Enigma Machine Simulator

Elliot Brooks

Supervisors:

April 2024

# Abstract

Contents

[1 Abstract 2](#_Toc160610271)

[2 Introduction 5](#_Toc160610272)

[2.1 Aims and Objectives 5](#_Toc160610273)

[3 History & Background 6](#_Toc160610274)

[3.1 History of the Machine 6](#_Toc160610275)

[3.2 Design of the Machine 6](#_Toc160610276)

[3.2.1 Keyboard & Lampboard 6](#_Toc160610277)

[3.2.2 Rotors 7](#_Toc160610278)

[3.2.3 Reflector 7](#_Toc160610279)

[3.2.4 Plugboard 8](#_Toc160610280)

[3.3 Enigma Machine and Abstraction 8](#_Toc160610281)

[3.3.1 Enigma’s Encryption 8](#_Toc160610282)

[3.3.2 Abstractions 9](#_Toc160610283)

[3.4 Design Flaws & Remedies 12](#_Toc160610284)

[3.4.1 Cypher Strength 12](#_Toc160610285)

[3.4.2 Improving the Machine 12](#_Toc160610286)

[3.5 Related work 13](#_Toc160610287)

[4 Design & Implementation 15](#_Toc160610288)

[4.1 Tools and Technologies 15](#_Toc160610289)

[4.2 Enigma Model Design 15](#_Toc160610290)

[4.2.1 Keyboard & Lamp board 16](#_Toc160610291)

[4.2.2 Reflector 16](#_Toc160610292)

[4.2.3 Plugboard 17](#_Toc160610293)

[4.2.4 Rotors 17](#_Toc160610294)

[4.2.5 Enigma 19](#_Toc160610295)

[4.2.6 EnigmaPlus 20](#_Toc160610296)

[4.2.7 Additional Features 21](#_Toc160610297)

[4.3 Design of the Applications 21](#_Toc160610298)

[4.3.1 Command Line Interface 23](#_Toc160610299)

[4.3.2 GUI 24](#_Toc160610300)

[4.3.3 Visualization 24](#_Toc160610301)

[5 Results 24](#_Toc160610302)

[6 Evaluation 24](#_Toc160610303)

[6.1 Model Evaluations 24](#_Toc160610304)

[6.2 GUI Evaluation 24](#_Toc160610305)

[7 Conclusion 24](#_Toc160610306)

[8 References 24](#_Toc160610307)

[9 Table of Figures 25](#_Toc160610308)

[10 Tables of Tables 25](#_Toc160610309)

[11 Table of Equations 25](#_Toc160610310)

[12 Table of Code Blocks 26](#_Toc160610311)

# Introduction

The Enigma Cypher Machine (*Enigma I)* is most well-known for its usage and eventual breaking, during World War 2 (WW2). The machine used several mechanical rotors, a reflect and a plugboard to redirect electrical signals, scrambling a plaintext message into cyphertext. Whilst most have heard of the machine, either from a study of cryptography or through history, many do not understand how the machine functions and its significance in cryptography. By creating a visual tool to demonstrate the inner workings of the Enigma, the aim of the project was to offer a deeper insight into the functionality, significance, but also the weaknesses of the machine.

There already exists a handful of examples of Enigma simulations which tend to offer a skeuomorphic approach to modelling the machine. The challenge I wanted to overcome was to create a tool that teaches the user how the machine functions logically. I began my project by thoroughly researching *Enigma I* to understand the inner workings of the machine and to compile this research into a simpler, abstracted model using Java. I created both a multi-platform command line interface (CLI) and a multi-platform graphical user interface (GUI) to allow the user to interact with the model, the former offering a step-by-step visualisation of the encryption of a message. To complete the project, I developed a second “EnimgaPlus” model which aims to correct the two key cryptographic weaknesses of the machine. I identified the accuracy of my model by comparing with similar products as well as some real-world messages found amongst the German forces during WW2. I also identified the effectiveness of the two Enigma models by using some basic cryptoanalysis.

## Aims and Objectives

The following list adapted from the original project description, however there are some additional objectives I have included myself. For each task I set my own criteria to assess the success of each task’s implementation.

|  |  |
| --- | --- |
| Aim / Objective | Criteria for Success |
| Create a standalone package that simulates *Enigma I* | The package should simulate only the Enigma machine and should be able to be used in different applications. It should also produce the correct output for a given input. The package should allow the Enigma Machine to be fully configurable as the ‘Enigma I’ |
| Create a new “EnigmaPlus” machine with an aim of fixing cryptographic weaknesses of *Enigma I* | The “EnigmaPlus” machine should function similarly to the original Enigma and should be more secure cryptographically |
| Create a basic command line interface | The CLI should be allow full configuration of the machine and allow the user to enter a plaintext message |
| Create a GUI with a visualisation tool | The GUI should have the same capabilities, if not more than the CLI. In addition, it should aim to visually reflect the machine.  The visualisation tool should provide a simplified and informative representation of the encryption steps |
| Support multiple platforms | The application should be compatible with Windows/MacOS/Linux |

# History & Background

## History of the Machine

The Enigma cypher is one of the most famous cipher machines (Enigma n.d.) due to its role in WW2 and the work undertaken at Bletchley Park to crack the code. The Enigma was used extensively by the German forces to transmit coded messages for more secure communication. Whilst most people refer to this machine as “The Enigma”, Enigma is a brand name for a series of cypher machines (Enigma n.d.). The one used most in WW2 was *Enigma I* and was the key focus of this project.

To use the Enigma machine, it had to be configured to an exact setting so that the machine could be correctly decoded by the intended parties. These settings were distributed to the German forces each month in a code book, containing each day’s settings.

