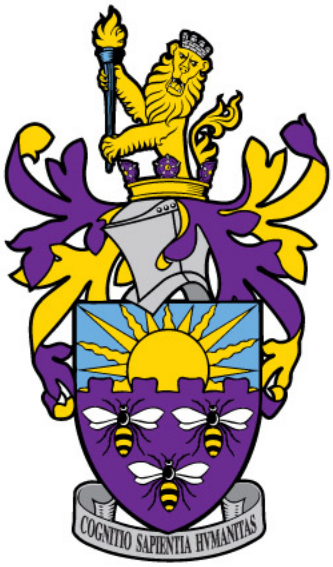
The University of Manchester
Department of Computer Science
Project Report 2023



Enigma Machine Simulator

Author: Holly Foster

Supervisor: Thomas Carroll

Abstract

The Enigma I machine is a cipher encryption device, used by the German military during World War II to encrypt and decrypt secret messages. Its complexity and use of changing key settings made the cipher initially appear unbreakable. However, the cracking of the code by the Allies played a significant role in their eventual WWII victory. The Enigma machine has had a large impact on modern cryptography techniques and continues to have a lasting legacy in the field of computer science.
The aim of this project was to create an Enigma machine simulator. It is a tool for users to encrypt and decrypt their own messages, that simulates the functionality of the Enigma I machine. It replicates the machine's encryption process, using the plugboard, three rotors and reflector components, in order to encrypt a plaintext letter into ciphertext, and vice-versa. This report describes the details of how Java and Swing were used to create a graphical user interface application that supports the full component configuration of the original machine.

Acknowledgements

I would like to thank my supervisor, Tom Carroll, for his continuous support and encouragement throughout the entirety of this project. I am grateful for the guidance he gave me and his faith in my abilities.

I would also like to thank my incredible friends and family, who have given me unwavering love and support during every step of my degree.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Enigma machine is one of the most infamous and intriguing encryption devices in history. Developed in Germany during the early 20th century, the machine was used heavily by the Germans during World War II to encode their military communications. Symmetric encryption was used to encode plaintext messages into scrambled, unreadable ciphertext using a complex system of components. The resulting ciphertext could only be deciphered by someone with the exact same machine and settings. Despite being designed with a seemingly unbreakable code, the Enigma was ultimately cracked by the Allied forces, playing a significant role in the Allies victory of World War II.

The goal of this project is to create an Enigma machine simulator. Overall, this project aims to:

- Accurately simulate the functionality of the Enigma machine.
- Support the full configuration of the machine.
- Present the simulator in the form of a graphical user interface (GUI).

With regards to the first point above, the simulator should be capable of taking input from the user and giving the correct encrypted output, given the data goes through the precise sequence of the plugboard, rotor, and reflector components. These components will mirror that of the true Enigma machine. Again, just like the original, the simulator should encrypt and decrypt text in the same way, meaning that to decrypt the ciphertext back to the original plaintext message, the user should be able to use the identical settings.

The full configuration of the machine means the user can choose the rotor placement, rotor configurations and reflector configuration, as well as up to ten plugboard pairs. Further details of such are discussed in Chapter 2. By including all setting options, the simulator will provide a stronger data encryption method and a more accurate representation of the Enigma machine.

This projects aims to create a simulator that allows users to understand the entire encryption process, despite their level of prior knowledge of the original Enigma machine. For each character in the inputted text, the simulator should display the result of each stage the letter goes through during encryption. Consequently, this tool can be used to learn about the Enigma machine, in addition to it's practical use of encrypting messages.

1.1 Motivation

The Enigma machine truly changed the encryption game, as it was and still stands to be a pioneering example of cryptography. Learning about how it was designed and operated can provide us with a deeper understanding of encryption, and allows us to apply this knowledge to modern-day technology and encryption techniques. As well as demonstrating the importance of secure communication and highlighting the risks involved with inadequate encryption methods, the machine has a strong historical significance. As it played a critical role in World War II, understanding it's functionality gives us an insight into that period of time in history and how technology has evolved ever since. In today's world, encryption is an essential tool for ensuring privacy and security. With the population's increasing digital footprint, protecting sensitive and personal data is crucial for many.

1.2 Objectives

Further to the criteria discussed in Chapter 1, several objectives were set for this project. In order to achieve the main aims, these objectives needed to be successfully met.

Firstly, deep research of the machine itself is conducive to having a clear understanding of the technical process behind the encryption operation.

Secondly, the specific settings and configurations needed to be incorporated into the machine so that the simulator is historically accurate. This includes the rotor, reflector and plugboard options. Additionally, these features need to be adjustable, so the user can vary the settings and experiment with different configurations to see how they affect the encryption process.

This functionality then needs to be implemented into an intuitive user interface (UI), by which the user can input their messages and receive the encrypted outputs. Each feature needs its own UI component along with self explanatory labels of such. The simulator should be interactive and provide an engaging user experience.

To evaluate the success of the project, multiple methods will be used. These include comparing results to other existing Enigma machine simulators, unit tests for back-end functions and gathering front-end feedback from users via a questionnaire. A detailed discussion of this is provided in Chapter 5.

1.3 Report structure

This report is divided into five main sections. The first, Chapter 2, describes the background of encryption, the history of the Enigma machine and an explanation of it's behaviour. Next, there are two chapters, Chapter 3 and Chapter 4, detailing the design of the project and the decisions made during the implementation of the simulator, respectively. The following Chapter 5 demonstrates the results of the project and analyses the evaluation of its success. Finally, this report concludes with a critical personal reflection of the project achievements in Chapter 6.

Chapter 2

Background

This chapter will cover the background information required for this project, in particular it will look at how encryption works, the make up of the Enigma machine and how it functions. It will also discuss related work and existing versions of the simulator similar to the one this project aims to create.

2.1 Encryption

Encryption is the process of converting plaintext (human-readable unencrypted data) into ciphertext (incomprehensible encrypted data), as a means of making the data unreadable to any unauthorised parties. It is a two-way function by which the original message is uncovered by the recipient by decoding the resulting ciphertext back to plaintext, using a designated key. Complex mathematical algorithms and an encryption key encode the data, scrambling the original message into a unique, complicated pattern of bits. A matching decryption key then allows for the ciphertext to be converted back to plaintext, once again being readable to the intended recipient.

There are three stages during which data can be encrypted; at rest, in transit or in use. Encryption at rest keeps the key used to encrypt and decrypt data safe, for files stored on hard drives, cloud storage and smartphones etc., with an authentication system protecting the key. When data is transmitted from one network to another, encryption protects data during movement. Routers have WiFi Protected Access (WPA) encryption enabled, with protocols such as the Secure File Transfer Protocol (SFTP) and Transport Layer Security (TLS) used for transmission over the internet [21]. While data is being viewed or edited, mobile and cloud applications protect data by ensuring the most secure encryption is adopted within its source code.

Encryption methods can either be symmetric or asymmetric [3]. Asymmetric encryption uses a public key to encrypt the data, and a different private key to decrypt it. This report will focus on symmetric encryption as that is the form of the encryption the Enigma machine uses.

2.1.1 Symmetric encryption

Symmetric encryption is a type of encryption where the same key is used to both encrypt and decrypt the data. The key is a secret piece of information that only the sender and the receiver know and use, in order to transform plaintext into ciphertext and vice versa.

The steps involved in the process of symmetric encryption are:

1. Key Generation: A random key is generated and agreed on by the sender and trusted recipient.
2. Encryption: The plaintext is encoded into ciphertext using the key and a cryptographic algorithm.
3. Transmission: The encrypted data is sent via a secure channel to the recipient.
4. Decryption: The receiver decrypts the ciphertext using the same key and algorithm, revealing the original plaintext.

These stages are also shown in Figure 2.1.

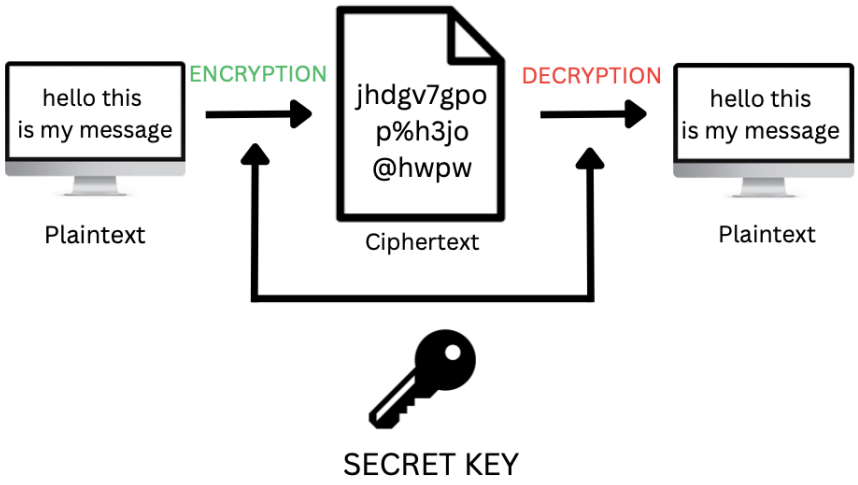


Figure 2.1: Symmetric encryption process (Adapted from [7]).

Some common symmetric encryption algorithms include the Advanced Encryption Standard (AES), Data Encryption Standard (DES), and the Blowfish algorithm [7]. These algorithms differ in factors such as their key size, block size, and encryption speed. Symmetric encryption is widely used in applications that require secure communication and data storage, such as online banking, e-commerce, and email. The main limitation of this method is that sharing a single key increases the risk of key compromise.

2.1.2 Substitution encryption

Substitution encryption is a type of encryption in which each letter or character in a message is replaced with a different letter or character, according to some predefined set of rules. Typically messages are encoded using a substitution cipher, which maps each letter of the alphabet to a different one.

A common example of such is the Caeser cipher. A Caeser cipher shifts each letter in the plaintext by a certain number of places down the alphabet [14]. So, if the shift value is 2, for example, the letter A would be replaced with the letter C, H with J, and so on for the entire alphabet. Figure 2.2 below shows a Caeser cipher with a shift of 3.

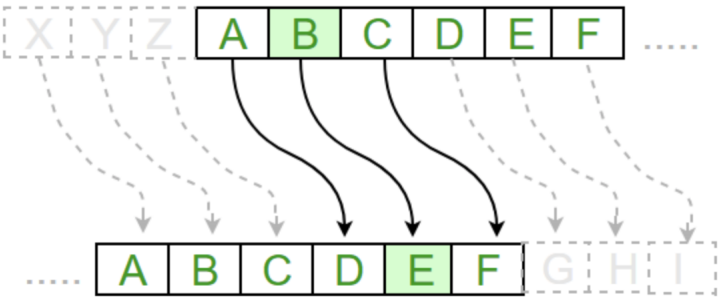


Figure 2.2: Caeser cipher with shift 3 [14].

Substitution ciphers are usually relatively easy to break, by using frequency analysis (analysing the number of letter occurrences to determine the most common letters) or other cryptanalysis techniques. As a result, they are not generally considered to be very secure on their own. However, they can be used as one component of a more complex encryption scheme, or just as a means of slightly obscuring low-risk casual communications.

The Enigma machine uses a form substitution encryption, but is a lot more powerful than just a simple Caeser cipher. The reason for this is explained in much more detail in Section 3.

2.2 The Enigma Machine

2.2.1 History

The Enigma machine was first invented just after the end of World War I, in 1918, by the German electrical engineer Arthur Scherbius. Scherbius then started a company in 1919 called Scherbius & Ritter, and filed a patent for the machine to produce and market the 'Enigma' product [23]. Initially, the Enigma machine was intended just for commercial use, marketed as a tool for businesses and governments to keep their communications secret. However, shortly the German military recognised the machine's potential and adopted it to encrypt their own military correspondence.

Multiple versions of the Enigma machine were made, with the design evolving each time and the newer models advancing to be more complex and consequently harder to crack. By the start of World War II, the German Army (Heer) and Air Force (Luftwaffe) had developed and started using their own version, officially known as Enigma I [24]. Enigma I was an electromechanical cipher machine developed by Chriffriermaschinen AG in Berlin, based off the chassis of the commercial machine. The first prototype was built in 1927, and was ready for deployment and military use by 1932. Later in the war, it was also used by the German Navy, where it became known as the M1, M2, and M3, with additional components being introduced.

This report will focus on Enigma I, as it was the main version of the machine used during WWII.

