor
        self._rotorR = rotor(rotorR, notchR, stepR, startR, ringR) #Create a rotor object for the right rotor

        self._UKW = reflector(reflectorPairs) #Create a reflector object using the reflector pairs

        self._plugboard = plugboard(plugboardPairs) #Create a plugboard object using the plugboard pairs


    def displayMenu(self):
        """Outputs main menu"""
        print("**************      MENU      **************")
        print("1. Encrypt/Decrypt using default wiring")
        print("2. Encrypt/Decrypt using custom wiring")
        print("3. Encrypt/Decrypt using imported settings")
        print("4. Generate cipher table")
        print("5. Guidance and operation")
        print("6. Exit application")
        print("Choose an option: ", end = "")

    def doubleStep(self, rotorL, rotorM, rotorR):
        """Checks if a double step will occur on the next key press"""
        Mpos = rotorM.GetRotorPosition() #Find the current position of the middle rotor
        Mturn = rotorM.GetTurnoverNum() #Find the turnover position of the middle rotor

        if (Mpos == Mturn): #If the rotor is at its turnover position
            if self.visuals == 'Y': #And the user wants to visualise encryption
                print('Double Step') #Notify the user that there is a double step
            rotorM.incrementRotor() #Increment the middle rotor
            rotorL.incrementRotor() #And increment the left rotor

    def encryptMessage(self):
        """Encrypts the message and outputs the encrypted ciphertext"""
        loopPrompt = True

        while loopPrompt == True: #While user input is invalid

            try:
                plaintext = str(input("Enter message to encrypt: ")).upper() #Prompt the user to enter their message
                plaintext = plaintext.replace(" ", "") #Remove spaces from the input message
                if not re.match("^[A-Z]*$", plaintext): #If the input is not within the alphabet
                    raise ValueError() #Raise an error
            except:
                print("Please limit characters to A - Z\n") #And notify the user that the input is outside the range
            else:
                loopPrompt = False #Otherwise continue with the program


        self.group = int(input('\nEnter the frequency of character grouping: ')) #Prompt user to enter frequency of
character grouping
        self.visuals = input('Do you want to visualise encryption (Y/N): ').upper() #Ask if user wants to visualize
encryption
```

```
        plaintext = list(plaintext) #Split the plaintext into an array

        ciphertext = ""

        for i in plaintext: #For each letter in the plaintext
            char = self.cipher(i) #Cipher the letter
            ciphertext = ciphertext + char #And add it to the ciphertext string

        ciphertext = [ciphertext[i:i+int(self.group)] for i in range(0, len(ciphertext), self.group)] #Split the ciphertext into an
array according to the user input frequency group
        ciphertext = ' '.join(ciphertext) #Join the array to make a single string

        print('\nCiphertext: ', ciphertext, '\n') #Return the split ciphertext to the user

    def generateSettings(self):
        """Generates daily settings"""
        database = data() #Create a data object
        database.exportTable() #Export the daily settings

    def GetUserSettings(self):
        """Choose settings from regular enigma wiring"""

        #               alphabet, notch, turnover, position, ringSetting
        #               ABCDEFGHIJKLMNOPQRSTUVWXYZ
        self._rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', 1, 1) #Create a default rotor I object
        self._rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', 1, 1) #Create a default rotor II object
        self._rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', 1, 1) #Create a default rotor III object
        self._rotor4 = rotor('ESOVPZJAYQUIRHXLNFTGKDCMWB','R','J', 1, 1) #Create a default rotor IV object
        self._rotor5 = rotor('VZBRGITYUPSDNHLXAWMJQOFECK','H','Z', 1, 1) #Create a default rotor V object

        self._UKWA = reflector('AE BJ CM DZ FL GY HX IV KW NR OQ PU ST') #Create a default A reflector object
        self._UKWB = reflector('AY BR CU DH EQ FS GL IP JX KN MO TZ VW') #Create a default B reflector object
        self._UKWC = reflector('AF BV CP DJ EI GO HY KR LZ MX NW QT SU') #Create a default C reflector object