## Design of the Machine

The Enigma Machine was a rotor-based machine and worked with a hybrid of mechanics and electrical signals. *Enigma I* contained 5 key components that worked together to produce cyphertext which were: the keyboard and lampboard, the rotors, the reflector, and the plugboard. The inclusion of all these components aimed to make the machine as unpredictable as possible as well as generating a large key-length of roughly 67 bits (nearly 159 quintillion different settings see Equation 6) (van Manen and Robertsson 2016).

The use of the machine was a simple process. A user would receive a message along with a key denoting the choice of rotors and their respective settings, the plugboard settings and a reflector choice. Each time a key was pressed on the keyboard, a light on the lampboard would emit light as demonstrated in Owen’s animation (Owen 2021) and the user would write down the corresponding letter. Due to the symmetric design of the Enigma machine, as long as two people had the same settings then they could simply input the cyphertext into the machine and receive the plaintext and vice versa. However, this design choice was overlooked and compromised the Enigma’s security (See 3.4) (Thimbleby 2016).

An old machine on a glass table

Description automatically generated

Figure A photo of Enigma I showing the plugboard, keyboard and lampboard. Only a small portion of the rotors are visible, and the reflector is hidden. Photo taken by author at Science Museum London (Science Museum n.d.)

### Keyboard & Lampboard

The keyboard and lampboard were the interface which the user could encode/decode messages. The former was comprised of the 26-letter alphabet omitting any special or numeric characters. Each key on the keyboard could be pressed which would cause a ratchet mechanism to move a lever (pawl) to step the rotors (Hamer 1997). The lampboard was a copy of the keyboard but instead of keys, there were small glass panels which would allow the bulbs underneath to shine through. These panels were also printed with the 26-letter alphabet and upon a keypress, any given lamp could light up to show the plaintext character’s corresponding cyphertext.

### Rotors

The rotors were the heart of the Enigma machine and were responsible for most of its unique properties. They were metal ratchet discs with 26 different positions representing each letter of the alphabet. Each position had a corresponding metal contact (Owen 2021) on both sides of the disk to allow electrical current to flow through the rotor. Inside the rotor, fixed wires were implemented which directed the current from one contact to another, thereby encoding the input.

Rotors also exhibited another property; each rotor had a notch at a fixed position on the ratchet which would allow the levers (pawls) mentioned earlier to ‘step’ the rotor (Hamer 1997). This stepping caused the rotor to rotate by one position. In the machine, three rotors were placed in series to allow current to pass through all three, causing a letter to be scrambled 3 times from one key press. Once a rotor reached its turnover position (the character shown to the user once the notch position is lined up to the pawl, the latter is not seen by the user of the machine), the rotor to left of the turnover rotor would be stepped as well. The right-most rotor would step every key press and the middle rotor would step with a period of 25 (Grime 2013) (Smart 2016) due to a quirk of the machine known as double stepping, where the middle rotor would also step when it reached its own turnover position as demonstrated by Hamer (Hamer 1997). The left-most rotor would step with a period of 262. This rotational property of the machine allowed for the encoding for a given letter to change each keypress.

The Enigma rotors could be swapped around and placed in any order in the three slots available. Usually, users were given a box containing 5 different rotors the choice of which to use formed part of the Enigma’s key. Table 1 depicts the 5 rotors that were included with *Enigma I* demonstrating their internal wiring as well as the location of the notch. For example, rotor I will map A to E and B to K provided the rotor is in rest position (Position A and Ring setting A).

|  |  |  |  |
| --- | --- | --- | --- |
| Rotor | Encoding (Position A - Ring setting A) ABCDEFGHIJKLMNOPQRSTUVWXYZ | Notch | Turnover |
| I | EKMFLGDQVZNTOWYHXUSPAIBRCJ | Y | Q |
| II | AJDKSIRUXBLHWTMCQGZNPYFVOE | M | E |
| III | BDFHJLCPRTXVZNYEIWGAKMUSQO | D | V |
| IV | ESOVPZJAYQUIRHXLNFTGKDCMWB | R | J |
| V | VZBRGITYUPSDNHLXAWMJQOFECK | H | Z |

Table 1 "Enigma I" rotor encodings (Enigma wiring n.d.)

Finally, the rotors had an additional setting known as the ring setting. This allowed the internal wires and ratchet to be shifted independently from the letter ring, allowing the notch position to move relative to the letter ring. Whilst generally, the ring setting formed part of the key, it is important to note that it had marginal impact on the strength of the cypher, only affecting the turnover position of the adjacent rotor (How does the Enigma machine work? n.d.).

Much of the complexity of the Enigma was due to these rotors. Alone, with an *Enigma I* model, there were 1054560 different ways to configure the rotors (See Equation 1)

Equation Permutations for rotors (excluding ring setting)

### Reflector

The reflector was a similar component to that of the rotors and together formed the subsystem that does most of the scrambling. The main differences regarding the reflector are that it does not rotate, and the current does not pass through but rather is ‘reflected’, travelling back in the opposite direction. This reflection allows the reflector to act as a similar substitution cypher akin to the rotors.

The design of the reflector was to enable the whole machine to be reciprocal, combining encryption and decryption into one operation. This, along with the fact that the reflector could not encode a letter to itself (the design of the rotor prevented the current being passed back through the same metal contact) were crucial flaws exploited (Thimbleby 2016) by the team at Bletchley Park to crack the code during WW2.

