University of Manchester

Project Report 2024

**The Enigma Simulator**

Author: Elliot Brooks

Supervisor: Dr Renate A. Schmidt

Abstract

The Enigma machine and the surrounding events, such as the breaking of the Enigma by Alan Turing and the Bletchley Park team, were extremely significant in the world of cryptography and computer science. Due to the fame of Enigma, there is a strong motivation to provide learning tools to support the understanding of the Enigma.

The project described in this report aims to accurately simulate the Enigma using Java, as well as provide two interfaces to interact with the model. Thorough research was undertaken in order to abstract and present the mechanism behind Enigma, as well as demonstrating insight into the flaws and remedies surrounding the cypher strength of Enigma.

The developed simulation includes several key features such as the Enigma model, the EnigmaPlus model, the command line interface, the graphical user interface, and the visualisation tool. All the features were developed with modern software engineering techniques such as object-oriented programming, model-view-controller architecture, and well-tested code to provide a complete Enigma experience. The results of the Enigma simulator demonstrate the accuracy and authenticity of the Enigma model. In addition, several screenshots of the command line interface, graphical user interface and visualisation tool are provided. These results were then critically evaluated through the use of genuine German messages from World War 2, as well as comparing the solution to similar tools.

This work contributes to the understanding of the Enigma by providing a complete Enigma experience. The proposed system requirements were achieved, and the system successfully implements the additional, self-proposed goals such as encouraging exploration through EnigmaPlus, custom components and real-time visualisations, thereby aiding the learning of users enthused by cryptography.

Acknowledgements

I would like to express my thanks to my supervisor Dr Renate Schmidt for the support and advice provided throughout the project. I also would like to deeply thank my family for their support and encouraging me to pursue my highest potential.

Contents

[1 Introduction 10](#_Toc164434278)

[1.1 Aims and Objectives 10](#_Toc164434279)

[1.2 Report Structure 12](#_Toc164434280)

[2 History and Background 13](#_Toc164434281)

[2.1 Design of the Enigma 13](#_Toc164434282)

[2.2 Enigma Machine and Abstraction 17](#_Toc164434283)

[2.3 Design Flaws and Remedies 22](#_Toc164434284)

[2.4 Related Work 26](#_Toc164434285)

[3 Design and Implementation 28](#_Toc164434286)

[3.1 Tools and Technology 31](#_Toc164434287)

[3.2 Enigma 31](#_Toc164434288)

[3.3 Config and Parsers 36](#_Toc164434289)

[3.4 Command Line Interface 37](#_Toc164434290)

[3.5 Graphical User Interface 38](#_Toc164434291)

[3.6 EnigmaPlus 45](#_Toc164434292)

[3.7 Model Tests 50](#_Toc164434293)

[4 Results and Evaluation 52](#_Toc164434294)

[4.1 Model Accuracy 52](#_Toc164434295)

[4.2 CLI, GUI and Visualisation 56](#_Toc164434296)

[4.3 Summary of Results 62](#_Toc164434297)

[5 Conclusion 63](#_Toc164434298)

[5.1 Summary 63](#_Toc164434299)

[5.2 Reflection 63](#_Toc164434300)

[5.3 Ideas for Future Work 65](#_Toc164434301)

[6 References 66](#_Toc164434302)

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 [5] 14](#_Toc164434303)

[Figure 2 A front-facing photo of the Enigma I plugboard with cables in place [1] 17](#_Toc164434304)

[Figure 3 A wiring diagram to show a plugboards potential encoding (A-F) 19](#_Toc164434305)

[Figure 4 A wiring diagram to show a potential encoding for a reflector (A-F) 20](#_Toc164434306)

[Figure 5 Two wiring diagrams to show a potential rotor encoding (A-F). The diagram on the right shows the same rotor as on the left, but with a rotation of one 21](#_Toc164434307)

[Figure 6 A wiring diagram representing a single state of an Enigma I machine (A-F) 22](#_Toc164434308)

[Figure 7 An example of an encryption/decryption taking place in an Enigma I machine (A-F). In this case A is encoded to C 22](#_Toc164434309)

[Figure 8 A circuit showing an Enigma style machine avoiding both self-coding and reciprocal coding. Taken from Figure 6 [4] 25](#_Toc164434310)

[Figure 9 A wiring diagram depicting "EnigmaPlus". Note that there is no reflector, in addition encoding and decoding take place in opposing directions. 26](#_Toc164434311)

[Figure 10 A diagram depicting the underlying packages for of this project, including Enigma, GUI, CLI, Config and Parsers 29](#_Toc164434312)

[Figure 11 UML diagram depicting the Enigma package 33](#_Toc164434313)

[Figure 12 A flowchart depicting the Enigma model’s encryption/decryption steps 34](#_Toc164434314)

[Figure 13 An early mock-up design for the GUI 40](#_Toc164434315)

[Figure 14 A high-fidelity prototype of the user interface provided with the GUI 42](#_Toc164434316)

[Figure 15 The MVC architecture applied to this project 43](#_Toc164434317)

[Figure 16 A high-fidelity prototype of the GUI including the “Visualisation” tab 44](#_Toc164434318)

[Figure 17 A mock-up example of the diagram that should be generated by the visualisation tool for Enigma 45](#_Toc164434319)

[Figure 18 The final UML diagram of the Enigma package, including EnigmaPlus 46](#_Toc164434320)

[Figure 19 A flowchart depicting the encryption/decryption steps of EnigmaPlus 47](#_Toc164434321)

[Figure 20 The configuration settings for EnigmaPlus within the GUI 48](#_Toc164434322)

[Figure 21 A mock-up example of the diagram that should be generated by the visualisation tool for EnigmaPlus 49](#_Toc164434323)

[Figure 22 A table demonstrating the test coverage of the Enigma package 52](#_Toc164434324)

[Figure 23 A collection of screenshots to show the “AAAAA” encryption using the Enigma model on Cryptii [1] 53](#_Toc164434325)

[Figure 24 A screenshot of the Enigma Emulator [14] to show the results of the “AAAAA” encryption 53](#_Toc164434326)

[Figure 25 A screenshot of the CLI output showing the results of the "AAAAA" encryption 54](#_Toc164434327)

[Figure 26 A genuine message encrypted by Enigma, taken from a collection of contemporary messages [24] 55](#_Toc164434328)

[Figure 27 A screenshot of the CLI depicting the decryption of a genuine Enigma message 55](#_Toc164434329)

[Figure 28 An example of the developed GUI being used for an Enigma encryption (Windows) 58](#_Toc164434330)

[Figure 29 A generated wiring diagram, showing all wire configurations in a given frame 59](#_Toc164434331)