        loopRotors = True
        while loopRotors == True: #While user input is invalid
            rotorL, rotorM, rotorR = input("\nEnter rotor setup: ").upper().split() #Prompt the user to enter the rotor setup
            if rotorL == rotorM or rotorL == rotorR or rotorM == rotorR: #If the user has used the same rotor
                print('Rotors can not be the same. Try again.') #Inform them that they cannot use the same rotors and
prompt again
            else:
                loopRotors = False #Otherwise continue with the program

        reflectorType = input("Enter reflector type: ").upper() #Prompt user to enter reflector type

        loopPlugboard = True
        while loopPlugboard == True: #While user input is invalid
            plugboardPairs = input('\nEnter plugboard pairs: ').upper().strip() #Prompt user to enter plugboard pairs
            stringPairs = plugboardPairs.replace(" ", "") #Remove any spaces

            if len(stringPairs) != 0: #If the user has entered plugboard pairs
                frequencies = collections.Counter(stringPairs) #Count the frequency of each character
                repeated = {}
                for k, v in frequencies.items(): #For every frequency pair
                    if v > 1: #If there is more than one occurrence
                        repeated[k] = v #Add it to the repeated dictionary
                if len(repeated) != 0: #If there are repeats in the repeated dictionary
                    print('Each character  may only connect to another character. Try again.') #Prompt the user to enter the
plugboard pairs again
```

```
                loopPlugboard = True
            else:
                loopPlugboard = False #Otherwise continue with the program
        else:
            loopPlugboard = False #Continue with the program if there is not input for the plugboard pairs

        rotors = {'I':self._rotor1, 'II':self._rotor2, 'III':self._rotor3, 'IV':self._rotor4, 'V':self._rotor5} #Match each rotor type to
their rotor object
        reflectors = {'A':self._UKWA, 'B':self._UKWB, 'C':self._UKWC} #Match each reflector type to their reflector object

        self._rotorL = rotors.get(rotorL) #Assign the corresponding rotor object to the rotor
        self._rotorM = rotors.get(rotorM)
        self._rotorR = rotors.get(rotorR)

        self._UKW = reflectors[reflectorType] #Assign the corresponding reflector object to the reflector

        self._plugboard = plugboard(plugboardPairs) #Assign the corresponding plugboard object to the plugboard

    def help(self):
        """Displays usage instruction for user"""
        print('\n********** USING DEFAULT WIRING **********')
        print('Enter rotor setup as a sequence of 3 rotors using uppercase or lowercase letter separated by a space
between each rotor. Choose from 5 default rotors: I II III IV V.')
        print('Enter reflector type as a single uppercase or lowercase letter. Choose from A B C.')

        print('\nEnter plugboard pairs as pairs of uppercase or lowercase letters. Each letter may only be paired with
one other letter. All letters do not have to be used.')
        print('Enter message as a string of uppercase or lowercase letters. Spaces in the text will be removed.')

        print('\nEnter character grouping as an integer number.')
        print('Enter choice to visualise encryption as a single uppercase or lowercase letter.')


        print('\n\n********** USING CUSTOM WIRING **********')
        print('Enter rotor alphabet as a string of uppercase or lowercase letters. Each letter may only occur once and
every letter must be used.')

        print('\nEnter turnover positions as a sequence of 3 uppercase or lowercase letters separated by a space
between each letter.')
        print('Enter notch positions as a sequence of 3 uppercase or lowercase letters separated by a space between
each letter.')

        print('\nEnter ring settings as a sequence of 3 numbers separated by a space between each number')
        print('Enter rotor starting positions as a sequence of 3 numbers separated by a space between each number')

        print('\nEnter plugboard pairs as pairs of uppercase or lowercase letters. Each letter may only be paired with
one other letter. All letters do not have to be used.')
        print('Enter reflector pairs as pairs of uppercase or lowercase letters. Each letter may only be paired with one
other letter. All letters must be used.')

        print('\nEnter message as a string of uppercase or lowercase letters. Spaces in the text will be removed.')

        print('\nEnter character grouping as an integer number.')
        print('Enter choice to visualise encryption as a single uppercase or lowercase letter.')


