Enigma Machine Simulator

Elliot Brooks

Supervisors:

April 2024

# Abstract

Contents

[1 Abstract 2](#_Toc161218631)

[2 Introduction 5](#_Toc161218632)

[2.1 Aims and Objectives 5](#_Toc161218633)

[3 History & Background 6](#_Toc161218634)

[3.1 History of the Machine 6](#_Toc161218635)

[3.2 Design of the Machine 6](#_Toc161218636)

[3.2.1 Keyboard & Lampboard 6](#_Toc161218637)

[3.2.2 Rotors 7](#_Toc161218638)

[3.2.3 Reflector 7](#_Toc161218639)

[3.2.4 Plugboard 8](#_Toc161218640)

[3.3 Enigma Machine and Abstraction 8](#_Toc161218641)

[3.3.1 Enigma’s Encryption 8](#_Toc161218642)

[3.3.2 Abstractions 9](#_Toc161218643)

[3.4 Design Flaws & Remedies 12](#_Toc161218644)

[3.4.1 Cypher Strength 12](#_Toc161218645)

[3.4.2 Improving the Machine 12](#_Toc161218646)

[3.5 Related work 13](#_Toc161218647)

[4 Design & Implementation 15](#_Toc161218648)

[4.1 Tools & Technology 16](#_Toc161218649)

[4.2 Enigma and EnigmaPlus 16](#_Toc161218650)

[4.2.1 Enigma 16](#_Toc161218651)

[4.2.2 EnigmaPlus 19](#_Toc161218652)

[4.2.3 Model Tests 20](#_Toc161218653)

[4.2.4 Additional Features 20](#_Toc161218654)

[4.3 Config and Parsers 21](#_Toc161218655)

[4.4 CLI 22](#_Toc161218656)

[4.5 GUI 23](#_Toc161218657)

[4.5.1 UI Design 23](#_Toc161218658)

[4.5.2 Model-View-Controller Architecture 25](#_Toc161218659)

[4.5.3 Visualisation 25](#_Toc161218660)

[5 Results 28](#_Toc161218661)

[6 Evaluation 28](#_Toc161218662)

[6.1 Model Evaluations 28](#_Toc161218663)

[6.2 GUI Evaluation 28](#_Toc161218664)

[7 Conclusion 28](#_Toc161218665)

[8 References 28](#_Toc161218666)

[9 List of Figures 29](#_Toc161218667)

[10 List of Tables 30](#_Toc161218668)

[11 List of Equations 30](#_Toc161218669)

[12 List of Code Blocks 30](#_Toc161218670)

# 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. This project began with thoroughly researching *Enigma I* to understand the inner workings of the machine and to compile this research into a simpler, abstracted model using Java. Both a multi-platform command line interface (CLI) and a multi-platform graphical user interface (GUI) were developed 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. The accuracy of the model was identified by comparing with similar products as well as some real-world messages found amongst the German forces during WW2. In addition, the effectiveness of the two Enigma models was identified with some basic cryptoanalysis.

## Aims and Objectives

The following list adapted from the original project description, however there are some additional objectives that were self proposed. For each task, a criteria was determined 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 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 be easy to use and reflect the layout of the physical 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 section 3.4) (Thimbleby 2016).

An old machine on a glass table

Description automatically generated

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. 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 three 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 1 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 2 "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 2 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 section 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 2 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 3 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 3 An equation and additional constraints to describe the behaviour of an Enigma reflector

A diagram of a diagram

Description automatically generated

Figure 4 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 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

A diagram of lines and dots

Description automatically generatedA diagram of lines and dots

Description automatically generated

Figure 5 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 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 performing a loop in the reflector. 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 7 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 three 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 5 Definition of double factorial

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).

### 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 three 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 three 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 section 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 three 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 8 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 9 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 aspects of this projects were inspired by this webpage

“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 key goal of this project was to create an exact simulation of the Enigma machine and provide both a command line interface and a graphical user interface to interact with it, focussing on educating the user on the machine’s functionality. Also, an additional model named EnigmaPlus was to be developed maintaining compatibility with the GUI and CLI as well as providing a stronger cypher than the original machine. Figure 10 provides an overview of the underlying system developed throughout this project. The subsequent sections provide a detailed account of each package; however, an overview of the development is given in this section.