[Figure 30 An example of the developed GUI being used to perform an EnigmaPlus encryption (Windows) 60](#_Toc164434332)

[Figure 31 A screenshot of the UI demonstrating rotor selection (MacOS) 61](#_Toc164434333)

List of Tables

[Table 1 Project requirements and success criteria 11](#_Toc164434334)

[Table 2 "Enigma I" rotor encodings [9] 15](#_Toc164434335)

[Table 3 "Enigma I" reflector encodings [9] 16](#_Toc164434336)

[Table 4 An exhaustive list of the packages developed for this project 30](#_Toc164434337)

[Table 5 Considered input modalities that were not implemented into the GUI 41](#_Toc164434338)

[Table 6 A snippet of a test plan for validation of the Enigma and EnigmaPlus models 51](#_Toc164434339)

List of Equations

[Equation 1 Permutations for rotors (excluding ring setting) 16](#_Toc164434340)

[Equation 2 The encryption steps of Enigma I 18](#_Toc164434341)

[Equation 3 An equation and additional constraints to describe the behaviour of an Enigma reflector 20](#_Toc164434342)

[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 21](#_Toc164434343)

[Equation 5 Definition of double factorial 23](#_Toc164434344)

[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 [12]. 23](#_Toc164434345)

List of Code Blocks

[Code Block 1 The Enigma's rotation mechanism demonstrated by pseudocode 35](#_Toc164434346)

[Code Block 2 rotor\_bank.xml contents showing an example custom rotor called "MyCustomRotor" 36](#_Toc164434347)

[Code Block 3 enigma\_settings.xml contents depicting the start settings of the machine 37](#_Toc164434348)

[Code Block 4 An example of the expected interaction with the CLI 38](#_Toc164434349)

# 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 reflector and a plugboard to redirect electrical signals, scrambling a plaintext message into cyphertext. Whilst most have heard of the machine, either from studies of cryptography or through history, many do not understand how the machine functions and its significance. By creating an accurate simulation and visual tool to demonstrate the inner workings of Enigma, the aim of this project was to not only offer a deeper insight into the functionality and significance, but also explain the weaknesses of the machine.

There already exists a handful of examples of Enigma simulations which offer a skeuomorphic approach to modelling the machine. The challenge of this project was to create a tool that provides a complete experience of Enigma for the user, including an accurate recreation supported by multiple applications. This project began with a thorough investigation into *Enigma I* to understand the inner workings of the machine and to compile this research into a simpler, abstracted model using Java. 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, a second model was developed called “EnigmaPlus” which aimed to correct the two key cryptographic weaknesses of the machine. The results regarding the accuracy of the models, as well as the outcome of the CLI and GUI, are presented in this report. Additionally, these results were evaluated through comparison to similar products and other metrics.

## Aims and Objectives

Table 1 shows a list of requirements that was adapted from the original project description, however there were some additional objectives that were self-proposed. For each task, a criterion was determined to assess the success of each task’s implementation.

|  |  |
| --- | --- |
| Aim / Objective | Criteria for Success |
| Create a Java package to accurately simulate the Enigma machine. | The code base within the package should conform to good software engineering practises to provide an accurate simulation of Enigma machine. In addition, the system should allow for Enigma to be fully configurable. |
| Create a new “EnigmaPlus” model, existing within the Enigma package, that improves upon the weaknesses of the original machine. | The EnigmaPlus model should reside within the Enigma package, functioning similarly to the original machine but with better cryptographic security. |
| Create a Command Line Interface (CLI) allowing basic interaction with the Enigma model | The user should be able to fully configure Enigma before either encoding or decoding a message. In addition, relevant information should be displayed to the user |
| Create a Graphical User Interface (GUI) allowing interactions with both Enigma and EnigmaPlus, providing a visualisation tool for both models | The user should be able to fully configure Enigma and EnigmaPlus before either encoding or decoding a message.  The visualisation tool should provide an abstracted and informative representation of both model’s encryption steps.  In addition, the design of the GUI should be aesthetically pleasing. |
| Support multiple platforms | Both the CLI and GUI should be compatible with Windows/MacOS/Linux |

Table Project requirements and success criteria

## Report Structure

This report consists of the following chapters:

* **Chapter** **1** introduces the project, providing the motivation behind it as well as presenting the aims and success criteria.
* **Chapter 2** provides background information regarding the Enigma, with a key focus on the mechanisms of the machine.
* **Chapter 3** provides a detailed account of the design and implementation throughout the project’s development.
* **Chapter 4** presents the finished product, as well as the results regarding accuracy of models. The quality of the solutions is then evaluated.
* **Chapter 5** concludes the outcomes of the project, as well as suggestions for further work.

# History and Background

This chapter provides an in-depth explanation of the components used in the Enigma. In addition, an abstraction of Enigma is presented to better explain how the machine functions.

The Enigma cypher is one of the most famous cypher machines [1] due to its role in WW2 and the work undertaken at Bletchley Park to create the Turing-Welchman Bombe, which helped crack the German codes. 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 [1]. The one used most in WW2 was *Enigma I* and was the key focus of this project.

The Enigma machine, had to be configured with exact settings in order for the intended parties to decode a message. These settings were distributed to the German forces each month in a code book and formed the cryptographic key for the Enigma cypher.

## Design of the Enigma

The Enigma Machine was a rotor-based machine and worked using a hybrid of mechanics and electrical signals. *Enigma I* contained 5 key components that worked together to produce cyphertext, they were: the keyboard, lampboard, the rotors, the reflector, and the plugboard. The inclusion of all these components aimed to make the machine as unpredictable as possible whilst also generating a large key-length of roughly 67 bits (nearly 159 quintillion different settings, see Equation 6) [2].

The use of the machine was a simple process. A user would receive a message along with a list of settings denoting the choice of rotors and their respective settings, the plugboard settings and a choice of reflector. Each time a key was pressed on the keyboard, a light on the lampboard would illuminate and the user would write down the corresponding letter. This functionality is demonstrated in Owen’s 3D animation of Enigma [3]. Due to the symmetric design of the Enigma machine, as long as two operators had the same settings, they could simply input the cyphertext into the machine and receive the plaintext and vice versa. Whilst this was an interesting aspect of Enigma, this design choice was overlooked and compromised the Enigma’s security (see section 2.3) [4].

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 [5]

### Keyboard and Lampboard

The keyboard and lampboard were the input and output modalities for Enigma. 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 [6]. 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 [3] 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) to ‘step’ the rotor [6]. 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 them, causing an electrical signal to be redirected 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 step. The right-most rotor would step every key press and the middle rotor would step with a period of 25 [7] [8]. This is due to a characteristic 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 [6]. The left-most rotor would step with a period of 262. This rotational property of the machine allowed for the final 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, operators were given a box containing 5 different rotors. The choice of which rotor to use formed part of the Enigma’s key. Table 2 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 "Enigma I" rotor encodings [9]

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, therefore allowing the notch position to move relative to the letter ring. Whilst, the ring setting formed part of the key, it is important to note that it had marginal impact on the strength of the cypher [10].