        print('\n\n********** USING IMPORTED SETTINGS **********')
        print('Enter name of database as a single string of characters')

        print('\nEnter rotor starting positions as a sequence of 3 numbers separated by a space between each number')
```

```
        print('Enter reflector type as a single uppercase or lowercase letter. Choose from A B C.')

        print('\nEnter message as a string of uppercase or lowercase letters. Spaces in the text will be removed.')

        print('\nEnter character grouping as an integer number.')
        print('Enter choice to visualise encryption as a single uppercase or lowercase letter.')


        print('\n\n********** GENERATING CIPHER TABLE **********')
        print('Enter name of database as a single string of characters.')
        print("Database will be exported to and saved in 'Enigma Settings' folder")
        time.sleep(2)
        print('\n\nPress enter to return to main menu...')
        input("")

    def importSettings(self):
        """Import daily settings from enigma"""

        self.ring1, self.ring2, self.ring3, self.ring4, self.ring5 = 0, 0, 0, 0, 0 #Initialise the ring setting values
        self.start1, self.start2, self.start3, self.start4, self.start5 = 1, 2, 3, 4, 5

        database = data() #Create a data object

        dailySettings = database.readData() #Import the daily settings
        row = dailySettings[0] #Assign the imported data to the row variable

        #FORMAT
        #[(1, 'IV V II', '20 09 23', 'TNUVHCQYOMFDRBAIKZGJSXEPLW', 'nft jlx nzj mbu')]

        rotors = row[1] #Fetch the data at the first index
        rotorL, rotorM, rotorR = rotors.split() #And split it into 3 seperate rotors
        rotorL, rotorM, rotorR = str(rotorL), str(rotorM), str(rotorR) #Ensure they are string variables

        ringSettings = row[2] #Fetch the data at the second index
        ringL, ringM, ringR = ringSettings.split() #And split it into 3 seperate ring positions
        ringL, ringM, ringR = int(ringL), int(ringM), int(ringR) #Ensure they are integer variables

        plugboardPairs = row[3] #Assign the element at the third index to the plugboard pairs
        charGroups = row[4] #Assign the element at the fourth index to the character groups

        startL, startM, startR = input('\nEnter rotor starting positions: ').split() #Prompt the user to enter the rotor starting
positions
        startL, startM, startR = int(startL), int(startM), int(startR) #Ensure they are integer variables

        reflectorType = input("Enter reflector type: ").upper() #Prompt user to enter reflector type

        ring = {'I':'ring1', 'II':'ring2', 'III':'ring3', 'IV':'ring4', 'V':'ring5'} #Match rotor types to string of their ring setting
variables
        start = {'I':'start1', 'II':'start2', 'III':'start3', 'IV':'start4', 'V':'start5'} #Match rotor types to string of their start position
variables

        setLStart = str(start.get(rotorL)) #Get the string of the rotors starting position
        setMStart = str(start.get(rotorM))
        setRStart = str(start.get(rotorR))

        vars(self)[setLStart] = startL #Create a dynamic variable using the string of the starting position and set its value
as the input value for the left rotor starting position
        vars(self)[setMStart] = startM
        vars(self)[setRStart] = startR
```

```
        setLRing = str(ring.get(rotorL)) #Get the string of the rotors ring setting
        setMRing = str(ring.get(rotorM))
        setRRing = str(ring.get(rotorR))

        vars(self)[setLRing] = ringL #Create a dynamic variable using the string of the ring setting and set its value as
the input value for the left rotor ring setting
        vars(self)[setMRing] = ringM
        vars(self)[setRRing] = ringR

        print("\n**********     Imported Settings     **********") #Output the imported settings to the user
        print("Left Rotor:", rotorL + ", Ring position:", str(ringL) + ", Start position:", str(startL))
        print("Middle Rotor:", rotorM + ", Ring position:", str(ringM) + ", Start position:", str(startM))
        print("Right Rotor:", rotorR + ", Ring position:", str(ringR) + ", Start position:", str(startR))
        print("Kenngruppen:", charGroups)

        print("Plugboard:", plugboardPairs)
        print("Reflector type:", reflectorType + '\n')