2.2.2 Components

The main components of the Enigma I machine were

- A keyboard (Tastatur);
- A plugboard (Steckerbrett);
- A set of five rotors (Rotoren);
- A reflector (Umkehrwalze (UKW));
- A lampboard (Lampenfeld);

The keyboard was the input device used to enter the plaintext message that wanted to be encrypted. It consisted of 26 keys, marked A-Z, one for each letter of the alphabet. The Enigma keyboard was arranged in a QWERTZ configuration with electromechanical key switches wired electrically to the plugboard. The QWERTZ layout is used in German-speaking countries, where the Enigma machine was primarily used. It's a variation of the QWERTY layout used in English-speaking countries that we are more familiar with.

The plugboard was placed at the front of the machine below the keys. It is a patch panel that allowed the operator to swap pairs of letters before they entered the rotors. It contained a set of sockets and dual-wired cables that permitted the variable wiring of connected pairs of letters [10]. For example, if one end of a cable was plugged into the socket labelled *B*, with the other end in the socket labelled *K*, *B* and *K* would be a steckered pair. Hence, when the letter *B* was pressed on the keyboard, the signal was diverted to *K* before being encrypted by the rotors, and vice-versa. Likewise, once the letter had gone through the main rotor unit, the pair was swapped again afterwards. This worked as each letter on the plugboard had two jacks and inserting both plugs at either end of the crosswired cable switched the connections of the two letters. Although there could be up to thirteen pairs, only ten were used usually [24]. The plugboard allowed for an extra layer of complexity and increased the security of the cipher by creating a large number of possible substitutions.

From a set of five rotors, three were used at any given one time in the machine. They could be arranged in any order, and were ordered such that 26 input points and 26 output points were positioned on alternate faces of the discs [32]. These points corresponded to the 26 A-Z letters of the alphabet. The operator could also specify the initial positions of the rotors (the letter the rotor starts at), as there were 26 serrations around the periphery of each rotor. With an alphabet along the rims, the initial orientations of the rotors were displayed at the top of the machine. The rotors rotated by the means of a movable ring on each rotor, placed to its left using a *notch* [25]. The wiring for the rotor wheels is defined as a translation of the input (right) to the output (left). Each rotor provides a different encoding scheme, as each wheel has a different wiring of the alphabet. These rotor wheels also had different corresponding notch turnovers. Table 2.2.2 shows the details of the five rotors in Enigma I [25][11].

Rotor	ABCDEFGHIJKLMNOPQRSTUVWXYZ	Notch Turnover
I	EKMFLGDQVZNTOWYHXUSPAIBRCJ	Q
II	AJDKSIRUXBLHWTMCQGZNPYFVOE	E
III	BDFHJLCPRTXVZNYEIWGAKMUSQO	V
IV	ESOVPZJAYQUIRHXLNFTGKDCMWB	J
V	VZBRGITYUPSDNHLXAWMJQOFECK	Z

Table 2.1: Enigma I rotor configurations.

Every keystroke caused one or more rotors to step forward by one twenty-sixth of a full rotation. The rotation occurred before the electrical connections were made, i.e. before the lampboard lit up. This meant the cryptographic substitution was different at each new rotor position, giving rise to a stronger polyalphabetic substitution cipher. The rotor on the right-hand side stepped once with each key press, whilst the other two rotors stepped less frequently. The point at which the other rotors advanced was the *turnover* point. If the rotor was at its particular *notch* position, the rotor mechanism would align, and the rotor to its left would be triggered to rotate (turn over). For example, if Rotor I steps from *Q* to *R*, the next rotor is advanced. So, for the single notch in the right-hand rotor, the middle rotor would step once for every 26 steps of the right-hand one. Similarly the middle rotor would only engage the left-hand rotor when it reached its notch position, once every 26 steps [11]. A feature known as *double-stepping* was where the middle rotor stepped twice on two successive key presses if the left-hand rotor also made a step. Figure 2.3 shows an example of this, where when the left wheel makes a single step, the middle wheel takes a 2nd step (even though it has not completed another full rotation/at the notch position) [26]. This *double-stepping* anomaly made the Enigma machine even more difficult to crack.

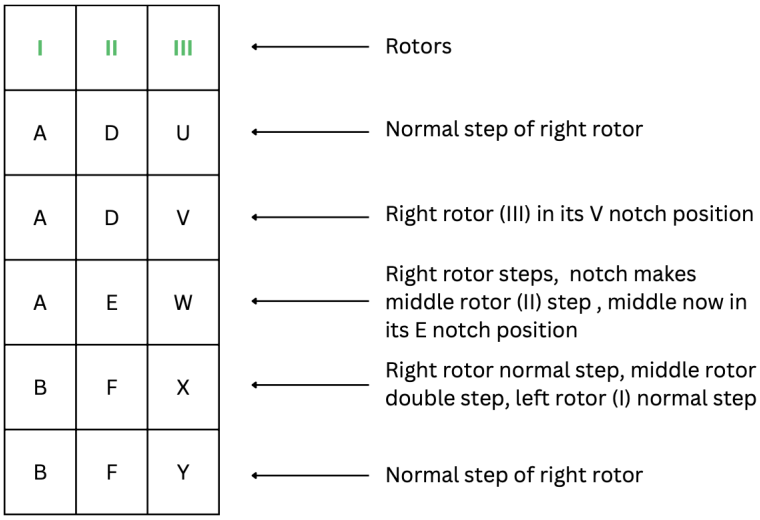


Figure 2.3: Double stepping example.

The ring setting was a stationary component used to change the internal wiring positions relative to the rotor [11]. Whilst the notch and alphabet ring were fixed to the rotor, the ring setting could rotate the wiring. Therefore by adjusting the ring setting, the electrical contacts on the rotor were shifted relative to the electrical contacts on the ring, changing the encryption of each letter. Again, this added an extra layer of complexity to the encryption process.

The reflector was like a half-rotor fixed at the end, that reflected the electrical signal back through the rotors via a different route. It was a large plate with a series of electrical contacts on each side. The contacts on one side of the reflector were wired to corresponding contacts on the other side, but in a scrambled order. When a letter was typed on the Enigma machine keyboard and passed through the rotors, it would eventually reach the reflector. The reflector would then cause the current to be reflected back through the rotors in a different path, effectively 'reflecting' the input letter to a new output letter, creating a different encryption path for each keystroke. This component ensured the encryption was reciprocal; so two identically configured machines could both encrypt and decrypt messages. However, this also meant that the Enigma could never encrypt a letter to itself - a big flaw in the machine's design [2]. Table 2.2.2 shows the configuration of three reflectors. UKW-A was primarily used before the war, UKW-B during and UKW-C in the later part of the war [25].

Reflector	**ABCDEFGHIJKLMNOPQRSTUVWXYZ**
UKW-A	EJMZALYXVBWFCRQUONTSPIKHGD
UKW-B	YRUHQSLDPXNGOKMIEBFZCWVJAT
UKW-C	FVPJIAOYEDRZXWGCTKUQSBNMHL

Table 2.2: Enigma I reflector configurations.

The lampboard displayed the encrypted messaged, by lighting up the corresponding letters based on the output from the encrypting components. It was a simple field of 26 lightbulbs, one for each letter of the alphabet. The electrical current returning from its path after its journey through the plugboard, rotors, reflector and back through the rotors and plugboard again, illuminates the connected bulb, indicating the enciphered letter.

These five components came together to create the machine, working in unison to encrypt messages. Below, Figure 2.4 shows an image of the Enigma I along with its labelled components. Section 3 discusses the specific encryption process in depth.

2.2.3 Cracking the Enigma code

In 1932, Polish mathematician and cryptologist Marian Rejewski managed to achieve a breakthrough in decrypting the Enigma cipher [33][10]. The Polish Cipher Bureau recruited 20 mathematics students from Poznan University to work on breaking the code [32]. Initially this seemed like an impossible task. They started with research on the commercial machines and quickly found success, with the help of French spies to obtain information on the German cipher materials (daily keys, plugboard settings). Using the theory of permutations and flaws in the encipherment procedures, as mentioned above, they developed techniques to break the message keys [6] [33]. Along with Jerzy Rozycki and Henryk Zygalski, Rejewski was able to build an 'Enigma double' and start to read the German's messages [32].

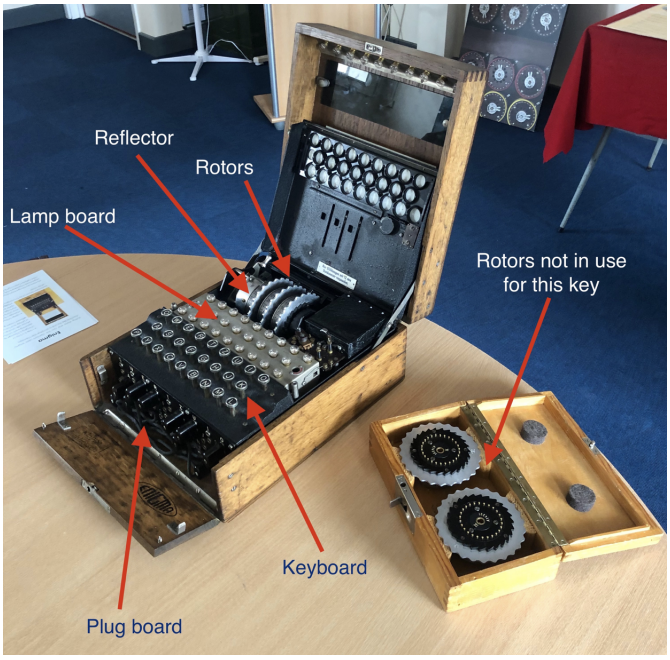


Figure 2.4: Labelled Enigma machine [2].

As mentioned previously, a flaw in the Enigma code was that a letter couldn't be encoded as itself. This gave the codebreakers a starting point to decrypt the messages, as if they could guess a word that is likely to appear in the message, they could use this information to begin cracking down the walls of the code. At the start of each message, the Germans would send a weather report, and usually end the message with the phrase "Heil Hitler" [28]. These common phrases gave the decoders something to look out for. They could compare a such phrase to letters in the code and if the letters matched with that of the phrase, they knew that part of the message did not contain the phrase. So, by the process of elimination, there was a good starting point for progress on the decryptions.

With the prospect of war ahead, the Poles passed on their knowledge to the British in 1939. An Enigma Research Section was set up, consisting of former WWI codebreaker Dilly Knox, Tony Kendrick, Peter Twinn, Gordon Welchman and Alan Turing. Together, they worked in a stable yard at Bletchley Park in Buckinghamshire, England [31]. Here, Turing and Welchman invented the Bombe machine, which tried to determine the Enigma rotor and plugboard settings given a coded messaged. The machine was essentially 36 Enigma simulators wired together, with the rotors represented by three drums (one for each rotor). The top drum would simulate the left-hand rotor, the middle the middle rotor and the bottom drum would imitate the machine's right-hand rotor. The drums would rotate and attempt a new configuration, giving the $26 \times 26 \times 26 = 17,576$ positions of the 3 rotor Enigma machine. For each turn of the drums, the Bombe would then take a guess at one of the plugboard settings, for example "H is connected to F". Then, the machine would run through and determine what all the other letters on the plugboard must be set to. If there was a contradiction, for instance it deduced that actually "H is connected to A", then it knows H is not connected to F. As the other letter mappings were based off of this "H to F" assumption, all of those combinations are now invalid. This was helpful as the Bombe then knew not to waste time checking those combinations again in the future. Instead, the machine shifted the drum positions, choose a new plugboard pair assumption and repeated this process until the test did not lead to a contradiction, and a candidate configuration solution could be recorded. As more configurations were tested, the list of potential solutions could be narrowed down [17].

In January of 1940, the British broke the first wartime Enigma messages and continued to decrypt Enigma traffic for the remainder of the war [5][31]. Consequently, this intelligence lead to crucial information which largely aided the Allied victory of World War II.

2.3 Related work

There are numerous Enigma machine simulator applications which have already been created in a variety of manners. The purpose of this section is to explore some of the existing applications, in order to help inform the design of this project and to use as a comparison later on in the evaluation (5).