There were 1054560 different ways to configure the rotors in *Enigma I*, contributing significantly towards its complexity (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 where most of the letter scrambling takes place. 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 through a different metal contact.

The design of the reflector enabled the whole machine to be symmetrical, 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 reflector prevented the current being passed back through the same metal contact) were crucial flaws exploited [4] by the team at Bletchley Park to crack the code during WW2.

The Enigma could only use one reflector; however, the choice of reflector was interchangeable. Operators of Enigma were given a box containing a choice of three reflectors; however, the reflector being used was rarely changed. The three reflectors that were available with the *Enigma I* were UKW-A, UKW-B and UKW-C as shown in Table 3. The encodings demonstrate the inner wiring of each reflector, for example UKW-A maps A to the letter E.

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

Table "Enigma I" reflector encodings [9]

### 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 as demonstrated in Figure 2. This created yet another scrambling of the letter, such that if socket A and socket E were connected then any signal 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 [1]

## Enigma Machine and Abstraction

Often the best way to understand the mechanisms of Enigma is to observe the encryption process through an abstraction of the machine. This section abstracts the previously mentioned components, building a complete and simplified model of Enigma.

### Enigma’s Encryption

As mentioned above, the user begins with configuring the Enigma’s settings. Upon a keypress on the keyboard, the rightmost rotor will rotate. Depending on the current rotation, the other rotors may also step. Then, 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 left-most rotor 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 [11], such that: represents the plugboard, represents the rotor in the left, middle or right position and represents the reflector, an equation can be formulated to describe Enigma’s encryption steps, as seen in Equation 2. 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 be difficult to predict the outcome of an encryption.

Firstly, Figures 3-7 are inspired by work from Smart [8] and Thimbleby [4] 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, more precisely, the letter-to-letter mappings of each component. These diagrams are designed such that any letter is **input on the right** and the **output on the left** is . 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 that it better reflects the physical layout of the Enigma machine. It is important to note that, in the case of rotors, the function refers to the component in rest state (no rotational effects).

The plugboard acts as a simple cypher, either swapping the inputs and outputs of two letters or leaving them unchanged. 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 plugboard that are connected using a cable are represented with a connection between nodes such as A and D. Due to the plugboard symmetry, these diagrams exhibit ‘X’ shapes.

A diagram of lines and dots

Description automatically generated

Figure A wiring diagram to show a plugboards potential encoding (A-F)

The reflector acts as a substitution cypher, with additional constraints being self-coding (a letter cannot be encoded to itself) and reciprocal coding (if A B, then B A) (see Equation 3). Like other components this can be represented using a wiring diagram in which the input and output both take place on the right-hand side, such as Figure 4. This diagram demonstrates an example of a reflector, where A is shown to be connected to F and vice versa, showing 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 (A-F)

The rotors are substitution cyphers with no additional constraints. They take an input letter and produce either the same or a different letter. Whilst a rotor can be represented using a wiring diagram, multiple diagrams are needed to convey the rotor’s rotational effects. Figure 5 provides an example of the same rotor with different rotations, where 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 in Figure 5. 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 where represents the rotor’s rotation and represents the rotor’s ring setting. In addition, the output of the letter will be shifted by (see Equation 4). It is clear from these generalisations that if the rotation and ring setting were equal, then there would be no effect on a letter’s encryption as mentioned in 2.1.2.

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 (A-F). 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. Figures 3-5 represent each component of the machine and can be combined to create a representation of the entire Enigma machine such as Figures 6-7. In these diagrams, 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. 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 (A-F)

A diagram of a network

Description automatically generated

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

## Design Flaws and Remedies

Whilst the Enigma machine is complicated, the machine suffered from crucial design flaws which were exploited by the code breakers in WW2. This section explains these flaws, as well as providing solutions to rectify them.

### Cypher Strength

At first glance, it may seem like the Enigma machine is unbreakable and indeed the Germans shared this over-confidence [4]. The Germans became complacent when operating the machine often opting to use the same three rotors and neglecting to change the reflector [12]. This led to a large reduction in the security of communication between operators by factors that could have been largely avoided [4].

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 [10]. In fact, work from Tang, Lee and Russo [12] suggests that the Enigma had a theoretical key space of compared to the more realistic demonstrated in Equation 6.

As suggested by Thimbleby [4], 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 section 2.1, 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 flaws of the reflector: self-coding and reciprocal coding [13] [4].

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 [12].

### 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 used 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 [13] 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 [13]. Ostwald’s study [13] shows numerous additional improvements that either were implemented or could have been implemented to improve the cypher key space.

Thimbleby [4] gives examples of multiple circuits that aimed to fix the two main weaknesses of the Enigma machine mentioned in section 2.3.1, 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 the self-coding and reciprocal coding weaknesses 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 without re-introducing the weaknesses. The removal of the reflector does reduce the key space however, this can be mitigated by simply adding another rotor in its place.

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 [4]

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 those in Figures 6-7, a logical model for this machine can be created as shown in Figure 9. This logical model formed the basis of EnigmaPlus.



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

## Related Work

Several similar tools 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, the paper written by Thimbleby [4] gave direct inspiration into the creation of “EnigmaPlus”.

“Enigma Machine Emulator” [14] 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 2D projection that resembles the real machine.

“The Enigma Machine” [15] 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 modern and simpler way to interact with it. Much of the design aspects of this projects were inspired by this webpage.

“Virtual Enigma” [16] offers extremely immersive, 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 an almost life-like simulation.

“The Enigma Machine” [17] was a similar product found 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 product was the inspiration for the visualiser functionality for this project.

# Design and Implementation

The principal goal of this project was to create an exact simulation of the Enigma machine and provide both a command line interface (CLI) and a graphical user interface (GUI) with high interactivity, thereby educating the user on the machine’s functionality. An additional goal of this work was to develop another model named EnigmaPlus based on the design in Figure 9 from the previous chapter, providing a stronger cypher than the original machine, whilst maintaining compatibility with the GUI.

This chapter provides an overview of the development in the project, as well as providing a detailed explanation of the design choices made for each aspect. Figure 10 illustrates the most important Java packages of the underlying system for this project whereas as Table 4 provides an exhaustive list of packages created during development.