        #                ABCDEFGHIJKLMNOPQRSTUVWXYZ
        self._rotor1 = rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ','Y','Q', self.start1, self.ring1) #Create a rotor object
using the user input for the starting position and ring setting values
        self._rotor2 = rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE','M','E', self.start2, self.ring2)
        self._rotor3 = rotor('BDFHJLCPRTXVZNYEIWGAKMUSQO','D','V', self.start3, self.ring3)
        self._rotor4 = rotor('ESOVPZJAYQUIRHXLNFTGKDCMWB','R','J', self.start4, self.ring4)
        self._rotor5 = rotor('VZBRGITYUPSDNHLXAWMJQOFECK','H','Z', self.start5, self.ring5)

        self._UKWA = reflector('AE BJ CM DZ FL GY HX IV KW NR OQ PU ST') #Create the default reflector objects
        self._UKWB = reflector('AY BR CU DH EQ FS GL IP JX KN MO TZ VW')
        self._UKWC = reflector('AF BV CP DJ EI GO HY KR LZ MX NW QT SU')

        rotors = {'I':self._rotor1, 'II':self._rotor2, 'III':self._rotor3, 'IV':self._rotor4, 'V':self._rotor5} #Match the rotor types to
their objects
        reflectors = {'A':self._UKWA, 'B':self._UKWB, 'C':self._UKWC} #Match the reflector types to their objects

        self._rotorL = rotors.get(rotorL) #Assign the corresponding rotor object to the rotor
        self._rotorM = rotors.get(rotorM)
        self._rotorR = rotors.get(rotorR)

        self._UKW = reflectors[reflectorType] #Assign the corresponding reflector object to the reflector

        self._plugboard = plugboard(plugboardPairs) #Assign the corresponding plugboard object to the plugboard

    def Run(self):
        choice = ""
        while choice != "6": #While the user has not exited the application
            self.displayMenu() #Display the main menu
            choice = input() #Read in user input
            if choice == "1": #If user has chosen the first option
                self.GetUserSettings() #Run GetUserSettings
                self.encryptMessage() #Then run encryptMessage
            elif choice == "2": #If user has chosen the second option
                self.createCustomWiring() #Run createCustomWiring
                self.encryptMessage() #Run encryptMessage
            elif choice == "3": #If user has chosen the third option
                self.importSettings() #Run importSettings
                self.encryptMessage() #Run encryptMessage
            elif choice == "4": #If user has chosen the fourth option
                self.generateSettings() #Run generateSettings
            elif choice == "5": #If user has chosen the fifth option
                self.help() #Run help
```

```
        elif choice == "6": #If user has chosen the sixth option
            print('\nThank you for using Enigma machine simulator. Goodbye!') #Output an end message
            time.sleep(4) #Wait 4 seconds before exiting the application

    def singleStep(self, rotorL, rotorM, rotorR):
        Rpos = rotorR.GetRotorPosition() #Find the current position of the right rotor
        Rturn = rotorR.GetTurnoverNum() #Find the turnover position of the right rotor

        Rturn += 1 #Increment the right rotors turnover position by one
        if Rpos == Rturn: #If the rotors current position is equal to the incremented turnover position of the right rotor
            if self.visuals == 'Y': #And the user has chosen to see the encryption
                print('Single Step') #Notify them that a single step will occur
            rotorM.incrementRotor() #Increment the middle rotor


def Main():
    """Creates the simulation object and runs the simulation"""
    EnigmaSim = simulation() #Creates the simulation object
    EnigmaSim.Run() #Runs the simulation