Figure 2.5: Button based existing simulator. [1]

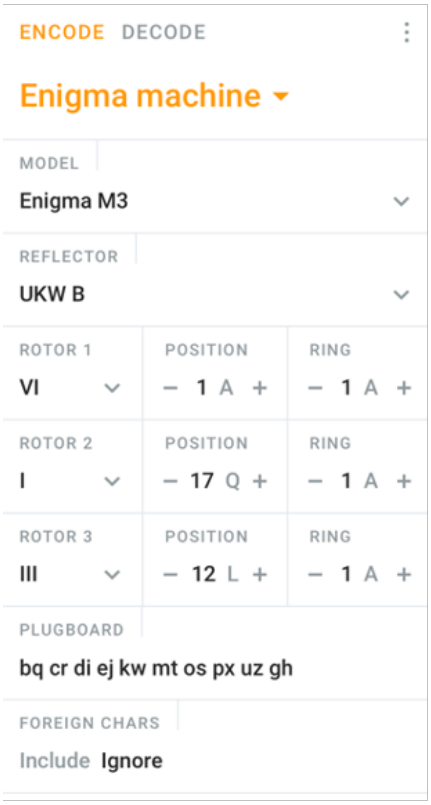


Figure 2.6: Text based existing simulator. [12]

Figure 2.7: Virtual based existing simulator. [8]

Figure 2.8: Examples of existing products

Figure 2.8 shows three very different approaches to the solution. Figure 2.5 uses buttons to create plugboard pairs and to input text on the keyboard [1]. They also give the option to see all of the encryption steps for each letter in the message. Figure 2.6 is a more text based version, with the use of dropdown menus and text boxes to enter input and display output [12]. This application supports multiple models of the machine and can include foreign characters, which is out of the scope for this project. Figure 2.7 is a 3D virtual simulation which enables the user to interact with every part of the machine, including open the box [8]. Again, this level of interactivity is beyond the limits of this project. However, elements from each application will be useful as inspiration.

Chapter 3

Design

This chapter will focus on the design aspect of the project. It will include an analysis of the project requirements and the mathematics behind the main encryption process, along with the designs for the structure of the back-end and front-end components of the simulator.

3.1 Requirements

The requirements for the simulator that emulates the functionality of the Enigma I machine are the following:

- It should have a method by which the user can input text.
- It should have all 5 rotors and their initial positions, 3 reflectors, and allow up to 10 plugboard pairs.
- The user should be able to customise all settings.
- The resulting ciphertext should be displayed to the user.
- The encryption should be symmetric - the machine can encrypt and decrypt using the same settings.
- A letter should never encrypt to itself.

These requirements will be referred back to throughout the implementation (Chapter 4) and used to inform the evaluation (Chapter 5) later on.

3.2 Encryption process

Before the operator could input their plaintext to be encrypted, the machine needed to be configured. Such set up includes choosing three out of the five available rotors to use, the order of the rotors (which would be left, middle, right), the initial starting position of each, and which letters should be wired as pairs on the plugboard (max 10 pairs).

With the set up complete, encryption can begin.

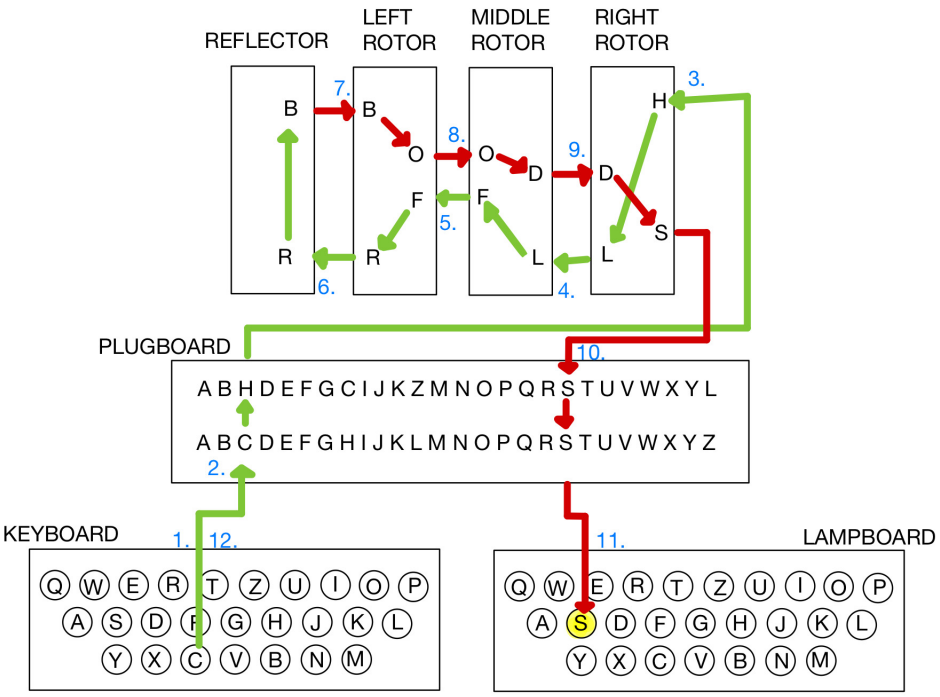


Figure 3.1: Path taken by a letter through an Enigma machine as it is encrypted.

Figure 3.1 demonstrates the path of a single letter through the encryption process. Below, the numbered list corresponds to the labels in Figure 3.1 and explains each stage of the diagram:

1. The plaintext message is typed in on the Enigma keyboard, one letter at a time.
2. The electrical signal from the keyboard passes through the plugboard.
3. The right-hand rotor rotates one position and the electrical signal passes through it.
4. The signal then passes through the middle rotor.
5. The signal then passes through the left-hand rotor.
6. The signal then reaches the reflector, which causes the current to be reflected back through the rotors in a different path.
7. The reflected signal passes back through the left rotor.
8. The signal then passes back through the middle rotor.
9. The signal then passes back through the right rotor.
10. The signal passes back through the plugboard.
11. The final output letter is displayed on the lamp board.
12. The user then types in the next letter of the message and the process repeats.

3.2.1 Mathematical analysis

As the Enigma machine was renowned for its complexity at the time, this section of the report explores the cryptographic strength of the machine by investigating how many configurations were possible with the Enigma I specifications.

In the first slot there are 5 rotors to choose from, in the second slot there are 4 and in the third, 3 left to pick from. This gives

$$5 \times 4 \times 3 = 60 \tag{3.1}$$

ways to configure the five rotors.
There are 26 starting positions for each rotor, so there are

$$26 \times 26 \times 26 = 17,576 \tag{3.2}$$

options for the initial rotor position configurations.

As the plugboard allowed for 10 pairs, there are 20 out of 26 letters involved in the plugboard. Additionally, the order of the 10 pairs does not matter, nor does the order of the letters within the pair. This results in a total of

$$\frac{26!}{6! \times 10! \times 2^{10}} = 150,738,274,937,250 \tag{3.3}$$

combinations yielded by the plugboard [32].
Putting all of the components back together, there are

$$60 \times 17,576 \times 150,738,274,937,250 = 158,962,555,217,826,360,000 \tag{3.4}$$

number of ways to set the Enigma 1 machine [17].

Theory of Permutations

The transformation of each letter can also be expressed mathematically as a product of permutations. Marian Rejewski's paper [33] from 1980 outlines how the theory of permutations was used in some aspects of breaking the Enigma cipher. A simplified version is explained here.

Definition 3.2.1. A permutation is a mathematical technique that determines the number of possible arrangements in a set, when the order of the arrangements matters.

Here let P denote the plugboard transformation, U denote the reflector, and L, M, R denote the left, middle and right rotors respectively.
The encryption E can then be expressed as

$$E = PRMLUL^{-1}M^{-1}R^{-1}P^{-1}. \tag{3.5}$$

However, the rotors turn, changing the transformation. These rotations can be represented as a cyclic permutation, p, mapping each letter to another.

Definition 3.2.2. A cyclic permutation is a permutation which shifts all elements of a set by a fixed offset, with the elements shifted off the end inserted back at the beginning. [35]

Let n, j, k denote the number of positions R, M, L rotate, respectively [10].
The encryption transformation can then be described as

$$E = P(p^nRp^{-n})(p^jMp^{-j})(p^kKp^{-k})U(p^kL^{-1}p^{-k})(p^jM^{-1}p^{-j})(p^nR^{-1}p^{-n})P^{-1}. \tag{3.6}$$

3.3 System architecture

The design for the back-end of the software development of the project is based on the logic of the encryption process just described in Section 3.2, as the aim here is to replicate the exact functionality.

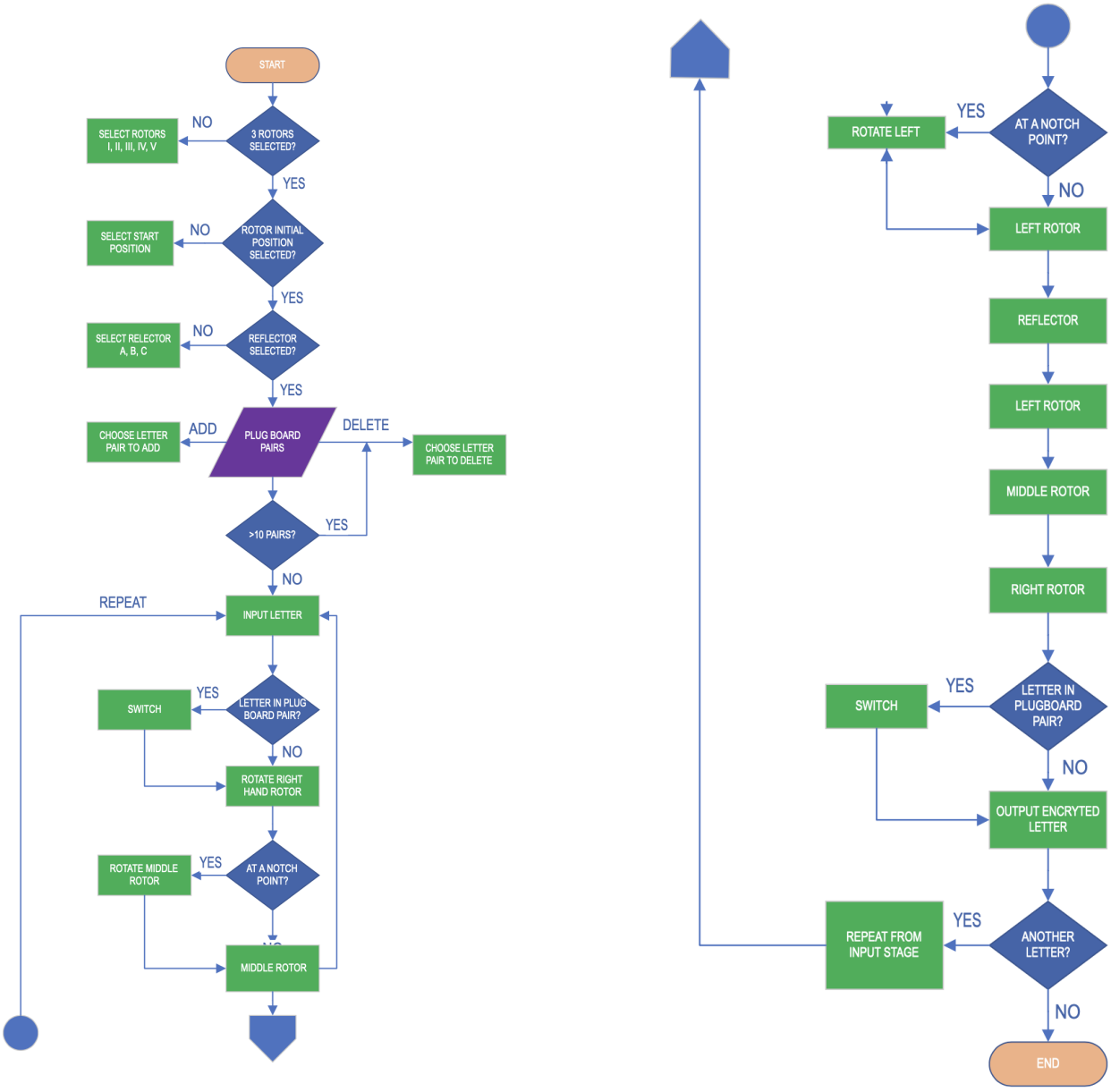


Figure 3.2: System architecture flowchart.

Figure 3.2 provides a high-level overview of the steps involved in operating the simulator. The structure demonstrates the order in which the steps should be executed, and how they are connected to and affect each other. The decision points indicate where the process branches depend on a certain input value, showing how the simulator will handle different scenarios.

The user will be asked to select their rotor, reflector and plugboard options and be prompted to input their plaintext message to be encrypted. The simulator will allow a maximum of 10 plugboard pairs. Each letter will then go through the encryption process, based off the given configurations. The changing current rotor position should be shown for the user to see as the rotations occur. After the encryption is complete, the ciphertext will be displayed back to the user. Depending on whether the user would like to see the individual steps of the process, these details may also be presented to the user. The user will then have the option to change the settings or encrypt/decrypt another message. To decrypt ciphertext back to plaintext, the user will need to revert the machine back to the initial settings that particular message was encrypted with.

As the machine utilises both numbers and letters in its functionality, Table 3.1 shows the mapping of each, along with the relative ASCII codes that will be needed during implementation. It is worth noting that these values would change if the simulator was based on different languages - this project focuses on the English A-Z uppercase alphabet.