The Enigma package contains all the necessary backend functionality to accurately simulate the Enigma machine and EnigmaPlus. Both models encrypt any plaintext message into cyphertext and vice versa. The Enigma package also provides pre-built rotor and reflector configurations to the Parsers package. The Parsers package and Config package then both provide custom component creation for both the CLI and GUI. In addition, Parsers and Config provide a means of configuring the Enigma model 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. Finally, the GUI provides numerous additional tools aimed at providing an informative representation of the encryption process through visualisation.

A diagram of a computer

Description automatically generated

Figure A diagram depicting the underlying packages for of this project, including Enigma, GUI, CLI, Config and Parsers

|  |  |  |
| --- | --- | --- |
| Java Package | Purpose | Relevant Files |
| Enigma | Encapsulates all components related to simulating the Enigma and EnigmaPlus models. | Enigma, EnigmaPlus, RotorMachineBase, Rotor, Plugboard, Reflector, ReflectorFactory, RotorFactory, EnigmaLogger, EngimaPlusLogger,  TranslationDirection. |
| GUI | Contains all necessary functionality to provide the graphical user interface and visualisation tool | EnigmaSimulatorApp, EnigmaController, EnigmaVisualiser, EnigmaPlusVisualiser. |
| CLI | Contains all necessary functionality to provide the command line interface | EnigmaSimulatorCLI. |
| Parsers | Contains numerous parsers to parse and store representations of the contents of the **Config** package (custom components and enigma settings). | ComponentCache, EnigmaSettingsParser, ReflectorBankParser, RotorBankParser. |
| Config | Allows the user to specify custom components and configure settings for the command line interface | enigma\_settings.xml, reflector\_bank.xml, rotor\_bank.xml. |
| Launcher | Contains a launcher class used to run both the CLI and GUI | Launcher. |
| Test | Provides a suite of tests to test the functionality of the above packages | ComponentCacheTest, EnigmaTest, EnigmaPlusTest, PlugboardTest, ReflectorFactoryTest, ReflectorTest, RotorFactoryTest, RotorTest. |

Table An exhaustive list of the packages developed for this project

## Tools and Technology

Throughout the development of this project, multiple tools and technologies were utilised to accomplish the requirements and objectives.

Java [18] was the chosen programming language used for this project due to its native use of object-oriented programming (OOP) and its extensive online support. The first of two Java packages used was JavaFX [19], which is a graphics package specialising in user interfaces. This 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 [20] which is a testing framework that enables test cases to be created and run 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 [21] is an external application that provides a “drag and drop” experience for GUI creation. This was used within the project to create prototypes for, and eventually implement, the GUI. The tool was chosen because it generates FXML files which are natively supported by JavaFX, creating a simpler GUI development.

The XML language is an object description language with the sole purpose of representing data. In the scope of the 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 [22] is a Java based build tool which aims to create a simpler process for the developer to build their Java project. Maven was used here to build the project as well as manage all other dependencies (such as JavaFX) to help maintain a minimal code base.

GitHub [13] is web-based platform tailored for version control using Git [23]. Both GitHub and Git were used during this project for the planning and tracking of development progress. In addition, in some cases additional branches were used for the implementation of larger features such as the GUI.

## Enigma

The Enigma package consists of four core classes: Enigma, Rotor, Reflector, Plugboard. These classes, and their respective attributes, are designed to properly reflect the components of the physical machine as demonstrated by the UML diagram in Figure 11. In addition, the diagram also demonstrates that the Rotor, Reflector and Plugboard class all hold a composite relationship with Enigma. It is important to note that Figure 11 does not include any mention of the keyboard or lampboard covered in the background chapter. This is because whilst they were significant components for the physical machine, in a logical sense, they are nothing more than input and output.

The components within Enigma are variations of substitution cyphers, therefore a common representation was needed for this. 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, this would have required 52 key-value pairs in the hash-map. In addition, hash-maps bring unnecessary complexity into the representation. Instead, a simple array for each component depicted as wiring, of length 26 integers, was used for the representation. In such an array, each index represents the positional encoding of a plaintext letter and the value at that index represents the positional encoding of the cyphertext letter. This method offers greater simplicity as well as maintaining fast indexing, a process that is abundant in an encryption. For the plugboard and reflector, this representation was enough, however the rotor required a more complex representation to account for its unique properties. To achieve this, additional attributes were included in the model: reverseWiring, currentRotation, turnoverPosition and ringSetting.

In addition to the components and the machine, the Enigma package contains two factory classes: RotorFactory and ReflectorFactory. These were implemented to encapsulate all rotor and reflector instance creation. This includes creation of pre-set components, as shown in Table 2 and Table 3 from chapter 2, as well as the 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 has the most constraints.

A diagram of a computer

Description automatically generated

Figure UML diagram depicting the Enigma package

The encryption process is based on two key functions: 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 encryption process for a single character is shown in Figure 12. The flowchart performs the rotation of the rotors, the pseudocode of which is presented as pseudocode in Code Block 1, before subsequently invoking the encode function of each component (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 A flowchart depicting the Enigma model’s 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 The Enigma's rotation mechanism demonstrated by pseudocode

### Additional Features

In addition to the models developed in this project, numerous classes were added to the Enigma package. During the development of the CLI and GUI, these features provided interesting information regarding each model.

A logger class was implemented to log Enigma. The Enigma model utilises this class to provide a step-by-step outline of the encryption/decryption process. The logger also provides 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.

Additional functionality was implemented for the Enigma model which generates a list of all wiring connections in the model at any given frame by using enumeration, where each letter of the alphabet is encrypted (without rotation applied) 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 contains three files, rotor\_bank.xml, reflector\_bank.xml and enigma\_settings.xml which allows the user to configure the settings for the Enigma machine. The Parsers package contains several XML parsers to read the contents of the files 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, an additional system allowing the user to configure the machine beyond the capabilities of the original machine was developed to provide a more explorative experience. Both rotor\_bank.xml and reflector\_bank.xml were created to allow the user to define their own corresponding components much like the definitions shown in the previously mentioned Table 2 and Table 3. Code Block 2 shows an example of an entry a user could create in rotor\_bank.xml, producing a new rotor called “MyCustomRotor”. At runtime of either the CLI or GUI application, the Parsers package creates 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 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 enigma\_settings.xml contents depicting the start settings of the machine

## Command Line Interface

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. Therefore, 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 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.

When the system is run, the Enigma settings are displayed to the user, to ensure that they understand the settings that were used to encrypt their message. Then, the user is prompted to enter a message before the correct cyphertext (or plaintext if this is a decryption) is displayed. Code Block 4 shows an example of the expected interaction with the CLI, showing the chosen settings and 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. The functionality to accomplish the requirements of the CLI was achieved by creating the EnigmaSimulatorCLI runnable Java class.

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 with the CLI

## Graphical User Interface

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.