#If the module is a standalone program, the Main function will be executed
#Otherwise, it can act as a module in another program
if __name__ == "__main__":
    Main()
```

TESTING

Testing done throughout the development of the program is included in each algorithm's respective section in the documented design element of this document. In this section, we will be carrying out the final tests for each objective point. The evidence will be referenced as a timestamp in a video which can be accessed at https://www.youtube.com/watch?v=2OfyZyNxbCE and is titled 'AQA NEA Testing Section: Enigma machine emulator'.

OBJECTIVES CHECKLIST

Section	Objective	Sub Objective	Evidence
1.1 Exit Option			10:20
1.2 Encode message	1.2.1 User prompted to enter enigma settings		01:43
	1.2.2 User asked if they would like to visualize encryption process		04:32
	1.2.3 User prompted to enter message (*Message may be entered with space characters included for clarity*). Error message displayed when foreign characters are entered as input and user is returned to main menu (*Altered to allow the user to continue upon entering correct input*)		04:32
1.3 Decode message	1.3.1 Similar objectives for '1.2 Encode message'		*Refer to evidence for section 1.2*
	1.3.2 User presented with option to group plaintext		04:32
1.4 Generate cipher table	1.4.1 Create a cipher table for the next 31 days		03:28
	1.4.2 Export cipher table	1.4.2.1 File will be exported using SQLite	*Refer to documented design section: database design - exporting database screenshots* *Refer to timestamp*

			03:05 for database file
		1.4.2.2 'datetime' module will be used to identify time and day of cipher table generation	*Refer to documented design section: database design - exporting database screenshots*
		1.4.2.3 File will be exported to folder for ease of use using the 'os' module	*Refer to documented design section: database design - folder creation screenshots*
1.5 Load Cipher table	1.5.1 Cipher table will be read in	1.5.1.1 Error message will be displayed if there are missing fields	*Unrequired as all entries for the daily settings are automatically generated by the application*
		1.5.1.2 Error message will be displayed if the settings are outdated	*Unrequired as the datetime module automatically selects the daily setting MO 1.4.2.2*
		1.5.1.3 Error message will be displayed if filename is incorrect	04:00
1.6 Guidance	*1.6.1* Provide the user with guidance on how to operate the enigma machine and the format of the input when using the application.		00:22
2.1 Rotor	2.2.1 Rotors must rotate		*Refer to documented design section: rotation algorithm - rotating the rotors screenshots* *Refer to timestamp 05:13 for visual representation of rotation*

	2.2.2 Stepping mechanism on rotors must step on following values respectively	Rotor I must step from Q	05:13
		Rotor II must step from E	05:38
		Rotor III must step from V	06:05
		Rotor IV must step from J	06:27
		Rotor V must step from Z	07:35
	2.2.3 Option to switch between rotors	2.2.3.1 Error message will be displayed if the user tries to use two of the same rotors	00:53
	2.2.4 Option to use 3 of the 5 rotors during encryption		01:44