The Enigma machine came with a reflector which could be replaced with others but generally remained the same. The three that were available with the *Enigma I* were UKW-A, UKW-B and UKW-C as shown in Table 2. The encodings demonstrate the inner wiring of each reflector, for example UKW-A maps A to the letter E as they have matching indexes.

|  |  |
| --- | --- |
| Reflector | Encoding ABCDEFGHIJKLMNOPQRSTUVWXYZ |
| UKW-A | EJMZALYXVBWFCRQUONTSPIKHGD |
| UKW-B | YRUHQSLDPXNGOKMIEBFZCWVJAT |
| UKW-C | FVPJIAOYEDRZXWGCTKUQSBNMHL |

Table "Enigma I" reflector encodings (Enigma wiring n.d.)

### Plugboard

The plugboard formed the final part of the Enigma’s encryption key and was located at the front of the machine. It displayed another representation of the 26-letter alphabet, each of which had a plug socket. These sockets, and the cables that came with the machine allowed two letters to be connected to each other on the plugboard. This created yet another scrambling of the letter, such that if socket A and socket E were connected then any current passing through the plugboard in wire A, would be directed to wire E and vice versa. Any letter left unconnected to another would result in the plugboard having no effect for that letter. The plugboards letter swapping effect only occurred twice in each encryption, once at the start of the encryption (after the keypress) and once at the end (before the lamp on the lampboard lights up).



Figure A front-facing photo of the Enigma I plugboard with cables in place (Enigma n.d.)

## Enigma Machine and Abstraction

### Enigma’s Encryption

Often the best way to understand the mechanisms of the machine is to focus on a single letter’s encryption. As mentioned in 3.2, the user begins with defining the machine’s key. Upon a keypress on the keyboard, the first notable event is that the rightmost rotor will rotate. Depending on the current rotation, the other rotors may also rotate but only ever by one position. Once this step is completed, an electrical signal will be induced passing through the plugboard towards the rotors. Depending on the plugboard settings, this input signal may be scrambled. This electrical signal then passes through rotors right-to-left with the signal being redirected at each rotor. The output of the three rotor redirections is then transmitted to the reflector where the signal’s direction is reversed and redirected to the contact of a different letter. The signal then passes through all three rotors for a final time, this time from left-to-right, before passing through the plugboard again. Finally, the signal is transmitted to the lampboard where the cyphertext is displayed. Any plaintext letter can be scrambled up to nine times before the cyphertext is displayed.

By representing each of the components of the Enigma as a transformation as demonstrated by Rejewski (Rejewski 1980), such that represents the plugboard, represents the rotor in the left, middle or right position and represents the reflector, an equation, such as Equation 2, can be formulated to describe Enigma’s encryption steps. It is important to note that due to the plugboard’s symmetry, .

Equation The encryption steps of Enigma I

### Abstractions

The Enigma machine is complex. With numerous components working together in both mechanical and electrical forms, it can prove difficult to predict the outcome of an encryption. This section aims to abstract each component into a logical model to help demonstrate the behaviour and weaknesses of the machine.

Firstly, The diagrams provided (Figures 3-7) are inspired by work from Smart (Smart 2016) and Thimbleby (Thimbleby 2016) and depict rectangles with nodes on each side to represent each letter in a truncated alphabet (A-F). The connections between these nodes represent the internal wiring of each component or, in a more logical sense, the letter-to-letter mappings of each component. These diagrams are designed such that the any letter is **input on the right** and the **output on the left** is map(x). Consequently, the left can also represent and the right-hand side represents . For example, Figure 3 shows the plugboard in which if the input on the right is “A” then the output will be map(“A”) which is “D”. The reason for representing input on the right-hand side is to better reflect the physical layout of the Enigma machine. It is important to note that, in the case of rotors, the function refers to the component with no rotational effects.

The plugboard acts as a simple cypher, which aims to swap two letters. This means that letters connected by a cable on the physical machine are encoded to each other; in absence of a cable, no letter scrambling takes place. As shown in Figure 3, sockets on the machine that connected are represented with a connection between nodes such as A and D. Due to the plugboard symmetry, these diagrams exhibit this ‘X’ shapes.

A diagram of lines and dots

Description automatically generated

Figure A wiring diagram to show a plugboards potential encoding

The reflector acts as a substitution cypher with additional constraints being self-coding and reciprocal coding (See Equation 3). Like other components this can be represented using a wiring diagram such as Figure 4 which demonstrates an example of a reflector. In the diagram, A is shown to be connected to F and vice versa such that any input into the reflector will output the letter at the connected node.

Equation An equation and additional constraints to describe the behaviour of an Enigma reflector

A diagram of a diagram

Description automatically generated

Figure A wiring diagram to show a potential encoding for a reflector

The rotors of the machine are substitution cyphers with no additional constraints. They take an input letter and produce either the same or a different letter. A rotor can be represented using a wiring diagram, however multiple diagrams are needed to convey the rotor’s rotational effects. As demonstrated in Figure 5, the image on the right depicts the same rotor displayed on the left but with a rotation of one. This causes the connections between nodes to move upwards whereas the ring setting will cause them to move downwards. This effect is easily seen with the horizontal connection between F on the left image. Once the rotor is rotated the same connection is moved upwards in the diagram to become a horizontal connection between E as shown by the image on the right. This effect can be generalised such that any input letter will be mapped to the input letter . In addition, the output of the letter will be shifted by (See Equation 4). It is clear from these generalisations that if the rotation setting was ten and the ring setting was ten, then there would be no effect on a letter’s encryption as mentioned in 3.2.2.

Equation A function to represent the encoding behaviour of the rotor where x and x` are letters, represents the rotor’s rotation and represents the ring setting

A diagram of lines and dots

Description automatically generatedA diagram of lines and dots

Description automatically generated

Figure Two wiring diagrams to show a potential rotor encoding. The diagram on the right shows the same rotor as on the left, but with a rotation of one

By abstracting all the electrical and mechanical features of the machine, logical diagrams demonstrating the letter scrambling that takes place in the Enigma machine can be created with greater ease. The diagrams from Figures 3-5 represent each component of the machine and can be combined to create a representation of the entire Enigma machine (See Figures 6-7). These diagrams depict an Enigma machine and demonstrate how the machine works in full. It is important to note that these diagrams only show a single state, upon each key press the rotor wirings will change leading to a potentially different output for the same input.