### GUI Design

As explained in section 2.4, numerous other products exist that attempt to apply skeuomorphic designs to their simulation. The UI for this project mostly avoids reflecting a physical representation of the machine and adopts a simpler approach. This is because the Enigma machine’s layout is rather obscure. Figure 13 shows an early mock-up of the design for the GUI, which follows a top-to-bottom approach for user interaction. The idea behind this design was that the user configures the reflector, rotors and plugboard before entering their message on an on-screen keyboard, displaying the cyphertext message below as they entered their message.

The rotors and the reflector are selected by drop-down menus, allowing the user to select one of the rotors stored within ComponentCache (both custom and pre-configured components). In order to stick with the simple design of the interface, the plugboard consisted of a text field in which the user enters cable pairs such as “AB CG OI” to represent the connections. A few different variations of input modalities were considered for each component in the GUI and Table 2 provides a comparison of the considerations.

A screenshot of a computer

Description automatically generated

Figure An early mock-up design for the GUI

|  |  |  |  |
| --- | --- | --- | --- |
| Input/Output Modality | Description | Benefits | Drawbacks |
| Clickable plugboard | Much like the online Enigma Emulator [14], the user clicks a graphical representation of the plugboard to create connections between letters | Authentic to the physical machine.  Interactive.  Restricts user input, stopping erroneous plugboard configurations. | Slower and tedious to enter plugboard configuration.  Harder implementation.  Removes ability to copy-paste plugboard settings. |
| On-screen keyboard | Instead of using the keyboard on the user’s device, an on-screen keyboard is displayed within the GUI in which the user clicks to enter text | Authentic to the physical machine. | Removes the ability to copy-paste messages.  Slower to encrypt longer messages. |
| Interactable rotors | Much like the Virtual Enigma [16], the rotors act as physical objects than can be rotated by the mouse | Authentic to the physical machine  Interactive. | Cumbersome to interact with  Difficult animations required to create a sense of rotation. |

Table Considered input modalities that were not implemented into the GUI

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 14 shows a prototype, which exhibits several differences to the original design. The biggest difference is the message input modality, which was changed from an on-screen keyboard to a simple text field. Whilst trying to build this prototype, it was decided the previously proposed on-screen keyboard created too much clutter in the application as well as provide a significant 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 A high-fidelity prototype of the user interface provided with the GUI

### Model-View-Controller Architecture

In order to create clean and easily modifiable code, the GUI follows the model-view-controller (MVC) architecture. This architecture aims to separate a graphical application’s code into three components: model, view, and controller. In the case of this project, the model is represented by the classes in the Enigma package. For the view, Gluon Scene Builder combined with minimal code provided FXML files to define the GUI (EnigmaSimulatorApp). Finally, an additional class was created within the GUI package known as EnigmaController, allowing the user to communicate with the model, thereby acting as the controller. Figure 15 provides a diagram to show how the MVC architecture has been applied to this project.

A diagram of a software system

Description automatically generated

Figure 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 about the workings of Enigma. As demonstrated in section 2.2.2, the simplest way to understand the Enigma is through abstractions, often taking the form of wiring diagrams. Therefore, the visualisation tool was developed and implemented into the GUI which generates wiring diagrams (see Figure 7) for each frame of the user’s encryptions.

The first task to complete was to integrate the visualisation tool into the existing GUI. To accomplish this, an additional tab was added to the bottom of the interface. Figure 16 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 A high-fidelity prototype of the GUI including the “Visualisation” tab

In order to draw a diagram programmatically with various shapes, a JavaFX canvas object was 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, an additional file called EnigmaVisualiser was created to control all aspects of the visualisation. The code within this class consisted mostly of geometric calculations to generate a legible and informative diagram.

The generated wiring diagrams are designed to represent Enigma in the same way as the diagrams shown in chapter 2, but now including the entire 26 letter alphabet. Figure 17 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 the information 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 3.2.1), however this is not shown in the figure. The actual results of this visualisation tool and further discussion are expanded in chapter 4.

A diagram of a graph

Description automatically generated

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

## EnigmaPlus

In order to accommodate EnigmaPlus, both the Enigma package and the GUI mentioned above required several changes to reflect the model. This section provides a summary of the changes made to each system.

### Enigma Package

The differences between EnigmaPlus and Enigma are small, but important. Therefore, the class hierarchy within the Enigma package was readjusted to better accommodate EnigmaPlus. Figure 18 shows the UML diagram of the readjusted Enigma package. The key difference is that both the Enigma and the newly introduced EnigmaPlus class both inherit from a base class named RotorMachineBase. This change helped reduce the amount of duplicated code in the package, ensured the two models work in similar fashions and allowed the additional features implemented for Enigma to also be used for EnigmaPlus. The restructuring of the Enigma package was implemented with relative ease, due to the encapsulated functionality of each component.

A diagram of a machine

Description automatically generated

Figure The final UML diagram of the Enigma package, including EnigmaPlus

The encryption process of EnigmaPlus works differently, where encoding and decoding are two distinct operations in which one is the inverse of the other. Figure 19 shows a flowchart depicting both the encode and decode function for EnigmaPlus. It can be noted that the decode function applies each components letter scrambling in the opposite order.

A diagram of a character

Description automatically generated

Figure A flowchart depicting the encryption/decryption steps of EnigmaPlus

### Graphical User Interface

To incorporate the EnigmaPlus model into the GUI, an additional tab was added to the top of the interface. This allowed the user to switch between which model they wanted to use. By selecting EnigmaPlus, the user is presented with a slightly different interface to configure the machine. Figure 20 shows a high-fidelity prototype of the settings options when EnigmaPlus is selected. The key differences between these options and the ones displayed in Figure 14 is that the user is unable to select a rotor, and there is an additional option select encode or decode. This settings panel was changed so that the interface better reflects the settings for EnigmaPlus.A screenshot of a computer

Description automatically generated

Figure The configuration settings for EnigmaPlus within the GUI

The visualisation aspect of the GUI was also required to support EnigmaPlus, so the EnigmaPlusVisualiser class was created to handle all diagram generation for this model, much like EnigmaVisualiser mentioned earlier.

The wiring diagrams for EnigmaPlus are similar to Enigma, however they do not make use of the reflector component and encoding and decoding take place in opposite directions as demonstrated by Figure 9 from section 2.3.2. Figure 21 shows a mock-up wiring diagram to demonstrate how a generated EnigmaPlus wiring diagram should look. This diagram shows “A” being encoded to “M” (right-to-left). In the case of decoding, the line should be represented in blue to represent the current flowing from left-to-right.

A diagram of a graph

Description automatically generated

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

## Model Tests

During implementation of both the Enigma and EnigmaPlus models, a suite of unit tests was developed to validate the inputs and outputs of the models. For both models, ensuring that: a message can be encoded into cyphertext decoded back to the original message was essential. In addition, both models were required to be fully configurable.