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
ASCII	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90

Table 3.1: A-Z uppercase letter mappings.

3.4 User interface

In order for the user to actually engage with the system, a user interface is required. For the beginning stages of development, a terminal based approach would suffice, just as a means of simply taking input and giving output. From here, the logic of the system and correctness of the encryption itself can be tested.

Figure 3.3 shows a mock up wireframe design for the front-end aspect of the project.

This initial design encompasses all aspects of the machine that need to be included. Labels for each component are clear so the simulator can be used without any in depth prior knowledge being required - the design aims to be as intuitive as possible so it is accessible to all users. Dropdown menus for the rotor selection, reflector selection, and rotor initial position selection ensure the user does not try to input an invalid option, and so they know what choices are actually available to select. The plugboard section allows for the addition and deletion of pairs, by the means of simply selecting the pair and then clicking the corresponding action button. The text box console area will show the individual stages of the letter encryption. As this will be a long, ever growing body of text, the scroll bar enables the box to remain compact, regardless of how long the contents becomes.

The purpose of these designs is to act as a guideline for the implementation of the code itself. They give a rough idea of where to begin, how to logically progress through the development and provide a visual initial goal to aim for.

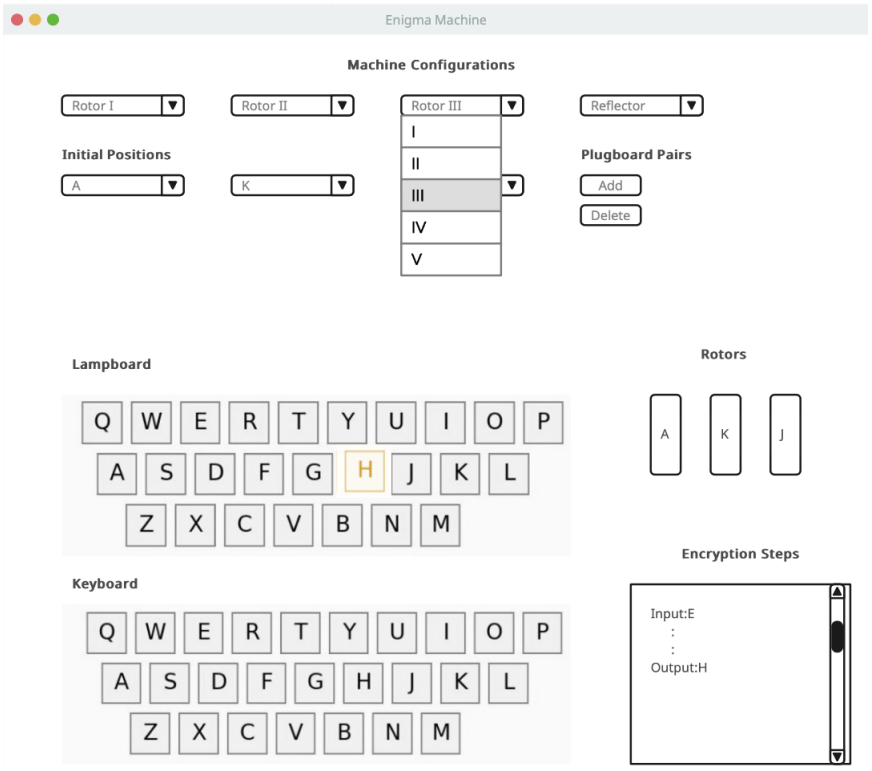


Figure 3.3: UI wireframe mock up.

Chapter 4

Implementation

Based on the information provided in the previous sections, this chapter will discuss the details of the implementation. The development consists of the implementation of the individual components, and then of each part working together in order to achieve the end result. Here, there will be an explanation of the steps taken to create the simulator, along with the challenges that arose during development.

This simulator was created using Java Version 17 for the back end, and Swing for the GUI. Java is a versatile high-level, object-oriented programming language and Swing provides a rich set of customisable GUI components implemented in Java. Although JavaFX is a more modern popular alternative, the main focus of this project was the encryption itself, so Swing was still an adequate option.

4.1 Rotors

As the main intricacies of the machine's encryption happens in the rotor unit, this class was particularly important and was the first focus of implementation. As a result, it was also the most challenging to get right. The functionality of the rotors was built up incrementally, starting with the plaintext letter just going through the rotors, to them then rotating, and then to changing the initial starting positions of each.

First, the Rotor objects were initialised using the alphabet and notch values from Table 2.2.2 and assigned the *I, II, III, IV, V* names accordingly. A `switch` statement was used to determine which three rotors had been selected. The selected rotors were then set in position using `setRotorAtPosition`, to determine which was the left-hand ([2]), middle ([1]) and right-hand ([0]) rotor. The notch points however have been hard-coded in. For encryption, before the reflector, the `convertForward` method was used to convert the character forwards through the rotor. This method would take in the current index position of the character, using `toIndex` and `currentPosition` to determine where in the rotor the character was. The new resulting index is returned to use in the next encryption step. Similarly, after the reflector, `convertBackward` converted the character in the same fashion, taking in the current index position and returning the new one.

As the rotating of the right-hand rotor occurred before encryption started, `moveRotors(j=0)` is called first. The `moveRotors` method utilises the `advance` and `atNotchPosition` methods. `advance` increments the `currentPosition` of the rotor by 1, essentially 'rotating' the rotor by 1/26 of a turn. `atNotchPosition` is a boolean flag that checks if the `currentPosition` is equal to the `currentNotch` of the given rotor. If it is equal, `moveRotors` is called recursively in itself, this time with j=j+1, i.e., the rotor to its left. This recursive element means each rotor can be rotated accordingly, if it is at the turnover position. Listing 4.1 provides the code for the `moveRotors` method. The default rotor initial positions are taken from the config file values and converted from a letter, to its equivalent position in the rotor using the `setToChar` method, which also used `setPosition` to assign the `currentPosition`. The encryption and rotations would then begin from these given starting values.

```
1     /**
2      * Allows the user to advance rotor at position j
3      *
4      * @param rotors rotors array
5      * @param j  instance to advance
6      */
7     public void moveRotors(final Rotor[] rotors, final int j) {
8         if (0 > j || j > rotors.length - 1)
9             throw new RuntimeException("com.fozzie.Rotor cannot be advanced at
    position " + j + "Size is out of scope of array length: " + rotors.length);
10        final Rotor r = rotors[j];
11        r.advance();
12        if (r.atNotchPosition() && rotors.length > j + 1) {
13            this.moveRotors(rotors, j + 1);
14        }
15    }
```

Listing 4.1: Code for moveRotors method.

For the UI aspect of the rotor selection, three `JComboBoxs` were used to represent the three rotors, populated with the *I, II, III, IV, V* rotor object options from the `PossibleRotors` list. The default selection comes from the initial choice in the config file and are displayed in the combo box using `setSelectedItem`. Each combo box then had its own `ActionListener`, where any changes in rotor selection were detected. If the user had picked a new rotor, this selection would update in the back-end using `getSelectedItem` and `setRotorAtPosition`. This would mean that on the next round of encryption, the new rotor option would be used instead. Dynamically changing the list of rotor options so the same rotor could not be used twice was extremely difficult. The decision to omit this feature from the simulator came from time constraints and checking another existing application (2.5), and discovering they allowed the same rotor to be used multiple times too. As the rotors were physical objects in the original Enigma machine, this wouldn't have been possible. The current rotor positions were displayed using `JLabels`, updating after each rotation using `setText` and the `String.valueOf` of the `getCurrentPosition` of each rotor, to display the value as a letter to the user, rather than just the index number.

```
        sdRot1.addChangeListener(changeEvent -> {
            enigmaMachine.getRotors()[2].setPosition(sdRot1.getValue());
            String alphaValue = String.valueOf(Rotor.toChar(sdRot1.getValue()));
            tfRotVal1.setText(String.valueOf(alphaValue));
            Rotor r = enigmaMachine.getRotors()[2];
            r.setToChar(Rotor.toChar(sdRot1.getValue()));
            setRotorLabelText(enigmaMachine.getRotors());
        });
```

Listing 4.2: Code for rotor initial position ChangeListener.

For the front-end aspect of the initial positions, `JSliders` were used. Although in the initial design (Figure 3.3) this was going to be another combo box, the boxes were extremely long as they had all 26 letters in and made the application look messy. Instead, the slider allowed the values 0-25 and the user to could move the pointer along to select a position. The character value of this position was displayed in a `JTextField` above. A `ChangeListener` for each slider was used to detect a new starting position. The slider value was then used in the `setPosition` of the given rotor, using `getValue`. This `getValue` was turned into its character equivalent with `toChar`, along with `setText` to display the chosen letter on the UI. An example of this is shown in Listing 4.2. During implementation, a bug arose whereby one letter on the slider would be wrong, for example a seemingly random 'Z' in the middle of the alphabet. In the end, this error was a quick fix and was just down to confusing the 0-25 and 1-26 alphabet/index scale.

4.2 Reflector

With the rotors functioning, the next step was to implement the reflector component. The reflector is essentially an extra rotor but with some different defaults and without the rotations. Therefore, the Reflector class `extends` the Rotor class. The Reflector objects were initialised using the values from Table 2.2.2 with the corresponding *A,B,C* names. The Rotor's `setPosition` method was over-written using `@Override` to get the new position from the reflector. During encryption, the Reflector would utilise the Rotor's `convertForward` method to use the current index position and obtain the new resulting index. This index would be the one used to then go back through the rotors, after the 'reflection'. This element of the simulator was relatively straight forward to implement as the base methods and logic were already in place from the rotors.

In a similar fashion on the UI, the 'Reflector Selection' component was a `JComboBox`, populated with the elements from the `PossibleReflectors List` of Reflector objects. The initial value of the combo box was that from the `selectedReflector` from the `EnigmaConfig` file. The combo box's `ActionListener` was then used to detect any changes in selection, setting the reflector in the back-end with `getSelectedItem` and `setReflector(Reflector)`, and updating the display of the combo box with `setSelectedItem` (shown in Listing 4.3). Again, this was much simpler to include than the rotor combo boxes, as only one reflector is used at a time.

```
        refBox.addActionListener(e -> {
            enigmaMachine.setReflector((Reflector) refBox.getSelectedItem());
```

Listing 4.3: Code for Reflector ActionListener.

4.3 Plugboard

The next part of the implementation was to add the plugboard element. As the plugboard works in pairs of letters, the most efficient way to store and access these pairs was to use hashmaps. The `HashMap` class of the Java collections framework provides the functionality of the hash table data structure. It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map [29]. Two hashmaps were used, one for `pairs` and one for `revPairs`, in order to allow for the letter swap in both directions. The `BuildPairs` method would take in two letter characters, first check if these characters were already in the hashmap pairs, and if not, add the characters to both maps. Listing 4.4 shows the `BuildPairs` method code.

```
1     /**
2      * Building each pair, for forwards and backwards encryption.
3      *
4      * @param c1 first character in the pair
5      * @param c2 second character in the pair
6      */
7     public static void BuildPairs(final char c1, final char c2) {
8         if (PlugboardPairs.pairs.containsKey(c1) || PlugboardPairs.revPairs.
    containsKey(c2)) {
9             throw new IllegalArgumentException("Character already in map");
10        }
11        PlugboardPairs.pairs.put(c1, c2);
12        PlugboardPairs.revPairs.put(c2, c1);
13    }
```

Listing 4.4: Code for BuildPairs method.

During encryption then, the `MapPlugboardPairs` method would perform the switch. If the given character was contained in one of the pairs in the map, the method would return its matching letter. This returned letter would then be the one to go through the rotors. Similarly, on the way backwards, the matching letter from the found pair would be the one that got displayed in the ciphertext.

For the front end, a `JList` and `DefaultListModel` were used for the pairs itself, and `JButtons` used to add and delete pairs. Using an `ActionListener` on the buttons to determine if they had been clicked, the user could select their plugboard pairs. To add, `JOptionPanes` were used to prompt the user to select a letter to include in the pair. The list given was the values from the `ArrayList`, `getRemainingAlphabetChars`. This would ensure only letters not already in a plugboard pair could be selected. These chosen letters would then build a plugboard pair and be displayed in the list. As mentioned previously, the Germans only used ten pairs. Hence, when the size of the list model reached 10, and the user tried to add another, `ShowMessageDialog` would inform them that they have reached the maximum number of pairs. To delete a pair, the pair could be selected in the UI and be removed by clicking the delete button. The `ActionListener` would then call the `RemovePairs` method where the pair would be taken out of the hash maps, and removed from the `List Model`. As a result, these deleted letters would then go back into the `getRemainingAlphabetChars`, available to be used again in a new pair.