A diagram of a network

Description automatically generated

Figure A wiring diagram representing a single state of an Enigma I machine. Input is received on the right-hand side before being scrambled by components performing a loop. The electrical signal received back from the plugboard represents the encoded letter. The names of each rotor/reflector do not match the encodings but are given as an example.

A diagram of a network

Description automatically generated

Figure An example of an encryption/decryption taking place in an Enigma I machine. In this case A is encoded to C.

## Design Flaws & Remedies

### Cypher Strength

At a first glance, it may seem like the Enigma machine is unbreakable and indeed the Germans shared this over-confidence (Thimbleby 2016). The Germans became complacent when operating the machine often opting to use the same 3 rotors and neglecting to change the reflector (Tang, Lee and Russo 2018). This led to a to a large reduction in the security of communication between operators by factors that could have been largely avoided (Thimbleby 2016).

Whilst the Enigma machine had a large key space (see Equation 6), it is not the only factor that contributes to a cypher’s strength (How does the Enigma machine work? n.d.). In fact, work from Tang, Lee and Russo (Tang, Lee and Russo 2018) suggests that the Enigma had a theoretical key space of . As suggested by Thimbleby (Thimbleby 2016), by imagining the Enigma machine without its internal components, it can be viewed a substitution cypher with different mappings from keyboard to lamp board. However, this assumes that there are no restrictions on how the mappings can be configured. As mentioned earlier in 3.2, this was not the case for *Enigma I*. The actual number of permutations for the Enigma machine was (See Equation 5). The reduction in permutations by a factor of approximately is due to two features of the machine: self-coding and reciprocal coding (see Equation 3) (Ostwald 2023) (Thimbleby 2016).

Equation Definition of double factorial

Equation The number of settings (key space) of Enigma I assuming 10 plugboard cables are used. Ring setting is omitted as it was not changed by the Germans. Based on work from Tang, Lee and Russo (Tang, Lee and Russo 2018).

### Improving the Machine

Clearly, the most obvious solution to improve the *Enigma I* is to increase its key space with more rotors. This can be done by adding an additional slot to the machine to allow for 4 or more rotors to be in use at any one time, or even using the usual 3 rotor slots but having a larger collection to choose from. In fact, rotor IV and V were introduced later in 1932 (Ostwald 2023) in order to increase the key space. Other machines inspired by Enigma, such as the British Typex, were developed to utilise more than 3 rotors at a time, thus increasing combinatorial complexity (Ostwald 2023). Ostwald’s study (Ostwald 2023) shows numerous additional improvements that either were implemented or could have been implemented to improve the cypher key space.

Thimbleby (Thimbleby 2016) gives examples of multiple circuits that aimed to fix the Enigma’s two main weaknesses mentioned in 3.4.1 demonstrating that the technology at the time period was capable of creating a much stronger cypher. In particular, Figure 8 demonstrates a circuit that only uses 3 rotors omitting both the reflector and the plugboard. By removing these two components, it avoids both self-coding and reciprocal coding by separating encoding and decoding into two distinct functions. For encoding, the current would pass from right to left, and vice versa for decoding. Whilst this model does not include the plugboard, so long as it is only applied once in an encryption, it can still be included so that both weaknesses are avoided. The removal of the reflector does reduce the key space however, this can be mitigated by simply adding another rotor.

A diagram of a machine

Description automatically generated

Figure A circuit showing an Enigma style machine avoiding both self-coding and reciprocal coding. Taken from Figure 6 (Thimbleby 2016)

The design in Figure 8 mainly focusses on the electrical behaviour of the hypothetical machine however it acted as the main inspiration for “EnigmaPlus”. By representing this circuit as a wiring diagram similar to Figures 6-7, a logical model for this machine can be inferred as shown in Figure 9.



Figure A wiring diagram depicting "EnigmaPlus". Note that there is no reflector as well as encoding and decoding take place in opposing directions.

## Related work

Before conducting the research to this product, several similar products were found that simulated the Enigma machine. This project and much of the work done around *Enigma I*were inspired by these products. In addition to this, the paper written by Thimbleby (Thimbleby 2016) gave direct inspiration into the creation of “EnigmaPlus”.

“Enigma Machine Emulator” (Enigma Machine Emulator n.d.) is a webpage consisting of a short description about the Enigma machine as well as offering an interactive emulator for the Enigma machine. The emulator follows a skeuomorphic design aiming to present a flat image that resembles the real machine.

“The Enigma Machine” (Enigma Machine n.d.) presents a sleeker UI allowing the user to configure the machine as well as encode/decode their messages. The design of this webpage strays from the original look and feel of the machine, offering a new and simpler way to interact with it. Much of the design points for my project were inspired by this page.

“Virtual Enigma” (Virtual Enigma n.d.) offers extremely interactive 3D simulation of the Enigma machine. This simulation allows full control of the machine down to each minute detail. Users can open different parts of the machine with their mouse and drag and drop rotors into place on a virtual model of the machine. The level of detail in this simulation offers an authentic experience for the user and is highly recommended for those who would like to experience the machine as a whole.

“The Enigma Machine” (MacWright n.d.) is the only similar product that I came across that offers a form of visualization to the inner workings of the machine. It represents the Enigma’s encodings as a circular wiring diagram analogous to Figures 3-7. This particular page was the inspiration for the visualiser functionality for this project.