A diagram of a computer program

Description automatically generated

Figure 10 A diagram depicting the underlying system of this project

The Enigma package contains all the necessary backend functionality to accurately model and simulate the Enigma machine and EnigmaPlus. These models are designed to reflect the real-world machines (in the case of EnigmaPlus, the machine is hypothetical), encrypting any plaintext message into cyphertext and vice versa. In addition, the Enigma package also provides pre-built rotor and reflector configurations to the Parsers package. The Parsers package and Config package both aim to provide custom component creation for both the CLI and GUI. The packages also provide a means of configuring the Enigma machine exclusively for the CLI. The CLI package contains a small demo application allowing the user to quickly configure and operate the model of the Enigma machine. Independently, the GUI package provides the user with a Model-View-Controller (MVC) based graphical application allowing the user to configure and operate the machine. In addition, the GUI provides numerous additional tools aimed towards providing an informative representation of the encryption process through visualisation.

## Tools & Technology

Throughout development of this project, multiple tools and technologies were utilised to accomplish the requirements and objectives. Below gives a brief overview of the key tools and technologies and their use in this project.

Java was the chosen programming language used for this project due to its native support of object-oriented programming and its vast online support. In addition, two existing Java packages were used to support development. The first of these packages was JavaFX which is a graphics package specialising in user interfaces. This package was used to create the GUI as well as provide the visualisation tool included with the GUI due to its easy-to-use interface components and drawing capabilities. The second package was JUnit which is a testing framework that enables test cases to be created and ran automatically. In this project, JUnit was used to create unit tests to validate both Enigma and EnigmaPlus and their expected outputs, supporting the development process.

Gluon Scene Builder is an external application that provides a “drag and drop” experience for GUI creation. In this project, this was used to create prototypes for, and eventually implement, the GUI. The reason this tool was chosen as it generates FXML files which is natively supported by JavaFX, creating a simpler process GUI development.

The XML language is an object description language with the soul purpose of representing data. In this project, this was used to allow the user to create and store custom components as well as configure the settings of the Enigma machine.

Maven is a Java based build tool which aims to create a simpler process for the developer to build their Java project. In this project, Maven was used to build the project as well as manage all other dependencies (such as JavaFX) to help maintain a minimal code base.

## Enigma and EnigmaPlus

### Enigma

After conducting research on the Enigma machine, which is explained in the previous chapter, it was clear that the most elegant solution to simulating the machine was by adopting object-oriented programming. By doing so, each class can reflect the core definition of each physical component therefore creating an accurate encryption process. Consequently, the Enigma package was created, and its UML diagram is demonstrated within Figure 11. This diagram does not include EnigmaPlus, as this was an additional requirement that was accommodated much later in the development process. The diagram shows the developed Java classes for each component of the Enigma machine, which hold a composite relationship with the Enigma class, such that it cannot exist without these components. Interestingly, the UML diagram does not include any mention of the keyboard or lampboard that were covered in the background section. This is because whilst they were significant components for the physical machine, in a logical sense, they are nothing more than input and output.

A diagram of a computer program

Description automatically generated

Figure 11 UML diagram depicting the Enigma package, omitting EnigmaPlus

The majority of the components within Enigma are some form of substitution cypher therefore a common representation was needed for this type of cypher. At first, it was thought the best way to represent the substitution cypher encodings would be with the use of hash-maps. Whilst this representation would have worked, there would have needed to be 52 key-value pairs in the hash-map to provide this. In addition, hash-maps would have brought unnecessary complexity into the representation. Therefore, it was decided that a simple array for each component depicted as wiring, of length 26 integers, would be used for the representation. In such an array, each index represents the positional encoding of a plaintext letter and the value at that index would represent the positional encoding of the cyphertext letter. This method offers much more simplicity as well as maintaining fast index speeds, like the hash-map, a process that will be abundant in an encryption. For most of the components, this representation was enough, however the rotor required a more complex representation requiring a representation for its inverse wiring, rotation, turnover position and ring setting. Therefore, the Rotor class exhibits additional attributes: reverseWiring, currentRotation, turnoverPosition and ringSetting to represent these.

In addition to the components and the machine, the Enigma package contains two factory classes. These were implemented to encapsulate all rotor and reflector instance creation. This includes creation of pre-set components as shown in Table 1 and Table 2 which were hard coded into these factory classes, as well as creation of custom components. By encapsulating this functionality, it also allows for a clean and concise method for checking additional constraints on the component’s encodings. This is especially important for the reflector which, as previously mentioned in section 3.2.3, cannot contain self-coding and must abide by reciprocal coding.