4.4 Keyboard

Until the encryption components were more or less working correctly, the input was just hard-coded in. The next step in the implementation was to allow the user to type in their own message. As shown in Figure 3.3, the keyboard component was intended to be similar to a classic typewriter, closer to the style used in the original. This would mean each letter 'key' would need to be its own separate button, and then arranged in the QWERTZ layout. In the first iteration, the buttons worked, but made the UI look clumsy, as the columns in the `mainPanel` were all thrown off. Also, when manually testing the simulator, entering a message via the buttons was slow and tedious, especially for anything more than a few words. Upon reflection, the design was changed to the user being able to type via their device's keyboard. The `KeyboardFocusManager` class from Java's Abstract Window Toolkit (AWT) API was used, with the `dispatchKeyEvent` method. This method is called by the current `KeyboardFocusManager` requesting that the `KeyEventDispatcher` dispatches the specified event on its behalf. The `KeyEvent` could then be used to check which key had been pressed by the user. Code for the first part of implementing this component is shown in Listing 4.5. The `getKeyChar` method then determines the value of the keystroke. After checking if the inputted key was a letter, this value can then be sent to the `EncryptAndDisplay` method for handling the encryption process of the given character and displaying the results.

```
1        KeyboardFocusManager.getCurrentKeyboardFocusManager()
2                .addKeyEventDispatcher(new KeyEventDispatcher() {
3                    @Override
4                    public boolean dispatchKeyEvent(KeyEvent e) {
5                                ⋮
6        });
```

Listing 4.5: Code for using the Java KeyboardFocusManager.

The method of checking the key code of the key pressed was also used in order to handle spaces and back spaces in the message. As each key has an assigned value by the class, for example, a space has key code 32, it meant it was possible to include these features in the input. However, if a letter was deleted in the message, it would disappear from the text field, but the rotors wouldn't reverse backwards. Although back spaces wouldn't be used with the original machine, it's a very useful feature of today's keyboards, so was still included. Additionally, punctuation was filtered out before encryption started, as well as transforming all letters to uppercase, for uniformity. For this, `isLetter`, `isUpperCase` and `toUpperCase` were used, respectively. Numbers were also filtered out, as with the original machine the Germans would just write out any numbers in their word form (seven not 7).

In terms of the GUI, this input area was just a `JTextField` component labelled 'Plaintext'. The contents of the text field was updated upon each key press, using `setText` and the `KeyEvent` data. To give the simulator a more 'authentic' look and feel in terms of the text, the `American Typewriter` font, provided by Swing, was used. As the original machine used a typewriter style keyboard, this element added a nice touch to the modern keyboard component of the simulator.

4.5 Lampboard

Originally, the lampboard component was going to be an arrangement of individual images, representing a keyboard. The idea behind this was that when the resulting encryption of a letter was found, the corresponding letter 'lightbulb' would 'light up', i.e. the image would change. However during implementation, similar to with the keyboard, trying to fit that many images together on the console proved difficult as they pushed all the other UI components off screen. Also, as the user can type very fast, the images barely had time to change before moving on to the next letter. To resolve this issue, the results of the encryption were displayed in a `JTextField` using `setText`, after checking if the inputted keypress value was in fact a letter. This way, spaces could still be included in the message, as well as accounting for back space deletions in the text fields, as mentioned above.

```
1         var result = enigmaMachine.encrypt(c);
2         resultChar = result.getResult();
3                      ⋮
4         if ((e.getKeyCode() > 64 && e.getKeyCode() < 91) || (e.getKeyCode() > 96 &&
      e.getKeyCode() < 123) || e.getKeyCode() == 8 || e.getKeyCode() == 32) {
5             if (!fromArr) {
6                 tfInput.setText(tfInput.getText() + e.getKeyChar());
7             }
8             tfResult.setText(tfResult.getText() + resultChar);
9             return;
10        };
```

Listing 4.6: Code for displaying resulting ciphertext.

Listing 4.6 shows how the Java `KeyEvent` class was used to determine which key had been pressed by the user, just as with the keyboard. Here, `fromArr` is a boolean flag to check if the key is a letter. This `getKeyCode` method also enabled 'copy and paste' to work in the text boxes, as it could check if Ctrl+C or Ctrl+V was being pressed. This was particularly useful when testing the symmetry of the machine, as the ciphertext could be copied, the machine set back to its initial settings, and then pasted back into the plaintext field to check the message decrypted back to the original.

4.6 Encryption steps

These components then all came together to use in the main `EnigmaMachine` class. It has a default constructor with 3 rotors. The machine was first initialised using the contents of the config file from Section 4.7, and contains the primary `encrypt` method. The Listing B.1 for this method is in the Appendix B, as it a crucial part of the code, but it just follows the same steps from Figure 3.1. In this class the individual steps of each stage of encryption for a letter are also managed. The `EnigmaResult` method contains the `List` for the steps and the final `String` result. The character value of the current index is added to the `steps` list, along with the 'stage' the character is at, continually throughout the `encrypt` process. `toChar` was used to convert the 'index % 26' number into the readable corresponding letter value. The modulus (%) is used to compute the remainder of the index number, so that if it was larger than 26 originally, the modulus would be less than or equal to, and therefore ensures it corresponds to a letter. An example of this code is shown in Listing 4.7.

```
1         index = this.rotors[0].convertForward(this.rotors[0], index);
2         steps.add("Rotor 1: " + Rotor.toChar(index % 26));
```

Listing 4.7: Code for adding encryption step to list.

For every element in the list, the string was appended to a `JTextArea` within a `JScrollPane` in the UI. This would happen for every letter being encrypted, so the user could view the changes to the letter at each stage in the 'wirings console', for all in the message. As mentioned in the design, the scroll bar ensured the information remained compact and tidy on the UI, as the contents is very large for a long message.

4.7 Config

Once the simulator was basically built, an `EnigmaConfig` class and `config.properties` file were created. The aim of this config file was to simplify the complex set up of the machine's configurations, and allow the user to have a sort of 'default' set-up, that could also be changed. Using the `FileWriter` from the `java.io` package, the `generateProps` method creates and stores the config file. Within the file, the properties of the simulators components are given using `setProperty`. This initialises the settings for the Enigma machine with the values provided. `readProps` utilises the `FileReader` to load and read the file contents. From here, `getProperty` retrieves the value of the objects. This data is used in both the back and front-end initialisation of the simulator.

The `checkAndGenerate` method checks to see if the config file already exists in the user's directory. If not, it calls `generateProps` to write the file with the given properties, or if it already exists, just goes straight to calling `readProps` to read the file and retrieve the property data. If the user wants to change the property values, they can just open the config file, type new values and save the changes. Upon running the simulator, the set up will have changed accordingly. Figure 4.1 shows the editable `config.properties` file.

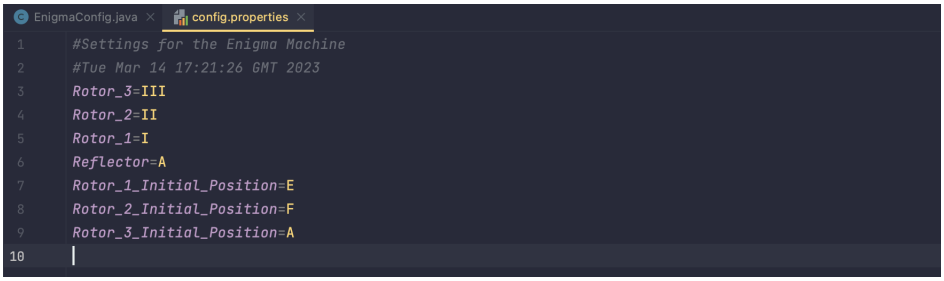


Figure 4.1: Open config.properties file.

This was extremely useful because while testing, instead of spending time changing the settings of each component every time before encrypting, the config file could be changed just once. It also means the user does not have to remember the settings they were using. Moreover, it ensures there are no 'hard-coded' values within the application, giving full flexibility of the system for the user.

4.8 Unit tests

To test the back-end logic and functionality of the simulator, JUnit tests were used. Here tests methods are written, which call the actual methods to be tested. A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system [34]. Test-driven development was mainly used to ensure the logic remained intact throughout and also aided in making the problem solving process more manageable. The test case verifies the behaviour of the code by asserting the return value against the expected value, given the parameters passed, using `Assertions.assertEquals`. If the 'expected' value matched the resulting 'return' value, the test passed. For example, one test, `testPlugboardPair()`, checked that if the pair 'X, B' is built, 'b' is given if 'x' is inputted. In a similar fashion, the `toChar` method is tested by ensuring 'A' is given as a result of 'toChar(0)'. `Assertions.assertNotEquals` is used in `testNotch` to check that the previous position and new position of the middle rotor are different, as if the right rotor is at a notch point, it should trigger a rotation. This example is shown in Listing 4.8.

```
1     @Test
2     public void testNotch(){
3         EnigmaConfig.checkAndGenerate();
4         EnigmaMachine enigmaMachine = new EnigmaMachine(3);
5         enigmaMachine.getRotors()[0].setToChar('Q');
6         int pos = enigmaMachine.getRotors()[1].getCurrentPosition();
7         enigmaMachine.moveRotors(enigmaMachine.getRotors(), 0);
8         int newPos = enigmaMachine.getRotors()[1].getCurrentPosition();
9         Assertions.assertNotEquals(pos, newPos);
10    }
```

Listing 4.8: Code for testNotch unit test.

The tests cover the main methods used in the simulator, with emphasis on the Rotor class methods as it is harder to see manually if the rotations of the rotors are actually working correctly. The complete list of unit tests is given in Listing B.2 in Appendix B and the results of such are discussed in Section 5.2. It is worth noting here that the tests were based off of the original config file settings - if the default values were changed, some of the tests would fail due to the return value being dependent on the exact rotor selections and initial positions.

Chapter 5

Results and Evaluation

This section discusses the various methods by which this project has been evaluated and the results of such evaluations. The back-end aspect of the project has mainly been evaluated using automated and manual testing, along with some comparisons to already existing Enigma machine simulators. The front-end has been evaluated primarily with a user questionnaire and further manual testing.

5.1 Final product

Figure 5.1: Screenshot of working simulator.

Figure 5.1 shows a screenshot of the final artifact, in use. Additional screenshots of the completed simulator demonstrating various features can be found in the Appendix A.

5.2 Automated testing

As described in Section 4.8, unit tests were used to evaluate the correctness of the logic for this project. These tests were able to be run and added to periodically throughout the development of the project.

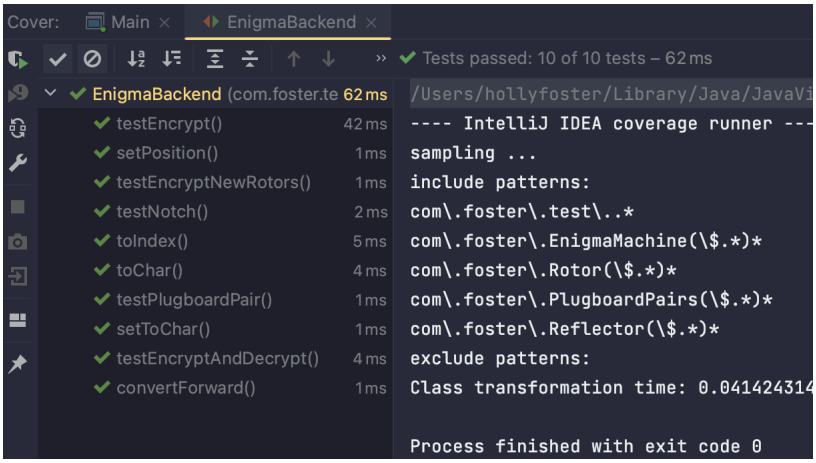


Figure 5.2: Passing unit tests.

Figure 5.2 demonstrates the list of unit tests used, intended to test multiple aspects of the simulator's functionality. As shown, the test suite passes all ten tests. Testing the most important methods within the system in isolation meant that there was more confidence the components would work correctly as part of the bigger main encryption process. For example, the encryption appeared as though it was working correctly, however, the `setToChar` test was failing. With around 14 usages of this method, the error was causing significant changes to the resulting ciphertext. Upon fixing the code in order for the test to pass and rerunning, it was clear to see various components of the simulator working differently, but now correctly. The tests were not a definitive way of ensuring correctness, but helped a great deal.