# Design & Implementation

The goal of this project was to create an exact simulation of Enigma and its components being: the rotors, the reflector, the plugboard and the keyboard and lampboard. To accomplish this, an Enigma model was developed to take an input plaintext message and produce a cyphertext message. The Enigma model is comprised of numerous Java classes, each reflecting the physical components of the Enigma machine. To offer a basic interaction with the model, a command line interface was also developed, allowing the user to fully configure and operate the model. To further develop the project, a GUI accompanied with a visualisation tool was created to offer a better experience operating, configuring and understanding the machine. Finally, an additional model was created to reflect the hypothetical machine mentioned in the section 3.4.2 which was named “EnigmaPlus”. This chapter gives an account of the design and implementation of the Enigma model, the EnigmaPlus model, the CLI and the GUI.

TODO : Re-evaluate this section, perhaps move the system overview here? Maybe diagram showing input and output?

## Tools and Technologies

Throughout development of this project, multiple tools and technologies were utilised to accomplish the task. Below shows a list of these tools with a brief description and use in this project.

TODO : Add additional information, give each tool/technology a paragraph?

* **Java 20** – Java programming language used for underlying code
* **Maven** – Java build tool utilised to support multiple platforms and manage dependencies.
* **JUnit** – Java testing library used for creating and running unit tests.
* **JavaFX** – Java graphics package exclusively used for GUI and visualisation.
* **Gluon Scene Builder** – A “drag and drop” style UI builder that accompanies JavaFX.
* **XML** – Object description language used to define custom components.
* **GitHub/Git** – Version control software used throughout project for version control.
* **Visual Studio Code** – Extension-based text editor used for debugging and project management.

## Enigma Model Design

With all the information from section 3.2 in mind, it was decided that building the simulation of the Enigma machine would best be done in Java due to its native support for object-oriented programming. This allowed for each component to be built independently from one another and properly reflect the physical machine.

The Enigma package that was built contains each component that comprises the Enigma as well as a class to model the complete machine. This code was kept in a separate package such that an interested user could, for example, use the Enigma package to create their own application that interacts with the model. To fortify this, the GUI and CLI applications built for this project were kept independent of the Enigma package (see section 4.3). For simplicity, the UML diagram shown in Figure 10 does not contain EnigmaPlus however this is introduced later. With this general structure of the Enigma package, each component can be addressed in the following sections, with a focus on their key behaviour and data structures.

A diagram of a computer program

Description automatically generated

Figure The UML diagram for the Enigma model contained within the "Enigma" package. This diagram omits any methods for each class for simplicity

### Keyboard & Lamp board

Regarding the physical machine, the keyboard and lampboard are large components, acting as the core interface for any given operator. In a logical sense, the keyboard and lampboard are nothing more than input and output respectively. Due to this, the Enigma model does not contain any reference to the two components. Instead, each component will allow for input and output such that any text/string modality can be used to interact with the machine.

### Reflector

As represented by Equation 3, the reflector is a simple substitution cypher with some additional constraints. As such there were two questions to answer: how should the reflector encodings be represented and how should the constraints be enforced?

At first, it was thought the best way to represent the encodings of any given reflector should be to use a hash-map, assigning each output letter (value) to an input letter (key). Whilst the same functionality would be possible with this structure, the high-level nature of hash-maps brought unnecessary complexity into the representation. In addition, for the standard alphabet of 26 characters, there would have needed to be 52 key-value pairs in the hash-map. It was decided that a simple array (depicted as wiring) of length 26 would be used instead. In this array, each index represents the positional encoding of a letter and the value at that index would represent the positional encoding of the paired letter. This method offers simplicity as well as quick index speeds and is also applied in the rotors and plugboard with small variations.

The construction of an array such as wiring is a difficult task for someone to complete and so a function was developed that takes a string encoding (such as the ones shown in Table 2) and builds an array from this. This same functionality is also used for the rotors. This addition allowed for a check on the encoding string to be performed, accounting for the constraints of the reflector.

TODO : REMOVE CODE BLOCK

FUNCTION validateEncoding(String encoding) {

// Ensures a complete mapping

IF (encoding.length ≠ 26) {

RETURN false

}

// Ensures all mappings are 1:1

IF (encoding CONTAINS DUPLICATE LETTERS) {

RETURN false

}

// Ensures self-coding weakness

charArray = encoding.toCharArray()

FOR (i IN charArray.length) {

IF (charArray[i] == i) {

RETURN false

}

}

RETURN true

}

Code Block Check reflector encoding meets constraints

Once the representation of the reflector was in place, the reflector’s encode function could be developed. Notice the similarities between Equation 3 and Code Block 2.

FUNCTION encode(characterIndex) {

RETURN wiring[characterIndex]

}

Code Block Reflectors encode function

### Plugboard

Once the practice of using an array to represent the wiring, the implementation of the plugboard became clear. By initialising this array with values where each value matches its index, the plugboard is modelled in an unconfigured state. Then once this has been initialised, several public functions were developed for the addition and removal of connections in the plugboard as shown in Code Block 3.The encryption function of the plugboard could then be applied in the same way as the reflector as in Code Block 2.

TODO : Replace with the plugboards encode

FUNCTION addCable(firstCharacter, secondCharacter) {

wiring[firstCharacter] = secondCharacter

wiring[secondCharacter] = firstCharacter

}

Code Block Plugboard function to add a new connection

At this point in the process, it was decided the simplest way to represent a plugboard connection in terms of input and output was as a string. For example, if there was a connection between A and B then this could be represented as “AB” or “BA”. This representation as input could be easily parsed and processed in order to reflect the connection in the wiring array.