With the representation of each component complete, the core Enigma model could be created. The attributes for this model are a list of rotors, a plugboard and a reflector. This properly reflects the components of the physical machine. In order to accurately recreate the encryption process, two key functions were developed, encode and rotate. The former takes a plaintext character as input and returns a cyphertext character as output whilst following the encryption steps formulated previously in Equation 2. This function only encodes one character; however, this can be simply extended to a whole message using iteration. A flowchart depicting the developed encryption process for a single character can be seen in Figure 12. The flowchart shows a call to the rotate function which is presented as pseudocode in Code Block 1, before subsequently calling each components encode function (scrambling the letter) in the following order: plugboard, right rotor, middle rotor, left rotor, reflector, left rotor, middle rotor, right rotor, plugboard.

A diagram of a character

Description automatically generated

Figure 12 A flowchart depicting the Enigma models encryption/decryption steps

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 1 The Enigma's rotation mechanism demonstrated by pseudocode

### EnigmaPlus

At a later point in the design and implementation process, it was decided that the EnigmaPlus model would be developed alongside the original machine. As such a readjustment was made to the structure of the Enigma package as demonstrated in Figure 13. This new UML diagram shows both EnigmaPlus and Enigma inheriting from a base class named RotorMachineBase. This class was included to help reduce the amount of duplicated code within the package as well as ensure the two models work in similar fashions.

A diagram of a computer program

Description automatically generated

Figure 13 A UML diagram showing both Enigma and EnigmaPlus

The differences between Enigma and EnigmaPlus are small but important. The encryption process of EnigmaPlus worked differently, in which encoding and decoding are two distinct operations. Consequently, both an encode and decode function were developed in which one is the inverse of the other. Figure 14 shows a flowchart depicting both the encode and decode function for EnigmaPlus. By observing the steps of these algorithms carefully, it becomes clear that the decode function applies each components letter scrambling in the opposite order.

A diagram of a character

Description automatically generated

Figure 14 EnigmaPlus encryption and decryption flowcharts

### Model Tests

During implementation of both the Enigma and EnigmaPlus models, it was decided that a suite of unit tests would be developed to validate the inputs and outputs of the models to aid development. For both models, ensuring that a message can be encoded into cyphertext as well as decoded back to the original message was essential. In addition, both models were required to be fully configurable. Consequently, a series of tests were designed and implemented to validate the models against these requirements, the results of which are covered in chapter 5. Specifically for the Enigma model, the requirements also state that the model should accurately reflect the output of the real-world machine. To validate this, it was decided that using the output cyphertext generated by the similar products would be used to ensure accuracy due to the difficulty of gaining access to the real-world machine. After performing some testing on the multiple similar product (Enigma Machine Emulator n.d.) (Enigma Machine n.d.)it was found that once configured in the same way output (using rotors I, II and III all in rest position, the reflector choice being UKW-B and no plugboard connections), all products produced the same output (“AAAAA” encoded to “BDZGO”). Table 3 shows a snippet of a test plan that was followed in order to meet the requirements mentioned in this section.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Test Steps | Expected Result |
| Enigma/EnigmaPlus encryption + decryption test | A test to validate whether a given message can be encoded to cyphertext, and then decoded back to the original image given the same settings for the Enigma model | 1. Configure model to chosen settings  2. Encode chosen message  3. Record cyphertext  4. Decode cyphertext  5. Check output of the decryption matches original message | Input message matches the decryption output |
| Enigma accuracy test | A test to validate that the Enigma model creates the same output as other Enigma related products | 1. Encode a chosen message  2. Compare output to several examples extracted from similar products | Output of the encryption matches that of similar products |
| Enigma/EnigmaPlus configuration test | A test to validate that both Enigma and EnigmaPlus have fully configurable settings such as rotors or plugboard | 1. Setup each model with chosen settings  2. Attempt to change the rotor settings (rotation, ring, selection), reflector choice and plugboard wiring  3. Validate these changes have taken place | All components can be configured, and the respective changes take place |

Table 3 A snippet of a test plan for validation of the Enigma and EnigmaPlus models

To implement these tests, the Java testing framework JUnit was used to create a test package with the focus of testing the components of the project. This package exists outside the scope of the overview of the project given at the beginning of this chapter. The contents of Table 3 is not exhaustive, and simply provides a snippet of the tests included in two Java test classes named EnigmaTest and EnigmaPlusTest.