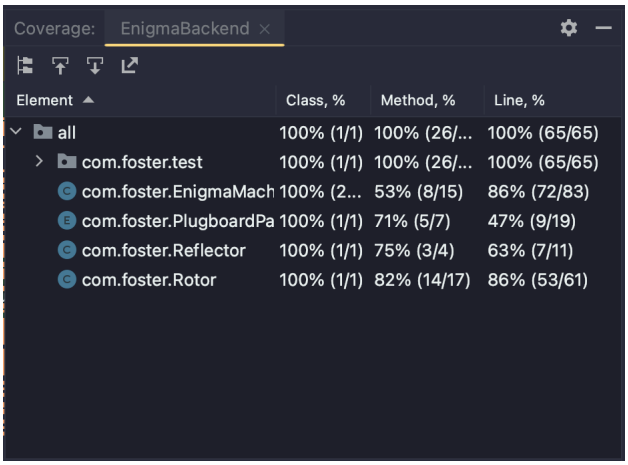


Figure 5.3: Coverage of unit tests.

Figure 5.3 shows the code coverage of the tests. On average, there was a coverage of 70.25% across the code base, excluding files such as the class for exceptions and main method. The optimal test coverage rate is between 70-80% [4], so this test suite is definitely on the lower end of that scale. This suggests a few more test cases would have preferably been included, potentially in the `EnigmaMachine` class as this had the lowest individual coverage. However, the `Rotor` class had the highest coverage of 82%. As this class handled the main stage of encryption, and the Enigma's signature feature, this is a satisfactory result.

```
    @Test
    public void testEncryptAndDecrypt(){
        EnigmaConfig.checkAndGenerate();
        String testString = "The Quick Brown Fox Jumps Over The Lazy Dog.";
        StringBuilder sbEncrypt = new StringBuilder();
        StringBuilder sbDecrypt = new StringBuilder();
        EnigmaMachine enigmaMachine = new EnigmaMachine(3);
        enigmaMachine.getRotors()[0].setToChar(EnigmaConfig.Rotor1Initial);
        enigmaMachine.getRotors()[1].setToChar(EnigmaConfig.Rotor2Initial);
        enigmaMachine.getRotors()[2].setToChar(EnigmaConfig.Rotor3Initial);
        for (char c : testString.toCharArray()) {
        sbEncrypt.append(enigmaMachine.encrypt(c).getResult());
        }
        enigmaMachine.getRotors()[0].setToChar(EnigmaConfig.Rotor1Initial);
        enigmaMachine.getRotors()[1].setToChar(EnigmaConfig.Rotor2Initial);
        enigmaMachine.getRotors()[2].setToChar(EnigmaConfig.Rotor3Initial);
        for (char c : sbEncrypt.toString().toCharArray()) {
        sbDecrypt.append(enigmaMachine.encrypt(c).getResult());
        }
        Assertions.assertEquals(testString, sbDecrypt.toString());
    }
```

Listing 5.1: Code for unit test to check encyrption is symmetric

The `testEncryptAndDecrypt` test checked the symmetry of the machine. As discussed previously, the machine should encrypt and then decrypt the message back, using the same settings. Code listing 5.1 shows how this unit test ensured that the phrase "The Quick Brown Fox Jumps Over The Lazy Dog." decrypted back to the same phrase. This also double checks that punctuation was handled correctly, and that the indexing was also correct, as every letter in the alphabet is used. Figure A.9 in Appendix A also shows an example of the machine's symmetry.

These tests provided a sense of confidence about the quality of the system during implementation. They particularly aided in the debugging process amid implementation, as it as easier to detect the impact of any changes in the code, and 'step-through' where the errors were originated from. The limitations of the unit tests are that they cannot test non-functional attributes such as usability, integration, the overall performance of the system, etc.. The following sections aim to evaluate such attributes.

5.3 Manual testing

Manual testing was used heavily throughout this project, to test both the back-end and front-end of the simulator. In terms of the back-end, it was easier to write out the rotations on paper and use an almost pseudocode like approach to determine the encryption outcome. The way the rotors worked was particularly difficult to understand, and so drawing diagrams and flowcharts on a whiteboard was helpful. For example, it was easier to check the index numbers and letter positions in a rotor by writing out the configuration, counting by hand, and then comparing to what the code outputted. This also made it easier to see which letter the rotor should 'rotate' to next, on paper, and compare with the system afterwards, as opposed to just relying on the code statements. Additionally, manual testing could determine whether the machine ever encrypted a letter to itself. Figure 5.4 shows an example of the simulator successfully never encrypting a letter back to itself.

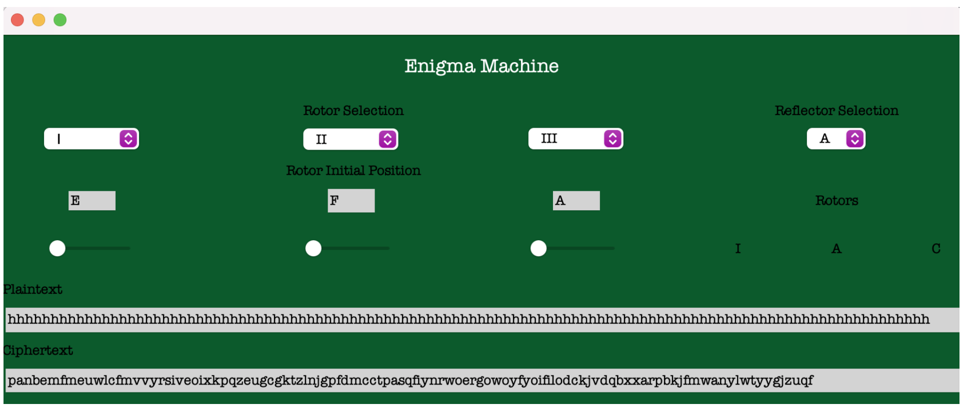


Figure 5.4: Example of a letter never encrypting to itself.

Manual testing proved very effect during the testing of the UI. It was clear on screen where values in the back-end were going wrong, or being displayed incorrectly, as the wrong text or labels were obvious. For example, for the bug mentioned in Section 4.1, the defect in the slider values were only noticeable after taking the time to drag the slider across all values and double checking the result. By doing this, the error of one incorrect letter was found and able to be fixed accordingly. Likewise, the Swing GUI components were able to be adjusted through visually testing the interface - properties such as padding values and colour options were hard to gauge until the application was up and running. In terms of aesthetics, the UI did not give the 'typewriter' style keyboard desired. This would have made the simulator look more 'authentic', but the keyboard version used in the final product is much more efficient.

5.4 User questionnaire

To help test the usability of the simulator and gain some qualitative feedback, a questionnaire was sent out to five respondents. The respondents were given the opportunity to use the simulator prior to completing the survey, and were also given a screenshot image on the questionnaire itself, to refer back to.

When asked at the start about their prior knowledge of the Enigma machine, 40% of respondents said they had 'None'. The last question asked the users if their understanding of the machine had improved since using the simulator, 100% of respondents said 'Yes'. Although the aim of this project was not to be an education tool, it's nice to see the simulator makes sense even to those who have no background knowledge in the subject.

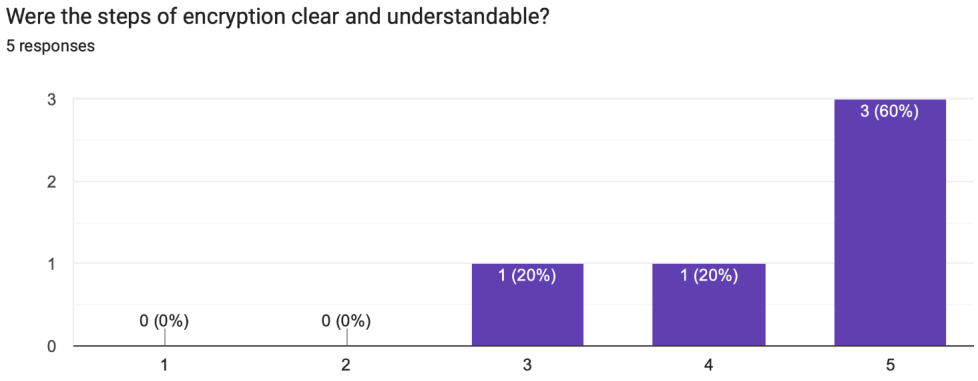


Figure 5.5: Survey question 5 result.

Every one of the respondents was able to successfully encrypt a message of their choosing using the simulator. Respondents were asked if the encryption steps were 'clear and understandable', on a scale of 1 to 5, with one being 'Not at all' and five being 'Very'. Figure 5.5 shows the result. One person gave a score of three, indicating they may not have completely been able to follow the process. Further feedback from that respondent would be needed to follow up on their response and gain an understanding of where in the application was confusing. Overall, the simulator was successful with all respondents, and proved to accomplish the project objectives.

The main response to asking how the simulator could be improved was to have some images included. In this respect, the simulator failed to provide any image elements in the UI. This could potentially have made the users feel less engaged. However, the aesthetics still gained a 4/5 from 60% of the respondents.

The feedback from the users suggests the simulator was easy to use, understandable, and fulfilled its aim of providing a message encryption tool. However, as only five people took part, the results are not very reliable, as the small samples size doesn't accurately represent the larger population, or have enough statistical power to detect meaningful trends in the data. For example, none of the respondents considered themselves an 'expert' on the Enigma machine - someone with a lot more knowledge may have very different opinions on the application.

The full form of responses to the questionnaire can be found in the Appendix C.

5.5 Comparison to similar work

Another important part of evaluating the simulator is comparing it to other existing applications.

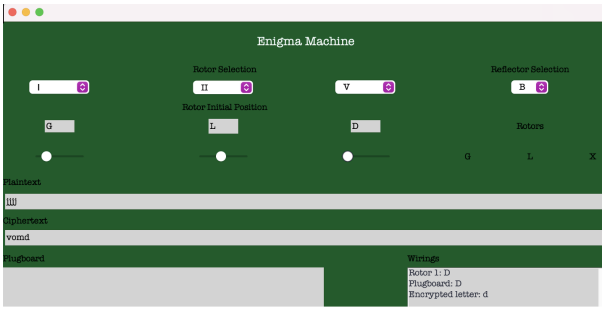


Figure 5.6: This simulator.

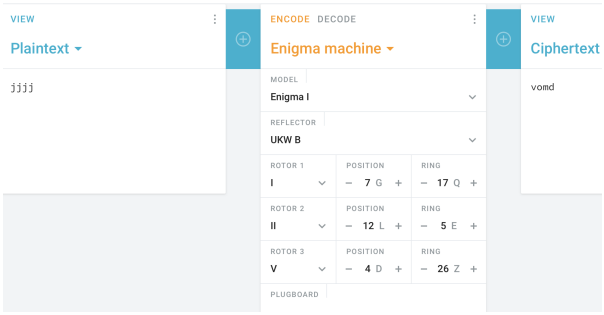


Figure 5.7: Cryptii's simulator. [12]

Figure 5.8: Correct outputs of different simulators with the same settings.

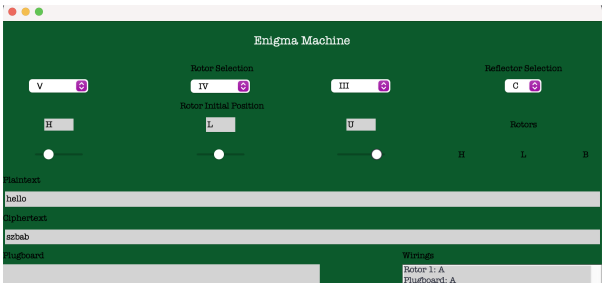


Figure 5.9: This simulator.

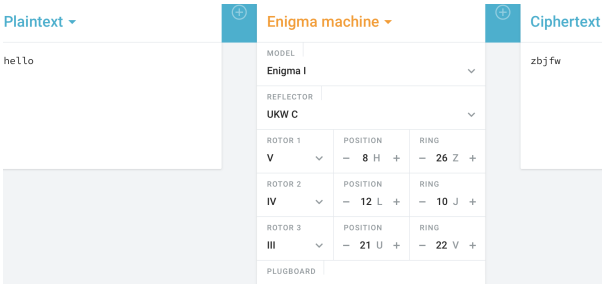


Figure 5.10: Cryptii's simulator. [12]

Figure 5.11: Incorrect outputs of different simulators with the same settings.

For the majority of settings, the simulator gave the same outputs for the plaintext as other applications, however, this wasn't the case every single time. Figure 5.8 shows an example of the simulator being compared to a different online Enigma machine, set to the same configurations. Here, the same output is given. Yet in Figure 5.11, the simulator's outputted ciphertext differs from the Cryptii's machine [12]. With the machines giving a different result, it suggests the simulator may not be 100% accurate. Without access to the source code of the other simulators, it is hard to determine whether this is down to, for example, a slightly different configuration of one of the reflectors, notch settings, or of the alphabets in one the rotors etc.. The best way to test this would have been to visit the original Enigma machine in Bletchley Park and test some inputs there. However, the simulator still successfully mimics the components of the machine and produces adequate ciphertext.