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, the output cyphertext generated by similar products was used for comparison to ensure accuracy due to the difficulty of gaining access to the real-world machine. After performing tests on similar products [14] [15], it was found that once configured in the same way (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 6 shows an extract of a test plan that was developed and followed in order to meet the requirements of the project. The tests demonstrated are an extract of the tests included in two Java test classes named EnigmaTest and EnigmaPlusTest. It is important to note that for EnigmaPlus, the message used to test the encryption was specially chosen in order to validate that self-coding and reciprocal coding were not present. To ensure an accurate simulation of Enigma, a pass was required in all tests. 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. The results of these tests and other metrics are presented and evaluated in chapter 4.

|  |  |  |  |
| --- | --- | --- | --- |
| 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/key | 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 result of the decryption |
| 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 given the same key |
| 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 A snippet of a test plan for validation of the Enigma and EnigmaPlus models

# Results and Evaluation

This chapter presents and evaluates the results of the project focussing on the accuracy of both Enigma models as well as demonstrating the finished product of both the CLI and GUI.

## Model Accuracy

Both the Enigma and EnigmaPlus models were required to be fully functional, such that a message could be encoded and decoded using the same key. In the case of Enigma, the model also had to reflect the functionality of the real-world machine as best as possible.

The Enigma and EnigmaPlus models were developed alongside unit tests in order to validate their accuracy. This suite consisted of 20 different tests which were based on examples from similar products. Both models were developed such that all 20 tests passed, ensuring that the functional requirements (demonstrated in the test plan within Table 6) of both models were met. Figure 22 shows the test coverage metrics generated by Maven, providing an indication of the robustness of the models, notably Enigma and EnigmaPlus both exhibit code coverage scores over 75%.

A screenshot of a graph

Description automatically generated

Figure A table demonstrating the test coverage of the Enigma package

### Enigma Authenticity

As explained in section 3.7 an example encryption used in the testing was the input string “AAAAA” which was encoded into “BDZGO”. Figure 23 shows a collection of screenshots demonstrating the example encryption taking place whilst using the Cryptii Enigma web application [1]. Figure 24 provides the input and output of the same encryption taking place using the Enigma Emulator [14]. These two examples were then compared to the output produced by the CLI (see Figure 25 below) showing that the output of the Enigma model developed in this project produces the same output.

A screenshot of a computer

Description automatically generatedA close-up of a computer screen

Description automatically generatedA close-up of a computer screen

Description automatically generated

Figure A collection of screenshots to show the “AAAAA” encryption using the Enigma model on Cryptii [1]

A book with text on it

Description automatically generated

Figure A screenshot of the Enigma Emulator [14] to show the results of the “AAAAA” encryption

A screenshot of a computer program

Description automatically generated

Figure A screenshot of the CLI output showing the results of the "AAAAA" encryption

In addition to comparing similar products, a real, contemporary Enigma message was used from a collection of different messages [24]. Due to the limited number of messages available, only one could be used for comparison. Figure 26 shows screenshots from the web page showing the message, and corresponding settings Enigma settings. Figure 27 shows a screenshot of the decryption of the message within the developed CLI. Due to arbitrary shorthand and typos, the message looks illegible, however with some additional processing, the message translates to “**FEINDLICHE INFANTERIE KOLONNE BEOBACHTET. ANFANG SUEDAUSGANG BAERWALDE ENDE DREIKM OSTWAERTS NEUSTADT**” which again translates to English as “**ENEMY INFANTRY COLUMN OBSERVED. BEGINNING SOUTH EXIT BAERWALDE END THREE KILOMETERS EAST NEUSTADT**” [25].

A white rectangular box with black text

Description automatically generated.

A black and white text

Description automatically generated

Figure A genuine message encrypted by Enigma, taken from a collection of contemporary messages [24]

A screenshot of a computer

Description automatically generated

Figure A screenshot of the CLI depicting the decryption of a genuine Enigma message

### Model Requirement Evaluation

Both Enigma and EnigmaPlus work as intended, allowing encryption and decryption of messages when using the correct key. In terms of features, the models are complete, enabling the user to fully configure the machine beyond the capabilities of the physical machine. In addition, code coverage results presented in this section demonstrate that both models are well tested and robust.

In the case of Enigma, authenticity was a key deliverable. Whilst direct access to the physical machine was not possible for comparison, numerous other Enigma simulations provided a baseline for comparison. It is clear from the results that this project provides the same output as all other similar products. In addition, the fact that genuine German messages can be decrypted using the simulator significantly increases the confidence that the developed model is authentic.

For EnigmaPlus, the requirement was to provide a similar experience to Enigma whilst also improving the strength of the cypher. The developed model provides a very similar experience to Enigma by using the same rotor and plugboard functionality of the original machine. Whilst EnigmaPlus does have a smaller key size than the Enigma model due to the lack of a reflector choice, the model does tackle the inherent design flaws of the Enigma model explained in section 2.3.

## CLI, GUI and Visualisation

This section presents the results of the CLI, GUI and visualisation tool. Within each section, an evaluation is provided to assess the success of each implementation.

### CLI

The CLI developed for this project was designed to be simple and quick to use. As such the finished product of the CLI is close to its original design demonstrated in section 3.4. The configuration of the machine takes place within the previously mentioned enigma\_settings.xml and the output of the CLI can be seen in Figure 27 and Figure 25 shown above.

### GUI and Visualisation

The GUI allows interactions with both Enigma and EnigmaPlus, ensuring the user can operate and configure both machines beyond their physical capabilities (see Figure 31 below, which includes “MyCustomRotor” in the drop-down menu). In addition, the application supports Windows, MacOS and Linux machines, as well as providing a visualisation tool to demonstrate the encryption process of each model. The results shown in this section follow the original design closely, with some minor additions. Most notably, the application is skinned to have a more modern aesthetic. Secondly, an additional feature has been implemented to allow the user to automatically step through the encryption, providing an animation of the encryption. Finally, a label called “Machine Permutations” is present, which indicates the cryptographic strength of the chosen model to the user.

Figure 28 shows an example of the GUI being used to encrypt a message displaying the visualisation at the bottom. In this particular example, the visualisation only shows the wire through which the current has passed as explained in Figure 17. This can be compared to Figure 29, which shows the same generated wiring diagram, but with all other wires also being visualised. These additional wires can be shown and hidden at any time by selecting the “Show all wires” check box. Interaction with the EnigmaPlus model is done by clicking the “EnigmaPlus” tab at the top of the screen and then proceeding as usual. Figure 30 shows an example of an encryption taking place using EnigmaPlus.