### Additional Features

In addition to the models developed in this project, numerous classes were added to the Enigma package to create ease-of-use. Whilst these features are not core components of Enigma, they were included in this section as they are useful to have whilst developing applications (such as the CLI and GUI in this project) by offering interesting information regarding the model. Some of these features are not present in previous UML diagrams for simplicity as well as being added much later in the development process.

Two logger classes were implemented that log both Enigma and EnigmaPlus. These loggers are static classes which are utilized by both models so that they can provide a step-by-step outline of the encryption/decryption process. The loggers also provide information such as the current rotation at any given frame in the encryption, the effect that each component has on an input letter and all other information regarding the state of the model.

Whilst developing the GUI for this project, it was decided that additional functionality was needed for each model. This new functionality generates a list of all wiring connections in the model at any given frame by using enumeration, in which each letter of the alphabet is encrypted (without rotation applied in this case) and the scrambling of the letter is recorded at each component. This addition allowed for a complete picture of the model to be generated at any time and was used primarily in the visualisation tool within the GUI.

## Config and Parsers

The Config and Parsers packages were implemented to create an XML based miniature database within the project. The Config package contained three files, rotor\_bank.xml, reflector\_bank.xml and enigma\_settings.xml which allowed the user to configure the settings for the Enigma machine. The Parsers package contained several XML parsers to parse and store the contents of the files stored within the Config package. As per the requirements of this project, the machine needed to be fully configurable akin to the physical machine. However, during the implementation of the Enigma package it was decided that a system allowing the user to configure the machine beyond the capabilities of the original machine would be beneficial.

Both rotor\_bank.xml and reflector\_bank.xml were created to allow the user to create their own definitions of the corresponding components much like the definitions shown in Table 1 and Table 2. Code Block 2 shows an example of an entry a user could create in rotor\_bank.xml, creating a new rotor called “MyCustomRotor”. Once this has been done, at runtime of either the CLI or GUI application, the Parsers package will create rotor objects based on the contents of this file and store it in ComponentCache along with pre-configured components. This cache is then used in both the CLI and GUI for configuration.

<rotor\_bank>

<rotor>

<name>MyCustomRotor</name>

<encoding>ZYXWVUTSRQPONMLKJIHGFEDCBA</encoding>

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

</rotor>

...

</rotor\_bank>

Code Block 2 rotor\_bank.xml contents showing an example custom rotor called "MyCustomRotor"

The enigma\_settings.xml file was created exclusively for the CLI for a simpler configuration experience. In addition, it also allows the use of any custom components that are stored within ComponentCache at runtime. Code Block 3 shows an example configuration of the Enigma using the custom component shown in Code Block 2

<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 3 enigma\_settings.xml contents depicting the start settings of the machine

## CLI

As per the requirements, the CLI was to be developed as a basic and simple application which acts more like a tool rather than an educational experience. Consequently, the code that underpins this application is relatively short and simple. As previously mentioned, when using the CLI the configuration of the Enigma model takes place within the Config package, specifically enigma\_settings.xml as demonstrated in Code Block 3. Originally, a multi-faced menu approach was experimented with to allow for the user to configure the machine. However, due to the complexity of the Enigma’s key, this led to a tiresome and confusing experience during the setup of the model. The Config package approach allowed for easier editing as well as enabling the settings of the Enigma to be stored.

With the configuration mechanism developed, design regarding what information the user should be provided with in the CLI could begin. Firstly, it was decided that the starting settings of the Enigma should be displayed to the user, to ensure that they understand the settings that were used to encrypt their message. Secondly, the user should be prompted to enter a message before the correct cyphertext (or plaintext if this is a decryption) is displayed. Code Block 4 shown an example of the expected interaction with the CLI, showing the chosen settings being most of the text, and showing the encryption result of the message “HELLO WORLD”. The same output would be expected if the user intended to decode a message, however the input “HELLO WORLD” would be replaced with “JCUGQ KVBVF” and vice versa.

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 4 An example of the expected interaction with the CLI

The functionality to accomplish the requirements of the CLI was achieved by creating the EnigmaSimulatorCLI class, which acted as a runnable Java class.