### Rotors

The rotor for the Enigma model posed the biggest challenge, as it required the most complex representation. As with the reflector, an array of length 26 was used to represent the mappings of the rotor which is again depicted as wiring. The inverse mappings were simple to represent as it required another array depicted as reverseWiring. This array followed the same structure as wiring with the only difference being the values in each index represented the inverse of wiring (See Code Block 4).

TODO : Remove and re write surrounding text

FUNCTION configureReverseWiring() {

reverseWiring = ARRAY[wiring.length]

FOR (i IN wiring.length) {

characterIndex = wiring[i]

reverseWiring[characterIndex] = i

}

}

Code Block Building the reverse wiring attribute using the forward mappings

To implement the rotations of the rotors, two attributes called currentRotation and ringSetting were needed, whose integer values kept a record of these two properties. Since the ring setting of a rotor is fixed during an encryption, no additional work was required apart from allowing the user to change this value with setters and getters. The rotation, however, can and will change in an encryption and so the rotor representation required a function to simulate this rotational effect as well as a method to identify once the rotor is at its turnover position (See Code Block 5). Although it is not directly used in the rotor class, each rotor also had its turnover position (turnoverPosition) represented as an integer which could be used for the machine model in the subsequent section.

FUNCTION rotate() {

currentRotation = (currentRotation + 1) MOD 26

}  
  
FUNCTION isAtTurnoverPosition() {  
 IF (currentRotation EQUALS turnoverPosition) {

RETURN true

}

ELSE {

RETURN false

}

}

Code Block Two functions to simulate the rotors rotations

With these properties implemented, a function could be employed that applies the rotors transformation to a given letter and direction (right-to-left or vice versa). This function is analogous to that of Equation 4 where is modelled by wiring and is modelled by reverseWiring (See Code Block 6).

FUNCTION encode(characterIndex, DIRECTION) {

rotationShift = currentRotation – ringSetting

IF (DIRECTION EQUALS FORWARD) {

RETURN (wiring[(characterIndex + rotationShift + 26) MOD 26] –

rotationShift + 26) MOD 26

}

IF (DIRECTION EQUALS BACKWARD) {

RETURN (revrseWiring[(characterIndex + rotationShift + 26) MOD 26] –

rotationShift + 26) MOD 26

}

RETURN ERROR

}

Code Block Rotor encode function, analogous to Equation 4

### Enigma

With all the core functionality of each component in place, the Enigma model can be created. This model consists of 3 attributes: an array of rotors, a plugboard object and a reflector object. In order to accurately recreate the functionality of Enigma, two key functions were developed: encode and rotate. The encode function uses an input character and performs the encryption steps as shown in Equation 2. Pseudocode of this function can be seen in Code Block 7.

FUNCTION encode(characterIndex) {

IF (characterIndex IS NOT LETTER) {

RETURN characterIndex

}

rotate()

outputCharacter = plugboard.enccode(characterIndex)

FOR (i IN rotors.length ; i++) {

outputCharacter = rotors[i].encode(outputCharacter, FORWARD)

}

outputCharacter = reflector.encode(outputCharacter)

FOR (i IN rotors.length ; i--) {

outputCharacter = rotors[i].encode(outputCharacter, BACKWARD)

}

outputCharacter = plugboard.encode(outputCharacter)

RETURN outputCharacter

}

Code Block Enigma encryption steps

As mentioned previously, the rotation turnover position of the rotor played no part in the context of the rotor. For the Enigma model however, the turnover position of each rotor dictates when the other rotors rotate. The function to control the Enigma’s rotation behaviour can be seen in Code Block 8. Ensuring this functionality was correct was vital for an accurate representation of Enigma, as a single out-of-place rotation would cause extremely different results.

FUNCTION rotate() {

doubleStepped = false

// The rare case of double stepping (the left and middle rotor rotate when the middle is at a turnover)

IF (rotors[MIDDLE\_ROTOR].isAtTurnoverPosition()) {

rotors[MIDDLE\_ROTOR].rotate()

rotors[LEFT\_ROTOR].rotate()

doubleStepped = true

}

// Rotate middle rotor if right-most rotor is at turnover

IF (rotors[RIGHT\_ROTOR].isAtTurnoverPosition() AND NOT doubleStepped) {

rotors[MIDDLE\_ROTOR].rotate()

}

// Right-most rotor rotates every key press

rotors[RIGHT\_ROTOR].rotate()

}

Code Block The Enigma's rotation mechanism

Part of the requirement for this Enigma model was to also allow it to be fully configurable with its settings. To enable this, public functions within the Enigma class were created to act as an interface to interact with the machine. These functions make subsequent function calls to the instances of each component stored within the Enigma instance, allowing for full control over the Enigma settings such as rotor rotation, plugboard cabling and reflector selection.

### EnigmaPlus

In order to create the EnigmaPlus model posed in 3.4.2, a readjustment was needed to the contents of the Enigma package. This is since the aim of EnigmaPlus was to create a stronger cypher, whilst still retaining key properties of the Enigma. Therefore, the class hierarchy shown in Figure 10 was reconfigured to include inheritance in which Enigma and EnigmaPlus would inherit properties from a base class as in Figure 11. This helped reduce code duplication as well as ensuring that both models work similarly.

A diagram of a computer program

Description automatically generated

Figure A UML diagram showing both Enigma and EnigmaPlus

The encryption process of EnigmaPlus needed to work differently from that of Enigma to ensure its cryptographic strength. In this model, encoding and decoding are two distinct operations whereas in Enigma, they are the same. Consequently, both encode and decode functions were to be developed separately in which one is the inverse of the other. The pseudocode for encode can be seen in Code Block 9; the decode function is nearly identical however each letter transformation is done in reverse.