Visually, the simulator is not as advanced as say the virtual based machine (shown previously in Figure 2.7). In the virtual machine, the user experience is much more engaging as it is entirely immersive. They also give a step-by-step tutorial on how to use the virtual simulator, as the user operates the machine. Although this is a useful feature, it is not as relevant for this simulator as it is less advanced and so much more straight forward to use. As shown previously in the questionnaire results, the users were able to intuitively understand how to utilise the simulator instead. However, the time limitation meant that this level of interactivity was not able to met for this project. On the other hand, compared to the text based simulator in Figure 2.6, the simulator is more aesthetically appealing as it makes use of a wider variety of UI components and colour. As oppose to every single component being a dropdown menu, the simulator makes for a more engaging experience. The labels, similar to that of the online application, ensure the simulator's features are clear and explicit.

This project's simulator included the main components of the machine like in similar tools, with customisation of settings at the forefront. Where the simulator lacks compared to others, is that this machine only allows for the Enigma I model to be used. Most simulators on the market give the user the option to change between, for example, the M3, Swiss-K versions etc., which allow for even more flexibility and configuration options.

Chapter 6

Conclusions

This chapter will provide an overview of the project, its key achievements, and ideas for where this work could be taken further in the future. A personal reflection on the project as a whole is also included.

6.1 Summary of achievements

The simulator successfully provides users with a tool to securely encrypt and decrypt their own messages in one application. Following the same process of the Enigma I machine, a plaintext message inputted by the user is encrypted into scrambled ciphertext. This result is displayed to the user. Using the same settings, the machine turns the ciphertext back to plaintext, revealing the original, readable message.

This application simulates the functionality of the Enigma I machine by including all the components of the original machine, namely the five rotors, their initial positions, three reflectors and ten plugboard pairs. These configurations are fully customisable, with the user being able to change and select the value of each. The encryption is symmetric and a letter never encrypts to itself, following the characteristics of the original machine.

A graphical user interface has been built, which give users the ability to choose their own configurations of the machine and visualise the stages of encryption. The wirings console provides a step by step account of the process for each letter that has been encrypted. The rotors 'rotate' on the UI, demonstrating how they 'move' during encryption.

Although it was not a direct aim of this project, the simulator also aids in giving the user some more knowledge of the Enigma machine and a better understanding of how it works.

6.2 Improvements and further work

Following from the discussion in the Evaluation (5), there are several improvements that could be made to the simulator. Without the limitation of time, some more features would have been good to include, and are starting points for further work on this project's simulator. Even more configurations of the machine would be a useful improvement. As mentioned previously, there are many versions of the Enigma machine - this project focused on just the Enigma 1 version. To improve the simulator, enabling the option to choose between versions of the machine and providing the corresponding different settings and components for each could be included. For example, the Enigma M3 machine had eight rotor options as opposed to just the five here. This feature would give the user a larger variety of encryption methods and ensure the simulator matches up to the standards of some of the better existing applications.

More testing was needed to ensure the simulator was accurate for every setting, every time. Travelling to take the simulator to be compared against the machine at Bletchley Park would be the next best step to ensure the results definitely match. This would give high confidence in all test results.

Taking on board feedback from the user questionnaire, the interface could be improved by including some image elements. An option here could be to create the lampboard using images, as originally planned in the design. Instead of individual images for each letter, that was attempted in the implementation stage, there could be an image of a complete keyboard, with 26 versions, each with a different highlighted letter. The entire image could be changed each time a letter was encrypted instead of just one singular letter image. This would potentially make the UI more engaging and more similar to the original machine.

An exciting idea for further work on the machine, outside the scope of this project, is a quantum Enigma machine. A quantum Enigma machine is a theoretical device that combines the principles of quantum mechanics and the Enigma's encryption techniques [19]. The encryption and decryption process would be performed using quantum bits (qubits) instead of classical bits. Qubits are the basic units of quantum information. They can exist in a superposition of states (multiple states simultaneously), allowing for more complex and secure encryption algorithms. A proposed implementation of this quantum machine involves using a network of interconnected qubits to simulate the behavior of the Enigma's rotating disks. This involves using quantum gates to simulate the electrical contacts on the Enigma machine's rotors. The advantage this machine would have over the classic is that it would provide an additional layer of security, that would be extremely difficult to crack using traditional methods. Researchers are currently working towards this goal, with Daniel Lum having already build the first proof of principle machine [20].

6.3 Personal reflection

Before starting this project, I had some knowledge of what the enigma machine was and why it was used, and found it truly fascinating. With problem solving being one of my personal favourite aspects of computer science, this project felt like a clear choice to complete. Without a doubt I learnt a great deal about the Enigma machine itself, in particular the technical details and the true complexity of its functionality. The experience of this project also enhanced my logic and problem solving skills, as understanding the machine first and then trying to implement it in code was at times tough to figure out. My knowledge of Java and Swing has improved greatly, as I implemented a lot of Java classes I had never used before, and Swing as a GUI toolkit was completely new to me. I completed the development of this project in 'iterations', starting with one feature and building up every few weeks to add the rest until the full simulator was built. This method allowed me to stay on track and highlighted the importance of agile and test driven development for me. This project was challenging, but I am proud of the achievements and the knowledge gained throughout the process of creating it.

6.4 Conclusion

Considering the initial aims and objectives that were set for this project, the project was successful in meeting its criteria. The results achieved show an encryption tool that simulates the functionality of the World War II Enigma I machine, complete with customisable configurations and a graphical user interface.

Bibliography

[1] 101Computing. Enigma machine emulator. https://www.101computing.net/enigma-machine-emulator/, 2019.

[2] J. Andrews. Enigma - historical lessons in cryptography. https://jgandrews.com/posts/the-enigma-machine/#setting-up-the-enigma-machine, 2020.

[3] Arnaud. Symmetric vs asymmetric encryption: What's the difference? https://blog.mailfence.com/symmetric-vs-asymmetric-encryption/, 2023.

[4] A. K. Barua. Test coverage definition. https://learn.microsoft.com/en-us/answers/questions/778016/test-coverage-definition-unit-testing, 2022.

[5] G. M. Bateman. The enigma cipher machine. http://www.jstor.org/stable/44326071, 1983.

[6] C. Christensen. Polish mathematicians finding patterns in enigma messages. http://www.jstor.org/stable/27643040, 2007.

[7] ClickSSL. What is symmetric encryption? https://www.clickssl.net/blog/what-is-symmetric-encryption, 2021.

[8] V. Colossus. Virtual enigma. https://enigma.virtualcolossus.co.uk, 2021.

[9] Computerphile. Cracking enigma in 2021. https://www.youtube.com/watch?v=RzWB5jL5RX0, 2021.

[10] Contributors. Enigma machine. https://en.m.wikipedia.org/wiki/Enigma_machine, 2023.

[11] Contributors. Enigma rotor details. https://en.m.wikipedia.org/wiki/Enigma_rotor_details, 2023.

[12] Cryptii. The enigma machine: Encrypt and decrypt online. https://cryptii.com/pipes/enigma-machine, 2023.

[13] A. Elen. The invention of the enigma machine, 2014.

[14] GeeksforGeeks. Caeser cipher in cryptography. https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/, 2023.

[15] A. Hodges. Alan turing: the enigma. In *Alan Turing: The Enigma*. Princeton University Press, 2014.

[16] D. Kahn. Codebreaking in world wars i and ii: The major successes and failures, their causes and their effects. http://www.jstor.org/stable/2638994, 1980.

[17] E. D. Karleigh Moore, Ethan W. Enigma machine. https://brilliant.org/wiki/enigma-machine/#citation-8, 2023.

[18] C. Kasparek. Enigma and poland revisited. http://www.jstor.org/stable/25779307, 2002.

[19] S. Lloyd. Quantum enigma machines. https://arxiv.org/pdf/1307.0380.pdf, 2013.

[20] D. J. Lum, J. C. Howell, M. S. Allman, T. Gerrits, V. B. Verma, S. W. Nam, C. Lupo, and S. Lloyd. Quantum enigma machine: Experimentally demonstrating quantum data locking. https://doi.org/10.1103%2Fphysreva.94.022315, 2016.

[21] A. Marget. Data encryption: How it works and methods used. https://www.unitrends.com/blog/data-encryption, 2023.

[22] A. R. Miller. The cryptographic mathematics of enigma, 1995.

[23] C. Museum. History of the enigma. https://www.cryptomuseum.com/crypto/enigma/hist.htm, 2014.

[24] C. Museum. Enigma i. https://www.cryptomuseum.com/crypto/enigma/i/index.htm, 2023.

[25] C. Museum. Enigma wiring. https://www.cryptomuseum.com/crypto/enigma/wiring.htm, 2023.

[26] C. Museum. How does an enigma machine work? https://www.cryptomuseum.com/crypto/enigma/working.htm, 2023.

[27] Numberphile. 158,962,555,217,826,360,000 (enigma machine). https://www.youtube.com/watch?v=G2_Q9FoD-oQ, 2013.

[28] T. U. of Manchester. Cracking stuff: how turing beat the enigma. https://www.mub.eps.manchester.ac.uk/science-engineering/2018/11/28/cracking-stuff-how-turing-beat-the-enigma/, 2018.

[29] Oracle. Hashmap (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html, 2023.

[30] J. Owen. How did the enigma machine work? https://www.youtube.com/watch?v=ybkkiGtJmkM&t=5s, 2021.

[31] B. Park. Enigma: Our story. https://bletchleypark.org.uk/our-story/enigma/, 2022.

[32] K. Prasad and M. Kumari. A review on mathematical strength and analysis of enigma. https://arxiv.org/pdf/2004.09982.pdf, 2020.

[33] M. Rejewski. An application of the theory of permutations in breaking the enigma cipher. https://cryptocellar.org/enigma/files/rew80.pdf, 1980.

[34] Smartbear. Unit testing. https://smartbear.com/learn/automated-testing/what-is-unit-testing/, 2023.

[35] E. W. Weisstein. Cyclic permutation. https://mathworld.wolfram.com/CyclicPermutation.html, 2023.

Appendix A

Final Simulator Features

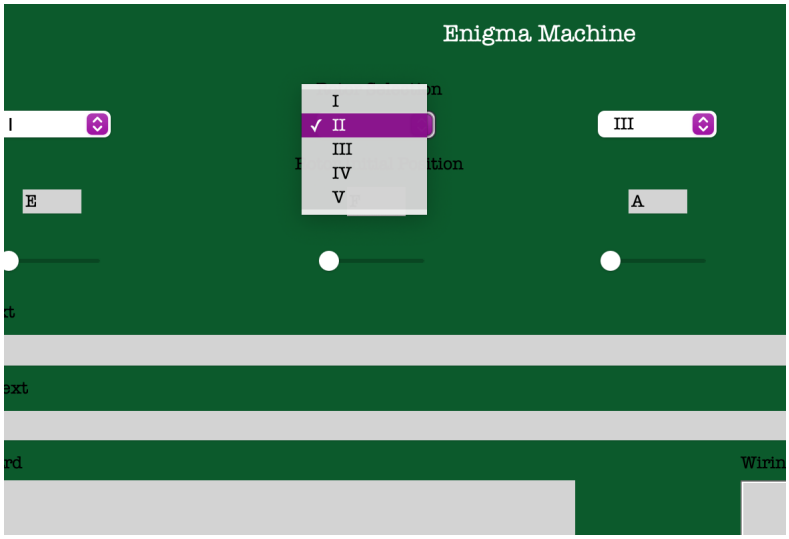


Figure A.1: Rotor selection menu.

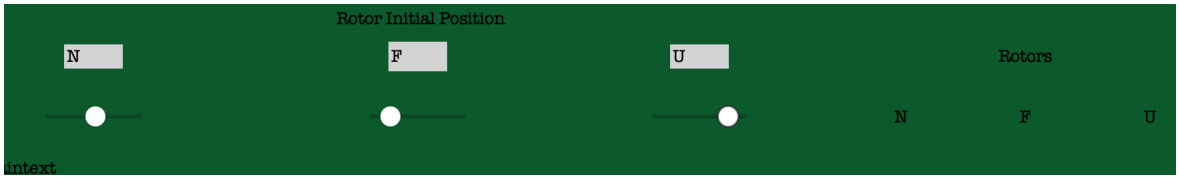


Figure A.2: Rotor initial position selection slider.