## GUI

### UI Design

The GUI was one of the larger requirements for this project. The goal was to create an application that could be used to configure and operate the Enigma machine as well as exhibiting an easy-to-understand user interface. As explained in section 3.5, numerous other products attempting to achieve the same thing exist. After experimenting with the other products, it was decided that the UI would mostly avoid reflecting a physical representation of the machine and adopt a simpler approach. This is because the Enigma machine’s layout is rather obscure and creating something like this approaches the limitations of the chosen graphics package. Figure 15 shows an early mock-up of the design of the GUI, which uses a top-to-bottom approach for user interaction. The idea behind this design was that the user would configure the reflector, rotors and plugboard before entering their message on an on-screen keyboard, displaying the cyphertext message below as they entered their message.

A screenshot of a computer

Description automatically generated

Figure 15 An early mock-up design for the GUI

At this point in the design, the modality of inputs needed to be considered. It was decided that rotors the reflector would be selected by drop-down menus, allowing the user to select one of the rotors stored within ComponentCache. In order to stick with the simple design of the interface, it was decided that the plugboard would consist of a text field in which the user would have to enter cable pairs such as “AB CG OI” to represent the connections. The other option would be to create a clickable interface like the online Enigma emulator (Enigma Machine Emulator n.d.) however this proved difficult to implement without creating interface clutter. Finally, it was decided that the message input would consist of an on-screen keyboard, and each keypress would cause the encoded letter to appear in the “Message Display” text field.

To aid the development of the GUI, Gluon Scene Builder was used to quickly create a high-fidelity prototype of the user interface. This tool creates FXML files (a variation of XML) which can be used alongside JavaFX. Figure 16 shows this UI prototype, which exhibits a number of differences to the original design. The biggest difference is the input mechanism, which was changed from an on-screen keyboard to a simple text field. Whilst trying to build this prototype, it was decided the previously envisioned on-screen keyboard would create too much clutter in the application as well as provide a large challenge for implementation. Additionally, a large text field was added at the bottom to display a log of the encryption. This was to provide information along each step of an encryption process and later became the foundation for the visualisation.

A screenshot of a computer

Description automatically generated

Figure 16 A high-fidelity prototype of the user interface provided with the GUI

To incorporate the EnigmaPlus model into the GUI, it was decided that an additional tab would be added to the top of the interface, to allow the user to switch between models. By switching models, the user is presented with a slightly different interface for the configuration of the machine. Figure 17 shows a high-fidelity prototype of the setting configuration panel for EnigmaPlus, which can be compared to that of Figure 16. The key different between these two panels, is that the EnigmaPlus does not refer to a reflector at all and an additional option allowing the user to choose which operation to perform on the plaintext.

A screenshot of a computer

Description automatically generated

Figure 17 The configuration settings for EnigmaPlus within the GUI

### Model-View-Controller Architecture

In order to create clean and easily modifiable code, it was decided that the GUI should follow the model-view-controller (MVC) architecture. This architecture aims to separate an applications code into three components: model, view, and controller. In this project’s case, the model is simply the Enigma package and so no additional work was needed here. For the view, Gluon Scene Builder combined with minimal code could provide FXML files to define and instantiate GUI components such as buttons and labels. Finally, an additional class was created within the GUI package known as EnigmaController which handles any communication between the view and the model, thereby acting as the controller. Figure 18 shows how the MVC architecture has been applied to this project.

A diagram of a user

Description automatically generated

Figure 18 The MVC architecture applied to this project

### Visualisation

The final requirement with the GUI was to provide the user with a visualisation of the encryption process, with the aim of educating the user on the workings of Enigma. As demonstrated in section 3.3.2, the simplest way to understand the Enigma is through abstractions, often taking the form of wiring diagrams. Therefore, a visualisation tool was developed and implemented into the GUI which generates wiring diagrams like Figure 7 for each frame of the user’s encryptions.

The first task to complete to support this visualisation was to determine how to integrate it into the currently existing GUI. It was decided that an additional tab would be added to the bottom of the interface. Figure 19 shows another high-fidelity prototype of the GUI, this time including the “Visualisation” tab along with other changes. The idea behind this was to display the wiring diagram of the encryption in the large blank space, providing additional information above (encryption and current rotation) and allowing the user to switch frames and display additional wires not directly used in the encryption (“<” and “>” buttons along with “Show all wires” check box).