	2.2.5 External alphabet on rotors must be matched to internal rotor wiring		*Refer to documented design section: rotation algorithm - rotating the rotors screenshots*
	2.2.6 Internal wiring must adjust to match external alphabet when a rotor rotates		*Refer to documented design section: rotation algorithm - rotating the rotors screenshots*
	2.2.7 Single step and double step instances must be recognized and completed (Single step evidence provided in section 2.2.2, this is the double step evidence)		08:05
2.2 Reflector	2.2.1 Reflector must ‘reflect’ the input and output the substitute character		*Refer to documented design section: rotation algorithm - ciphering the initial user input screenshots*
	2.2.2 Option to use a A-reflector		01:47

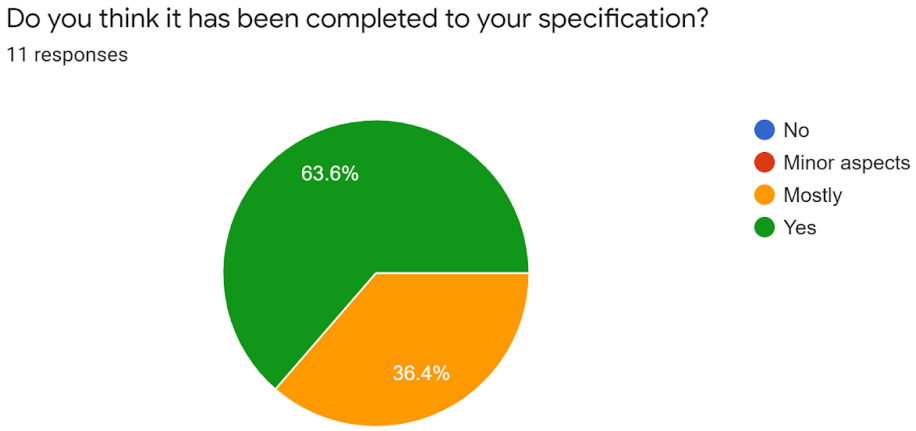
	2.2.3 Option to use a B-reflector		04:20
	2.2.4 Option to use a C-reflector		05:00
2.3 Plugboard	2.3.1 Option to connect to different letters	2.3.1.1 Prevent user from encrypting to the same character	01:06

EVALUATION

COMMENTING ON THE EFFECTIVENESS OF THE SOLUTION

After completing the first draft of the program, I sent a copy of the program along with the document (copy provided in Appendix B) so that the Bletchley park representatives could review the solution and test it for any issues. I also forwarded a copy to the history teacher interviewed in the research section of this document.

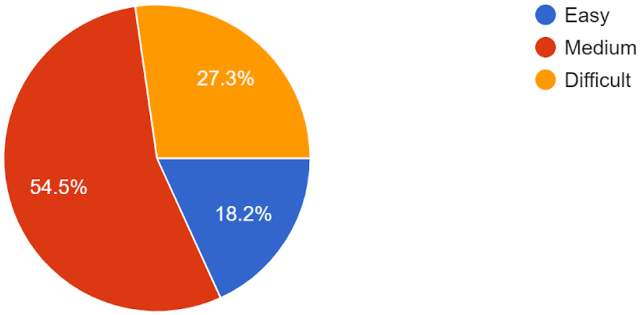
Below is a chart showing the satisfaction of the users in the ability of the program to accommodate their requirements. We can see that most of the users felt that it satisfied their requirements.



Some users cited errors, explained in further detail below, which have now been resolved. Most of the users felt that using the enigma was at a medium difficulty and that they were able to work it out in the end.

How easy was it to use the enigma?

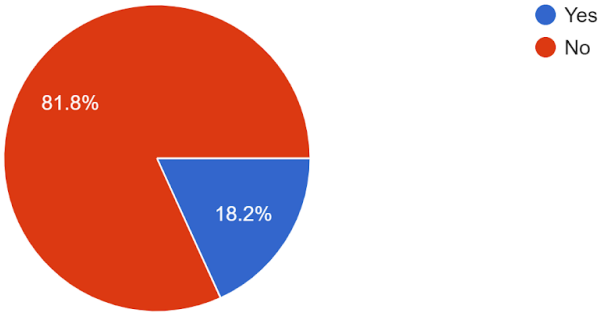
11 responses



Those that did not feel that it was easy to use the enigma cited syntax errors causing the program to crash. This issue has also been elaborated below and is now resolved.

Did you encounter any bugs?

11 responses



Some users encountered bugs which have also been elaborated in further detail below.

Overall there was positive feedback as the program fulfilled the users requirements but issues that were raised were:

1. Changing the ring settings did not alter the program in any way.

   The reason that this occured is because I have not been able to figure out what the purpose of the ring setting actually is and the support that I have received from Bletchley park has not improved my understanding in any way. However, I have tailored the program to take the inputs of the ring setting so that when I have a better understanding, I will be able to create an algorithm to satisfy this function in the future with minimal change to the program.

2. User should be allowed to enter text with spaces so they can read message

   This option has been added into the program and any spaces entered by the user are stripped by the program before the message is encoded. The user then has the option of splitting the encoded ciphertext into groups of their own preference.

3. Databases are unorganised