A screenshot of a computer

Description automatically generated

Figure An example of the developed GUI being used for an Enigma encryption (Windows)

A screenshot of a graph

Description automatically generated

Figure A generated wiring diagram, showing all wire configurations in a given frame

A screenshot of a computer

Description automatically generated

Figure An example of the developed GUI being used to perform an EnigmaPlus encryption (Windows)

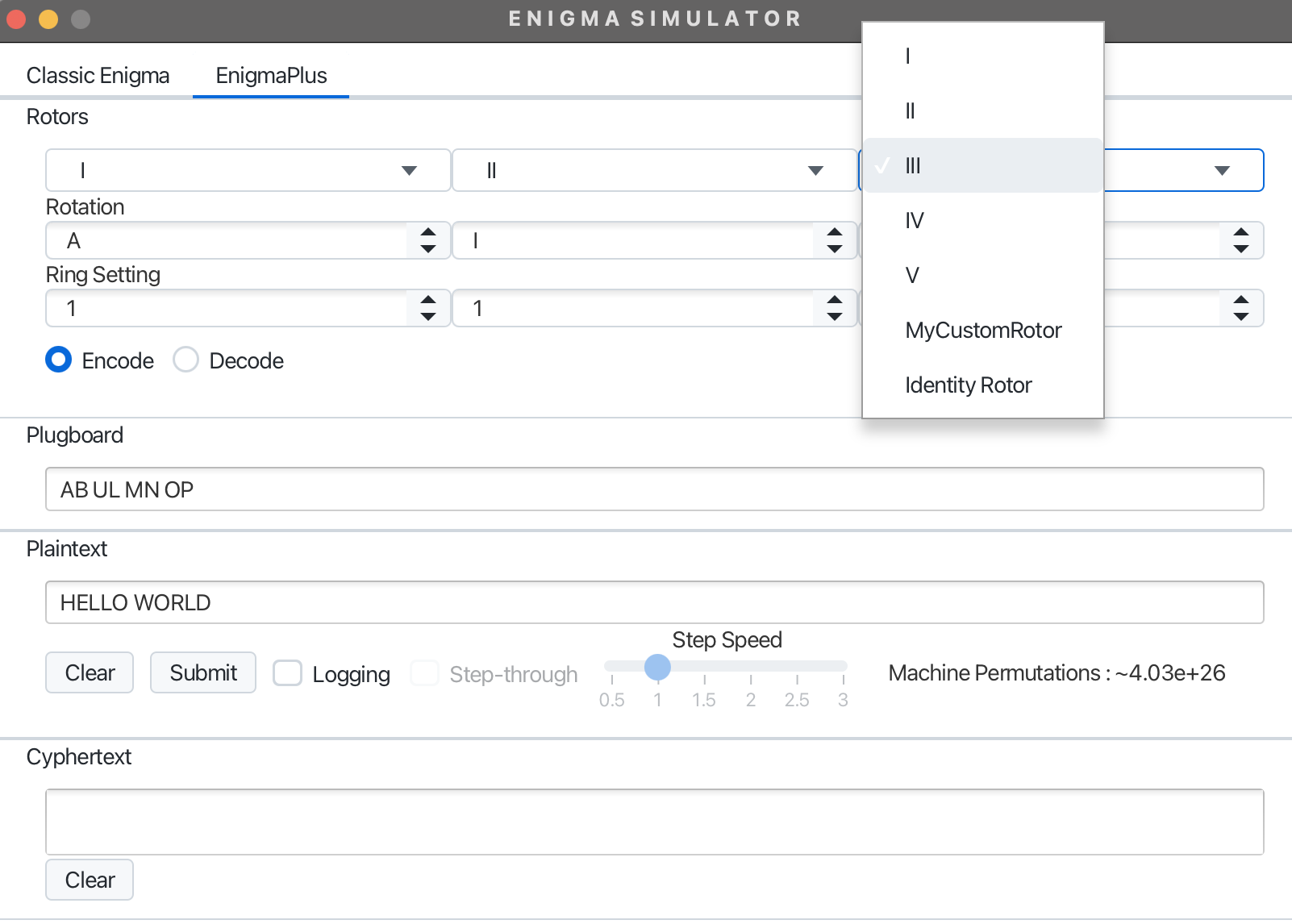


Figure A screenshot of the UI demonstrating rotor selection (MacOS)

### Interface Evaluation

The CLI provides an easy experience to configure the machine through the user of the Config package (enigma\_settings.xml). The GUI successfully implements various familiar interface components such as drop-down menus to enable the user to configure and operate both Enigma and EnigmaPlus, accurately depicting the differences in each model. In addition, both the CLI and GUI are compatible with multiple different platforms, as demonstrated in the results (Windows and MacOS).

In terms of aesthetics and usability, the CLI provides a simple and easy to use tool. Both configuration and operation of the Enigma model can be done with ease. In addition, by using the enigma\_settings.xml file, user’s chosen settings can easily be copied and sent to anther user for secret communications, much like the original machine.

The GUI exhibits a sleek aesthetic, offering a simpler experience than operating the physical Enigma machine. The design of the GUI follows a top-to-bottom approach, creating a natural and familiar environment for the user. Whilst Enigma and EnigmaPlus are different, the GUI successfully displays the differences whilst maintaining the similarities.

The visualisation provided alongside the GUI offers an accurate, real-time abstraction of the encryption process taking place. As mentioned in section 2.2.2, by abstracting the machine into wiring diagrams, the mechanism behind the Enigma becomes much clearer. Therefore, the visualisation tool brings significant value to a user wanting to understand how the machine works.

## Summary of Results

The project as a whole provides a complete Enigma experience, encouraging exploration through the use of EnigmaPlus and custom components to aid the learning of the user. The interfaces by which the user can interact with the models are distinct and accurately reflect Enigma.

The criteria of success for each aim and objective proposed for this project was met using whilst maintaining high-quality code and using modern design practises. The original specification for this project was simpler than the requirements in Table 1, and therefore final product of this project has exceeded the original expectations.

# Conclusion

This chapter provides a conclusion to the project, a summary of the project, reflections on the work focussing on what could be done better, and suggestions of further work surrounding this topic.

## Summary

In this project, thorough research was undertaken to accurately recreate the Enigma in Java. In addition to this, further research was used to create an improved version of Enigma called “EnigmaPlus” which aimed to fix the inherent cryptographic flaws within the original Enigma machine. Finally, two interfaces were provided allowing interactions with both models as well as providing an abstracted visualisation of the encryption process. The artefact produced during the project was planned and tracked through GitHub (<https://github.com/elliot-brooks/enigma-simulator>) and engineered to meet high standards by adopting modern software engineering practices such as OOP, MVC architecture and well-tested code. In addition, the GitHub page contains a README file providing a brief overview of the project and instructions on how to run the program.