FUNCTION encode(characterIndex) {

IF (characterINDEX IS NOT LETTER) {

RETURN characterIndex

}

rotate()

outputCharacter = plugboard.encode(characterIndex)

FOR (i IN rotors.length ; i++) {

outputCharacter = rotors[i].encode(outputCharacter, FORWARD)

}

RETURN outputCharacter

}

Code Block EnigmaPlus encryption function

### Additional Features

In addition to the two models, numerous additional classes were added to the Enigma package to enable others to create their own code using this package more easily. Whilst these are not core components to the Enigma machine, they are still extremely useful to have and offer information regarding the machine.

Two logger classes were implemented that log both the Enigma and EnigmaPlus. These loggers are static classes that are utilised by both models. When either of the models are used for encryption/decryption, a flag can be set to log the encryption, providing a step-by-step outline of the processing that took place in the operation. The information provided by these loggers includes the rotation at any given letter’s encryption, the letter scrambling that took place in each component for a given encryption and all other information regarding the state of the model.

In addition to the loggers, additional functionality was implemented into both models, allowing a list to represent all wiring connections in the model at any given time. The algorithm that generates this list is an enumeration technique, in which each letter of the alphabet is encrypted using the model and the scrambling of the letter is recorded each time. This addition allows for a complete picture of the model to be generated at any time and in fact is used to generate the wiring diagrams in the GUI mentioned later.

Finally, two factory classes were created to provide a simple method for creating custom rotors and reflectors. These are static classes that serve only one purpose, to take the necessary parameters and build an instance of the respective object type. Whilst it allows custom component creation, they also employ buildPresetRotor and buildPresetReflector functions, that both use a name parameter to build a hard-coded component. These hard-coded components were added to reflect real-world rotors and reflectors that are mentioned in Table 1 and Table 2 respectively. In addition, it was decided that these factory classes would contain all error-handling and constraint checking that is involved in the creation of components. This helps maintain the authenticity of components such that they match the real-world examples, whilst also maintaining cleaner code.

## Design of the Applications

The applications to be implemented were a command line interface (CLI) and a graphical user interface (GUI) that allowed the user to interact with the simulation models. The subsequent sections provide further detail into both applications (GUI and CLI package) however this section gives an overview of the entire system as well as common functionality. Figure 12 provides a diagram created during the design process, demonstrating the packages that provide the core functionality to both applications. For each application, there was four packages associated with it. Firstly, the application package (CLI or GUI depending on the application) contained all necessary code to create and control the application. Secondly, the Enigma package would be used in both applications to act as the model in the simulation. Finally, the Config package would contain numerous XML files allowing to user to add, remove and configure enigma components and settings which could then be parsed and stored in Parsers.

A diagram of a computer

Description automatically generated

Figure A diagram demonstrating the structure of the underlying system and interactions for both applications

Whilst designing the project, it was decided that implementing a system that allowed for custom rotor and reflector creation would be of value to the user. For anyone who simply wanted to use the Enigma at a surface level, rotor and reflector configurations would be provided using the factory classes and the Config package could largely be ignored. If the user desired to stretch themselves and customise the Enigma far beyond what the physical machine was capable of, then they could do so by editing the three XML files (rotor\_bank.xml, reflector\_bank.xml and enigma\_settings.xml) within the Config package. The Parsers package was created to parse and store the contents of these XML files. The first two of these files allow the user to input their own definition of the respective component as in Code Block 10.This functionality is used in both the CLI and GUI and the information in these two XML files is read at run-time. The enigma\_settings.xml file is used exclusively for the CLI to allow for a simpler configuration experience. This file also allows the use of custom components created by the user as shown in Code Block 11 which shows an example of an Enigma configuration using custom components.

<rotor\_bank>

<rotor>

<name>MyCustomRotor</name>

<encoding>ZYXWVUTSRQPONMLKJIHGFEDCBA</encoding>

<turnover\_position>E</turnover\_position>

</rotor>

...

</rotor\_bank>

Code Block rotor\_bank.xml contents showing an example custom rotor called "MyCustomRotor"

<enigma>

<plugboard encoding="AM FI NV PS TU WZ"></plugboard>

<rotor>

<name>MyCustomRotor</name>

<ring\_setting>22</ring\_setting>

<start\_position>L</start\_position>

</rotor>

<rotor>

<name>I</name>

<ring\_setting>13</ring\_setting>

<start\_position>B</start\_position>

</rotor>

<rotor>

<name>II</name>

<ring\_setting>24</ring\_setting>

<start\_position>A</start\_position>

</rotor>

<reflector>

<name>UKW-A</name>

</reflector>

</enigma>

Code Block enigma\_settings.xml contents depicting the starting settings of the machine

### Command Line Interface

As in the requirements, the CLI was to be designed as a basic and simple application. Consequently, the code that underpins this application is short and simple. All data such as custom rotors and reflectors in the Config package is parsed and stored such that the system will recognise these components. As mentioned earlier, it was decided that the user would be required to configure the Enigma’s settings through enigma\_settings.xml to enable a simpler CLI. It was decided that the user should first be presented with the settings of the Enigma machine before encryption, and then prompted to enter a plaintext message. This input message would then be encrypted by the Enigma model and the corresponding cyphertext is displayed. Code Block 12 shows an example of the expected interaction with the CLI given the settings are identical to that in Code Block 11.

Plugboard : [AM FI NV PS TU WZ]

Reflector : UKW-A (EJMZALYXVBWFCRQUONTSPIKHGD)

Right Rotor : III

Rotation : L

Ring Setting : 22

Encoding : BDFHJLCPRTXVZNYEIWGAKMUSQO

Middle Rotor : I

Rotation : B

Ring Setting : 13

Encoding : EKMFLGDQVZNTOWYHXUSPAIBRCJ

Left Rotor : II

Rotation : A

Ring Setting : 24

Encoding : AJDKSIRUXBLHWTMCQGZNPYFVOE

Enter plaintext message:

HELLO WORLD

Cyphertext

JCUGQ KVBVF

Code Block An example of the expected interaction of the CLI

To accomplish this, Java was used to create a EnigmaSimulatorCLI main class, which could be compiled and ran. As explained previously, this application imports the Enigma and Parsers packages in order configure and simulate the Enigma machine.