   The program has been adapted to include some code to generate a folder called 'Enigma Settings' where the databases are automatically exported to for ease of use.

4. Issue with rotor 5 not stepping

   The users reported an issue with the stepping mechanism for rotor 5. In order for the rotors to step, the turnover value is incremented by one so that upon identifying that specific position, the program will know to step at the next key press. As the turnover character for rotor 5 is Z, its index position would be 25, the incremented character is 26 which lies outside of the index of the range A - Z which is 0 - 25 so the rotors did not step from the transition from Z to A. This issue was resolved by using mod to find the remainder in order to continue indexing from the beginning of the alphabet so the turnover position for rotor 5 would be 0 instead of 26.

5. Instructions unclear

   Some users had difficulty using the enigma machine and did not quite understand the format of the input that the application could accommodate. This issue was resolved by creating a 'guidance' option to assist the user with the use of the application.

6. Getter methods

   Some of the methods in the classes start with 'Get' but are not getter methods. I did not know that this type of method existed before I programmed the majority of the application and changing this now would only cause more confusion as the documented design and testing for each algorithm uses the 'Get' prefix.

To conclude, the minor bugs and other issues have now been resolved and the program is complete to the users specification. The only issue lies with the function of the ring settings and I have not been able to work out how it works preventing me from using it in the program. However, the program takes and deals with the input for the ring settings but that is the limit of its functionality until I have a better understanding of it's function at a later stage.

APPENDIX

APPENDIX A

WORKINGS OF THE ENIGMA

The process of encrypting using the enigma machine can be broken down into the following steps:

1. User presses the key on the enigma machine
2. Rightmost rotor rotates
3. Signal passes through plugboard which substitutes character
4. Output passes through the 3 rotors
5. Output is substituted through the reflector and passed back though the hole system
6. Reflected signal passes through the plugboard again
7. Signal lights up on the light board
8. Operator writes down key of illuminated bulb

OPERATION OF THE ENIGMA

In order to ensure that the correct settings were used for communications between the German military, the operator of the enigma machine was provided with a codebook which listed the settings for that particular day. These settings were outdated by midnight, when the next day's settings came into effect and so the operators had to be diligent in order to prevent any confusion or miscommunication between their forces.

The codebook listed five parameters for the setup of the enigma:

1. Date
2. Rotor order
3. Ring setting
4. Plugboard settings
5. Discriminant

In order to demonstrate the operation of the enigma, we can assume the following settings as the current days settings:

Date	01 January 2020
Rotor order	V I III
Ring settings	04 15 21