A screenshot of a computer

Description automatically generated

Figure 19 A high-fidelity prototype of the GUI including the “Visualisation” tab

With the placement of the visualisation decided, the visualisation method could be developed. In order to draw a diagram programmatically with various shapes, a JavaFX canvas object had to be used. This object allows for shapes and lines to be drawn via a coordinate system however, all reference to any shapes created is lost. To encapsulate this functionality, two additional files called EnigmaVisualiser and EnigmaPlusVisualiser were created to control all aspects of the visualisation. The code within these classes consisted mostly of geometric calculations to generate a legible and informative diagram. As mentioned earlier, the generated wiring diagrams had to look like that of Figure 7 but instead they must the whole 26 letter alphabet. Figure 20 shows a mock-up example wiring diagram depicting the minimum requirement of the visualisation. In this diagram, the red coloured lines represent current flowing right-to-left, and the blue coloured lines represent current flowing left-to-right such that the plaintext character “A” is encoded to “W”. In addition to what is presented in the figure, it was also decided that all other internal wires in the given encryption frame should be shown as well (by utilising the functionality mentioned in section 4.2.3), but this is not shown in the figure. The actual results of this visualisation tool and further discussion are expanded on in chapter 5.

A diagram of a train

Description automatically generated with medium confidence

Figure 20 A mock-up example of the diagram that should be generated by the visualisation tool

# Results

A diagram of a graph

Description automatically generatedA diagram of a graph

Description automatically generated

# 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.

n.d. *Science Museum.* Accessed 2024. https://www.sciencemuseum.org.uk/home.

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. Photo taken by author at Science Museum London (Science Museum n.d.) 6](#_Toc161152530)

[Figure 2 A front-facing photo of the Enigma I plugboard with cables in place (Enigma n.d.) 8](#_Toc161152531)

[Figure 3 A wiring diagram to show a plugboards potential encoding 9](#_Toc161152532)

[Figure 4 A wiring diagram to show a potential encoding for a reflector 10](#_Toc161152533)

[Figure 5 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 11](#_Toc161152534)

[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 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. 11](#_Toc161152535)

[Figure 7 An example of an encryption/decryption taking place in an Enigma I machine. In this case A is encoded to C. 11](#_Toc161152536)

[Figure 8 A circuit showing an Enigma style machine avoiding both self-coding and reciprocal coding. Taken from Figure 6 (Thimbleby 2016) 13](#_Toc161152537)

[Figure 9 A wiring diagram depicting "EnigmaPlus". Note that there is no reflector as well as encoding and decoding take place in opposing directions. 13](#_Toc161152538)

[Figure 10 A diagram depicting the underlying system of this project 15](#_Toc161152539)

[Figure 11 UML diagram depicting the Enigma package, omitting EnigmaPlus 17](#_Toc161152540)

[Figure 12 A flowchart depicting the Enigma models encryption/decryption steps 18](#_Toc161152541)

[Figure 13 A UML diagram showing both Enigma and EnigmaPlus 19](#_Toc161152542)

[Figure 14 EnigmaPlus encryption and decryption flowcharts 20](#_Toc161152543)

[Figure 15 An early mock-up design for the GUI 23](#_Toc161152544)

[Figure 16 A high-fidelity prototype of the user interface provided with the GUI 24](#_Toc161152545)

[Figure 17 The configuration settings for EnigmaPlus within the GUI 24](#_Toc161152546)

[Figure 18 The MVC architecture applied to this project 25](#_Toc161152547)

[Figure 19 A high-fidelity prototype of the GUI including the “Visualisation” tab 26](#_Toc161152548)

[Figure 20 A mock-up example of the diagram that should be generated by the visualisation tool 27](#_Toc161152549)

# List of Tables

[Table 1 "Enigma I" rotor encodings (Enigma wiring n.d.) 7](#_Toc160964776)

[Table 2 "Enigma I" reflector encodings (Enigma wiring n.d.) 8](#_Toc160964777)

# List of Equations

[Equation 1 Permutations for rotors (excluding ring setting) 7](#_Toc161152554)

[Equation 2 The encryption steps of Enigma I 9](#_Toc161152555)

[Equation 3 An equation and additional constraints to describe the behaviour of an Enigma reflector 10](#_Toc161152556)

[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](#_Toc161152557)

[Equation 5 Definition of double factorial 12](#_Toc161152558)

[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](#_Toc161152559)

# List of Code Blocks

[Code Block 1 The Enigma's rotation mechanism demonstrated by pseudocode 18](#_Toc161152550)

[Code Block 2 rotor\_bank.xml contents showing an example custom rotor called "MyCustomRotor" 21](#_Toc161152551)

[Code Block 3 enigma\_settings.xml contents depicting the start settings of the machine 21](#_Toc161152552)

[Code Block 4 An example of the expected interaction with the CLI 22](#_Toc161152553)