### GUI

The GUI was by far the most complicated and time consuming task of the project.

### Visualization

A second part to the GUI, explain the design choices and implementation as well as a section “interpreting the diagram.”

# Results

# Evaluation

## Model Evaluations

* Test Coverage
* Actual German messages
* Permutations compared with improved model.
* Compare to other products.

## GUI Evaluation

* Usability?
* Cross-platform?

# Conclusion

# References

n.d. “Enigma.” *Crypto Museum.* Accessed 2024. https://www.cryptomuseum.com/crypto/enigma/.

n.d. *Enigma Machine.* https://cryptii.com/pipes/enigma-machine.

n.d. *Enigma Machine Emulator.* Accessed 2024. https://www.101computing.net/enigma-machine-emulator/.

n.d. “Enigma wiring.” *Crypto Museum.* Accessed 2024. https://www.cryptomuseum.com/crypto/enigma/wiring.htm.

Grime, J. 2013. “Maths from the talk "Alan Turing and the Enigma Machine".”

Hamer, David H. 1997. “Enigma: Actions involved in the ‘double stepping’of the middle rotor.” *Cryptologia 21, no. 1* 47-50.

n.d. “How does the Enigma machine work?” *Crypto Museum.* Accessed 2024. https://www.cryptomuseum.com/crypto/enigma/working.htm.

MacWright, Tom. n.d. *Enigma Machine.* Accessed 2024. https://observablehq.com/@tmcw/enigma-machine.

Ostwald, Olaf. 2023. “Cryptographic design flaws of early Enigma.”

Owen, Jaren. 2021. “How did the Enigma Machine work?” *Youtube.*

Rejewski, Marian. 1980. “An application of the theory of permutations in breaking the Enigma cipher.” *Applicationes Mathematicae. 16, No. 4,* 543-559.

Smart, Nigel P. 2016. “The enigma machine.” *Cryptography Made Simple* 133-161.

Tang, Lynda, Nayoung Lee, and Sophie Russo. 2018. “Breaking Enigma.”

Thimbleby, Harold. 2016. “Human factors and missed solutions to Enigma design weaknesses.” *Cryptologia 40, no. 2* 177-202.

van Manen, Dirk Jan, and Johan O.A. Robertsson. 2016. “Codes and ciphers.” *Part I: Decoding nature’s disorder: GeoExpro 13* 38-41.

n.d. *Virtual Enigma.* Accessed 2024. https://enigma.virtualcolossus.co.uk/VirtualEnigma/.

# List of Figures

[Figure 1 A photo of Enigma I showing the plugboard, keyboard and lampboard. Only a small portion of the rotors are visible, and the reflector is hidden (Enigma n.d.). 5](#_Toc158033854)

[Figure 2 A front-facing photo of the Enigma I plugboard with cables in place (Enigma n.d.) 7](#_Toc158033855)

[Figure 3 Wiring diagram to show a plugboard's potential encoding for a reduced alphabet size e.g. A encodes to D and vice versa. Notice the plugboard wiring diagrams tend to create X shapes. 8](#_Toc158033856)

[Figure 4 Wiring diagram to show an example reflector encoding for a reduced alphabet size e.g. A encodes to F and vice versa. 9](#_Toc158033857)

[Figure 5 Two Wiring diagrams to show an example rotor encoding. The diagram on the right shows the same rotor as on the left, but with a rotation of one. The left diagram shows F encoded to F and the diagram on the right shows E encoded to E. The “wire” is still the same, but the rotation of the rotor will lead to an F input/output, interpreted as E input/output. 10](#_Toc158033858)

[Figure 6 A wiring diagram representing a single state of an Enigma I machine. Input is received on the right-hand side before being scrambled by components in the following order: plugboard, rotor III, rotor II, rotor I, reflector UKW-B, rotor I, rotor II, rotor III, plugboard. The electrical signal received back from the plugboard represents the encoded letter. The names of each rotor/reflector do not match the encodings but are given as an example. 11](#_Toc158033859)

[Figure 7 An example of an encryption/decryption taking place in an Enigma I machine. In this case A is encoded to C. 11](#_Toc158033860)

# List of Tables

[Table 1 "Enigma I" rotor encodings (Enigma wiring n.d.) 7](#_Toc159928316)

[Table 2 "Enigma I" reflector encodings (Enigma wiring n.d.) 8](#_Toc159928317)

# List of Equations

[Equation 1 Permutations for rotors (excluding ring setting) 7](#_Toc159928326)

[Equation 2 The encryption steps of Enigma I 9](#_Toc159928327)

[Equation 3 An equation and additional constraints to describe the behaviour of an Enigma reflector 10](#_Toc159928328)

[Equation 4 A function to represent the encoding behaviour of the rotor where x and x` are letters, represents the rotor’s rotation and represents the ring setting 10](#_Toc159928329)

[Equation 5 Definition of double factorial 12](#_Toc159928330)

[Equation 6 The number of settings (key space) of Enigma I assuming 10 plugboard cables are used. Ring setting is omitted as it was not changed by the Germans. Based on work from Tang, Lee and Russo (Tang, Lee and Russo 2018). 12](#_Toc159928331)

# List of Code Blocks

[Code Block 1 Check reflector encoding meets constraints 16](#_Toc159929943)