Plugboard settings	EJ OY IV AQ KW FX MT PS LU BD
Discriminant	BLA

Using the example above, the operator would adjust the ring position of rotor V until the 04 position was lined up with the zero position on the rotor (A), repeating this process for each rotor so that the I and III rotors have their respective ring position of 15 and 21, aligned with the A position on each rotor.

These rotors would then be slotted into the enigma machine, starting from the right, in the order, V I III where the operator would use a lever to complete the circuit connection, keeping the levers in place.

The plugboard pairs are then read off the codebook as the operator will connect each of the corresponding pairs to each other to add another layer of security.

The discriminant is then used to enter into the enigma machine at the beginning of a message used to indicate the ‘degree of secrecy’ of the message or to distinguish ‘one type or section of traffic from another’.

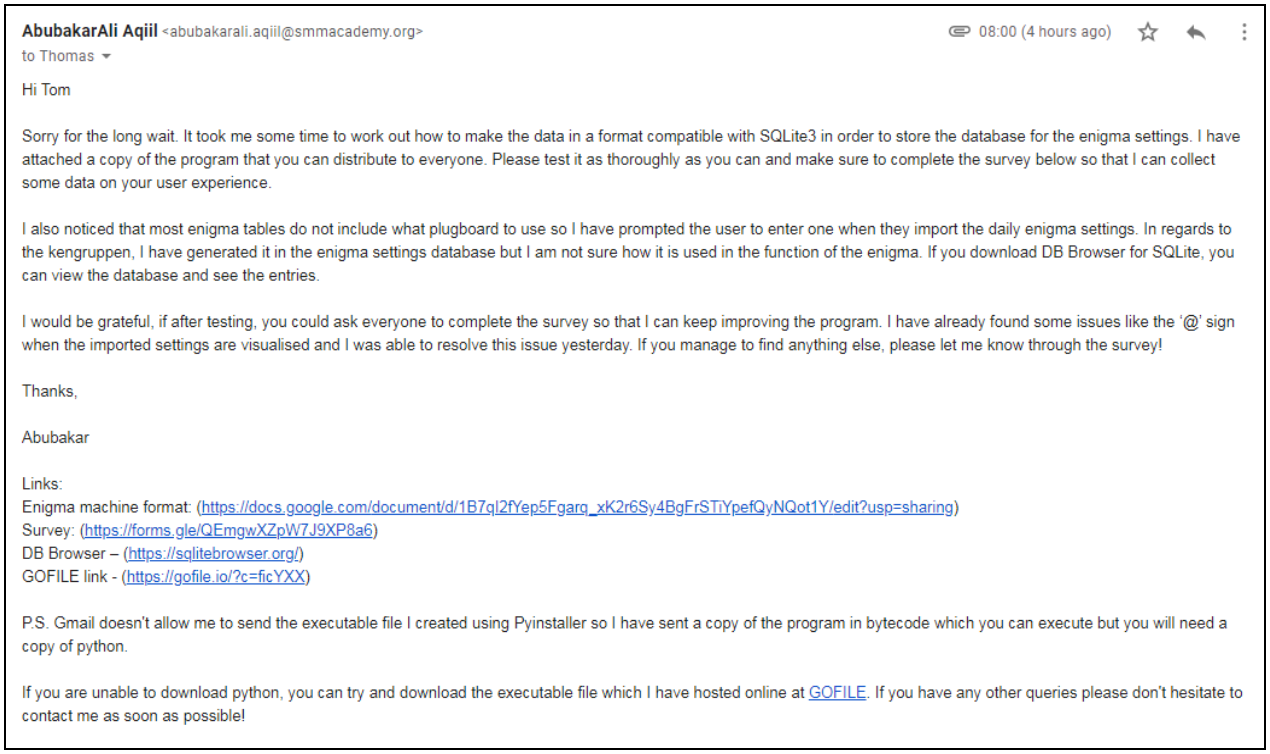
REFERENCES

More detail on the operation and workings of the enigma can be found at the web addresses below.

Codes and ciphers: https://www.codesandciphers.org.uk/enigma/enigma1.htm
Wikipedia: https://en.wikipedia.org/wiki/Enigma_machine
Bletchley Park: https://bletchleypark.org.uk/our-story/the-challenge/enigma

APPENDIX B

CORRESPONDENCE TO CLIENT



AbubakarAli Aqiil <abubakarali.aqiil@smmacademy.org> 08:00 (4 hours ago)
to Thomas

Hi Tom

Sorry for the long wait. It took me some time to work out how to make the data in a format compatible with SQLite3 in order to store the database for the enigma settings. I have attached a copy of the program that you can distribute to everyone. Please test it as thoroughly as you can and make sure to complete the survey below so that I can collect some data on your user experience.

I also noticed that most enigma tables do not include what plugboard to use so I have prompted the user to enter one when they import the daily enigma settings. In regards to the kengruppen, I have generated it in the enigma settings database but I am not sure how it is used in the function of the enigma. If you download DB Browser for SQLite, you can view the database and see the entries.

I would be grateful, if after testing, you could ask everyone to complete the survey so that I can keep improving the program. I have already found some issues like the '@' sign when the imported settings are visualised and I was able to resolve this issue yesterday. If you manage to find anything else, please let me know through the survey!

Thanks,

Abubakar

Links:
Enigma machine format: (https://docs.google.com/document/d/1B7qI2fYep5Fgarq_xK2r6Sy4BgFrSTiYpefQyNQot1Y/edit?usp=sharing)
Survey: (https://forms.gle/QEmgwXZpW7J9XP8a6)
DB Browser – (https://sqlitebrowser.org/)
GOFILE link - (https://gofile.io/?c=ficYXX)

P.S. Gmail doesn't allow me to send the executable file I created using Pyinstaller so I have sent a copy of the program in bytecode which you can execute but you will need a copy of python.

If you are unable to download python, you can try and download the executable file which I have hosted online at GOFILE. If you have any other queries please don't hesitate to contact me as soon as possible!