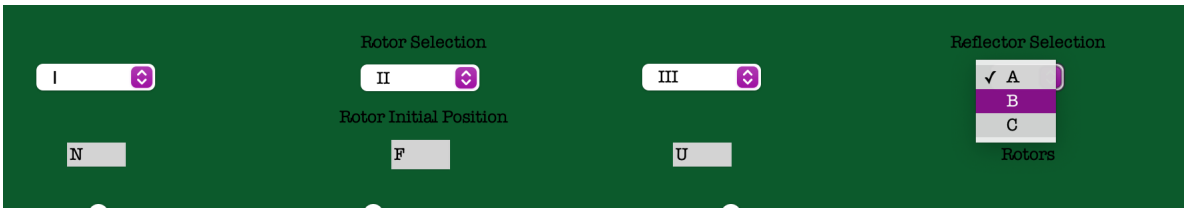


Figure A.3: Reflector selection menu.

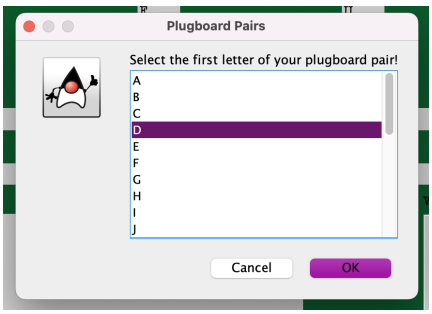


Figure A.4: Plugboard pair selection first letter.

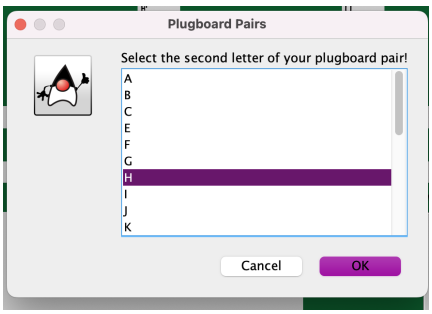


Figure A.5: Plugboard pair selection second letter.

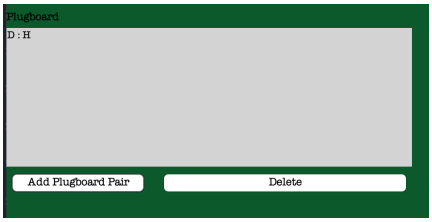


Figure A.6: Built plugboard pair.

Figure A.7: Plugboard feature.

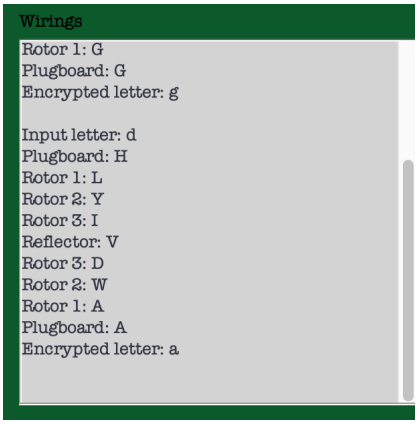


Figure A.8: Console of encryption steps.

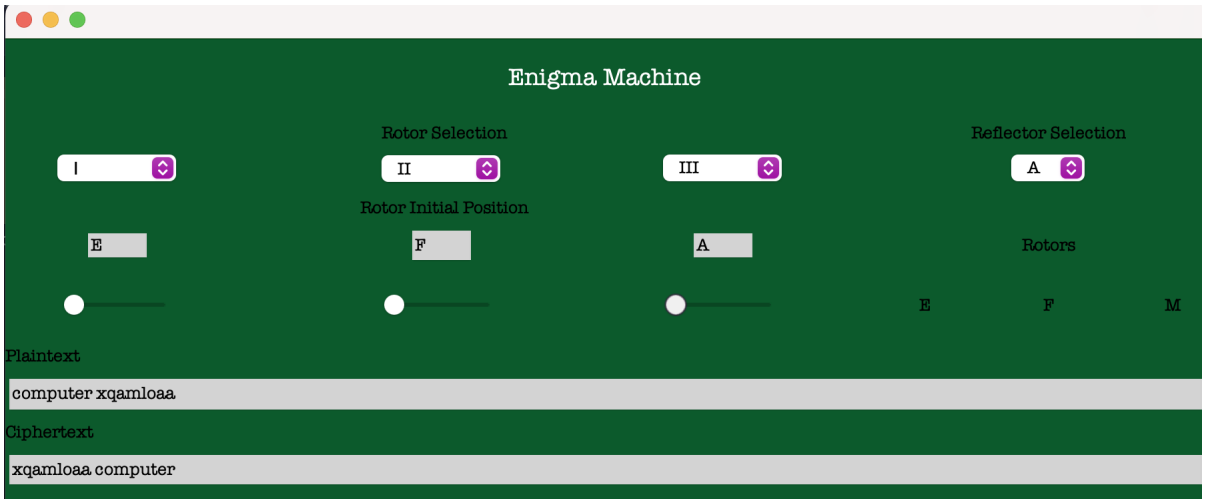


Figure A.9: Encrypt and decrypt using same settings.

Appendix B

Code Snippets

```
/**
     * Encrypts the given inChar.
     * The message is passed through the plugboard and mapped to a pair.
     * Then the message is passed forward through the rotors, the reflector, and then passed back through the rotors in
     * reverse order.
     * Then the end message should be passed through the plugboard once more for final output.
     *
     * @param inChar Input message.
     * @return result
     */
    public EnigmaResult encrypt(final char inChar) {
        int index = 0;
        String result = "";
        final EnigmaResult res = new EnigmaResult();
        final List<String> steps = new ArrayList<>();
        boolean isCaps = false;
        if (!Character.isLetter(inChar)) {
            result = result + String.valueOf(inChar);
            res.result = result;
            return res;
        }

        if (Character.isUpperCase(inChar)) {
            isCaps = true;
        }

        this.moveRotors(this.rotors, 0);

        steps.add("Input letter: " + inChar);
        final char c = Character.toUpperCase(this.MapPlugboardPairs(inChar));
        steps.add("Plugboard: " + c);
        index = Rotor.toIndex(c);
        index = this.rotors[0].convertForward(this.rotors[0], index);
        steps.add("Rotor 1: " + Rotor.toChar(index % 26));
        index = this.rotors[1].convertForward(this.rotors[1], index);
        steps.add("Rotor 2: " + Rotor.toChar(index % 26));
        index = this.rotors[2].convertForward(this.rotors[2], index);
        steps.add("Rotor 3: " + Rotor.toChar(index % 26));
        index = this.reflector.convertForward(this.reflector, index);
```

```
        steps.add("Reflector: " + Rotor.toChar(index % 26));
        index = this.rotors[2].convertBackward(this.rotors[2], index);
        steps.add("Rotor 3: " + Rotor.toChar(index % 26));
        index = this.rotors[1].convertBackward(this.rotors[1], index);
        steps.add("Rotor 2: " + Rotor.toChar(index % 26));
        index = this.rotors[0].convertBackward(this.rotors[0], index);
        steps.add("Rotor 1: " + Rotor.toChar(index % 26));

        if (0 > index) {
            index += 26;
        }

        char resultChar = Rotor.toChar(index % 26);
        resultChar = this.MapPlugboardPairs(resultChar);
        steps.add("Plugboard: " + resultChar);
        if (!isCaps) {
            resultChar = Character.toLowerCase(resultChar);
        }

        final String resultCharAsString = Character.toString(resultChar);
        steps.add("Encrypted letter: " + resultCharAsString);
        result = result + resultCharAsString;
        res.steps = steps;
        res.result = result;
        return res;
    }
```

Listing B.1: Code for the encrypt method.

```
public class EnigmaBackend {
    @Test
    public void testEncrypt(){
            ⋮
    }
    @Test
    public void testEncryptNewRotors(){
            ⋮
    }
    @Test
    public void testPlugboardPair(){
            ⋮
    }
    @Test
    public void testNotch(){
            ⋮
    }
    @Test
    public void testEncryptAndDecrypt(){
            ⋮
    }
    @Test
    void toIndex() {
            ⋮
    }
    @Test
    void toChar() {
            ⋮
    }
    @Test
    void setPosition() {
            ⋮
    }
    @Test
    void convertForward() {
            ⋮
    }
    @Test
    void setToChar() {
            ⋮
    }
}
```

Listing B.2: List of unit tests.

Appendix C

Questionnaire Responses

Figure C.1: Questionnaire responses Q1-3.

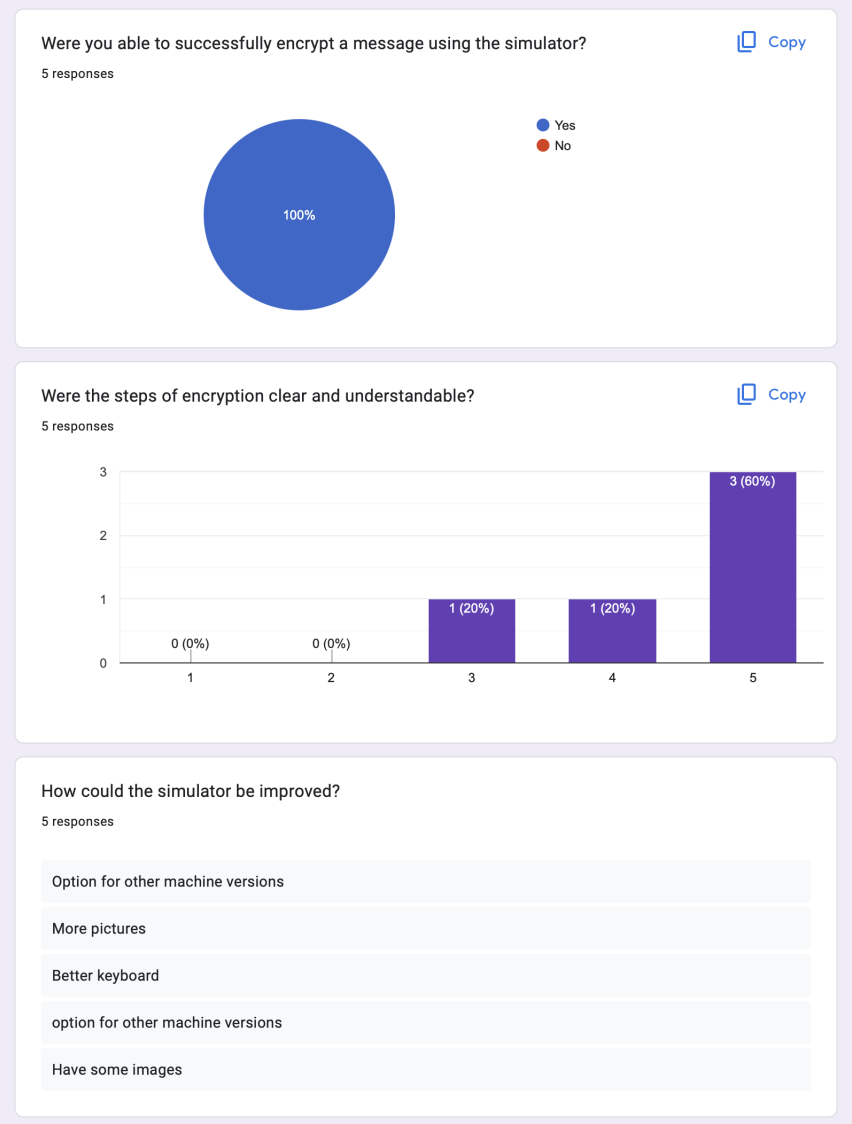


Figure C.2: Questionnaire responses Q4-6.

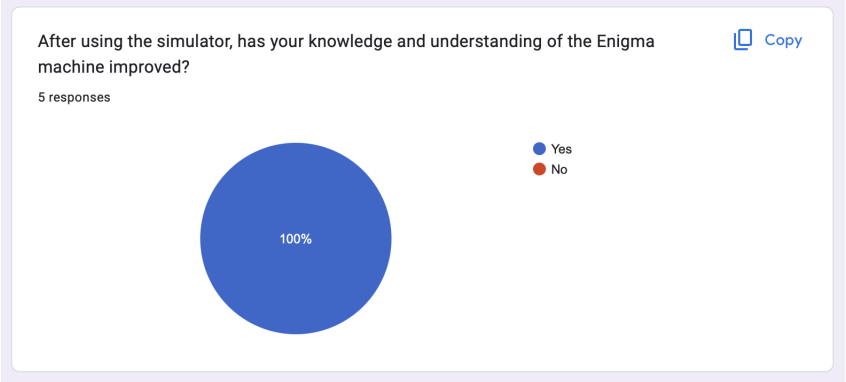


Figure C.3: Questionnaire responses Q7.