The original specification for this project only required a configurable simulation of the Enigma machine along with a simple CLI or GUI. Due to the personal interest that developed during this project, additional goals were established, providing unique aspects to the project, such as EnigmaPlus, the visualisation tool and enabling further configuration to Enigma. Therefore, not only did the project meet the specification, but it also exceeded the anticipated results.

On a personal note, the project has also been a very significant learning experience, further developing my engineering skills as well as being one of the biggest research projects I have undertaken.

## Reflection

There are still several improvements that could be made to the existing product. This section a discussion about the limitations as well as potential improvements.

### Improving The model

As mentioned in the beginning of this report, the developed model reflects *Enigma I*, which is only one machine belonging to the Enigma brand. Many of the machines inspired by *Enigma I*included more complicated rotor mechanisms and additional rotor slots. The model could be improved by allowing the user to select an arbitrary number of rotors to be used in an encryption, or even extending the alphabet of the Enigma model to include special or numeric characters.

### Improving The Interfaces

Whilst both the GUI and CLI are easy to use and allow complete interactions with Enigma, improvements could be made to make the interfaces more engaging and authentic. For example, the GUI could include more skeuomorphic elements to better reflect the physical machine, whilst still retaining the contemporary design developed during this project.

Quality of life improvements to the interfaces could include allowing the user to export the settings of Enigma as a text file in the style of the original WW2 settings sheets or allowing the user to export the visualisation animation as a .gif file for use in educational materials such as presentations.

### An Educational Perspective

During the development of this project, elevating the educational value of the product above the existing products was a key aim. The developed solution to the specification provides a contemporary and interactive experience with Enigma, allowing the user to operate the machine and explore the otherwise hidden aspects by using custom components. Clear examples here are the in the ability to switch between Enigma and EnigmaPlus, displaying the cryptographic strength of each model so that the user may understand the consequences of particular configurations. Overall, I believe that this provides an enriching experience for the user, without over complicating the CLI and GUI.

To further support the user’s learning, the project could provide further background. In its current form, the project is likely to serve those who already have some knowledge of the physical machine or the similar products and want to learn more. To improve the learning experience and make the product accessible to a wider audience, additional materials could be included alongside the project, for example in-application tutorials or guides. These materials could be aimed at multiple demographics.

## Ideas for Future Work

Aside from the improvements explained above, there are several ways this project can be extended. Much like the skeuomorphic designs of similar products, the Enigma package within this project could be used for another project attempting to provide a 3D application in which the user can interact with Enigma and EnigmaPlus.

Cryptography has progressed significantly since WW2 and so a similar project could be developed to attempt to enhance Enigma even further than EnigmaPlus. This can be done by opting to use modern encryption techniques such as asymmetric encryption, a technique used extensively for secure communication over the internet.

Substantial future work is also naturally suggested by the events surrounding Enigma. For example, a future project could investigate the breaking of Enigma, providing an algorithm to break the Enigma or even simulate the Turing-Welchman Bombe.

# References

|  |  |
| --- | --- |
| [1] | “Enigma,” [Online]. Available: https://www.cryptomuseum.com/crypto/enigma/. [Accessed 2024]. |
| [2] | D. J. van Manen and J. O. Robertsson, “Codes and ciphers.,” *Part I: Decoding nature’s disorder: GeoExpro 13,* pp. 38-41, 2016. |
| [3] | J. Owen, *How did the Enigma Machine work?,* 2021. |
| [4] | H. Thimbleby, “Human factors and missed solutions to Enigma design weaknesses.,” *Cryptologia 40, no. 2,* pp. 177-202, 2016. |
| [5] | “Science Museum,” [Online]. Available: https://www.sciencemuseum.org.uk/home. [Accessed 2024]. |
| [6] | D. H. Hamer, “Enigma: Actions involved in the ‘double stepping’of the middle rotor.,” *Cryptologia 21, no. 1,* pp. 47-50, 1997. |
| [7] | J. Grime, “Maths from the talk "Alan Turing and the Enigma Machine",” 2013. |
| [8] | N. P. Smart, “The enigma machine,” *Cryptography Made Simple,* pp. 133-161, 2016. |
| [9] | “Enigma wiring,” [Online]. Available: https://www.cryptomuseum.com/crypto/enigma/wiring.htm. [Accessed 2024]. |
| [10] | “How does the Enigma machine work?,” [Online]. Available: https://www.cryptomuseum.com/crypto/enigma/working.htm. [Accessed 2024]. |
| [11] | M. Rejewski, “An application of the theory of permutations in breaking the Enigma cipher,” *Applicationes Mathematicae. 16, No. 4,,* pp. 543-559, 1980. |
| [12] | L. Tang, N. Lee and S. Russo, “Breaking Enigma,” 2018. |
| [13] | O. Ostwald, “Cryptographic design flaws of early Enigma,” 2023. |
| [14] | “Enigma Machine Emulator,” [Online]. Available: https://www.101computing.net/enigma-machine-emulator/. [Accessed 2024]. |
| [15] | “Enigma Machine,” [Online]. Available: https://cryptii.com/pipes/enigma-machine. |
| [16] | “Virtual Enigma,” [Online]. Available: https://enigma.virtualcolossus.co.uk/VirtualEnigma/. [Accessed 2024]. |
| [17] | T. MacWright, “Enigma Machine,” [Online]. Available: https://observablehq.com/@tmcw/enigma-machine. [Accessed 2024]. |
| [18] | “Java,” Oracle, 2024. [Online]. Available: https://www.java.com/en/. [Accessed 2024]. |
| [19] | “JavaFX,” [Online]. Available: https://openjfx.io. [Accessed 2024]. |
| [20] | “JUnit 5,” JUnit, [Online]. Available: https://junit.org/junit5/. [Accessed 2024]. |
| [21] | “Gluon Scene Builder,” Gluon, [Online]. Available: https://gluonhq.com/products/scene-builder/. [Accessed 2024]. |
| [22] | “Apache Maven Project,” Apache, 2024. [Online]. Available: https://maven.apache.org. [Accessed 2024]. |
| [23] | “Git,” [Online]. Available: https://git-scm.com. [Accessed 2024]. |
| [24] | “Enigma/Sample Messages,” [Online]. Available: http://wiki.franklinheath.co.uk/index.php/Enigma/Sample\_Messages. [Accessed 2024]. |
| [25] | “Enigma/Sample Decrypts,” [Online]. Available: http://wiki.franklinheath.co.uk/index.php/Enigma/Sample\_Decrypts. [Accessed 2024]. |
| [26] | “GitHub,” [Online]. Available: https://github.com. [Accessed 2024]. |